DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
53

PILLE PULLONEN-RAUDVERE

Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

53

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
53

PILLE PULLONEN-RAUDVERE

Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in informatics on May 31, 2024 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Assoc. Prof. DSc.	Sven Laur University of Tartu Institute of Computer Science Tartu, Estonia
PhD.	Dan Bogdanov Cybernetica AS Tartu, Estonia

Opponents

Assoc. Prof. PhD.	Mayank Varia Boston University Faculty of Computing & Data Sciences Boston, United States of America
Assoc. Prof. PhD.	Bernardo Machado David IT University of Copenhagen Department of Computer Science Copenhagen, Denmark

The public defence will take place on August 13, 2024 at 13:00 in Narva mnt. 18–1021.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (pdf)
ISBN 978-9916-27-590-0 (print) ISBN 978-9916-27-591-7 (pdf)

University of Tartu Press
http://www.tyk.ee/

"Perhaps having the courage to find a better path is having the courage to risk making new mistakes."
– Robin Hobb, *Golden Fool*

ABSTRACT

Secure multiparty computation is a method for computing on private inputs without disclosing anything but the computation outcome. There exist various protocols for such computations such as homomorphic encryption, secret sharing or garbled circuits as well as methods specialised for a specific computation task like information retrieval. This thesis focuses on programmable secure multiparty computation where any algorithms can be executed. Furthermore, some outputs may be published and used to choose the following computations. For example, a result may be published and used as a condition to determine the next step.

In such a case, it is critical to precisely define the security of computations. Sometimes secure programs executing some algorithm publish more values than one would usually consider as the output of such algorithm. In this case, it is important to analyse that the protocol does not reveal more information about the private inputs by publishing these values. This thesis focuses on simplifying such proofs for two separate cases.

First is the case where the protocol outputs do not reveal information about the inputs. If this happens, the commonly used security definitions are stronger than necessary. Instead, the thesis defines an input privacy property sufficient for such protocols. However, in many cases, input-private protocols can be extended to secure protocols.

The second is the case where new complex algorithms are implemented using existing smaller secure computation functionalities as building blocks. For example, basic arithmetic operations are already defined and then used to implement some statistic tests. In existing formal frameworks, to prove the security of such algorithms, the prover has to work with many different details. This thesis derives an abstract execution model for secure multiparty computation protocols that can be used instead. In the abstract model, the prover can focus on the public values and the basic flow of the protocol execution. Under some natural restrictions about the secure computation framework and the protocol, the proof in the abstract model implies that security holds also in the standard model.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ABB	Arithmetic black-box
CCA	Chosen ciphertext attack
MPC	Secure multiparty computation
OT	Oblivious transfer
RSIM	Reactive simulatability
SUC	Simpler universal composability
UC	Universal composability
$[\![x]\!]$	Secret sharing of value x
$[\![\mathbf{x}]\!]$	Secret sharing of a vector $\mathbf{x}$
Env	Environment
Env'	Component of the environment giving inputs in an input privacy configuration
$\mathsf{Env}'_{\perp}$	Component of the environment receiving outputs in an input privacy configuration
$\mathbb{E}$	Class of environments
$\mathbb{E}_{\mathsf{pd}}$	Class of environments against the protection domain
A	Adversary
$\mathbb{A}$	Class of adversaries
$\mathbb{A}_{\mathsf{c}}$	Class of coherent adversaries
Sim	Simulator
Π	Protocol
Π_{e}	Inner environment representing computations in the secure computation framework
$\mathbb{P}$	Class of protocols
Sys	System
$\mathcal{P}_i$	Protocol participant i
$\mathcal{I}_i$	Code interpreter of $\mathcal{P}_i$
$\mathcal{Z}_i$	Corruption manager for $\mathcal{P}_i$
$\mathcal{F}_{\triangle}$	Secure setup functionality
$\mathcal{F}_p$	Ideal functionality p
$\mathcal{F}_{\mathsf{pd}}$	Ideal functionality of a protection domain
$\mathcal{F}_{\mathrm{io}}$	Input-output functionality
$\mathcal{G}_{p,i}$	Local functionality p for $\mathcal{P}_i$
$\mathcal{G}_p^0$	Local functionality p on values
$\mathcal{S}$	Secret sharing functionality
$\mathcal{R}$	Reconstruction functionality
$\mathcal{M}$	Memory
$\mathcal{T}_{\mathcal{S}}$	Machine inside $\mathcal{F}_p$ that manages timing of $\mathcal{S}$
$\mathcal{T}_{\mathcal{R}}$	Machine inside $\mathcal{F}_p$ that manages timing of $\mathcal{R}$

$\mathcal{T}_{\mathcal{M}}$	Machine inside $\mathcal{F}_p$ that manages access to memory $\mathcal{M}$
$\mathcal{E}$	Extractor
$\mathcal{L}$	Storage of values in a storage domain
$\mathcal{L}^*$	Storage of shares in a storage domain
$\mathcal{M}_0$	Memory holding only values
$\mathfrak{s}$	Internal state of the protocol participant
$\mathfrak{gs}$	Global state combining $\mathfrak{s}$ of all participants
$\mathfrak{s}_0$	State kept in $\mathcal{M}_0$
ϕ	Transformations of the adversary
ψ	Transformations of the environment
ρ	Transformation defined in the security proof
δ	Storage domain called δ
$\mathcal{A}_\delta$	Adversary structure for storage domain δ
$\ominus_\delta$	Modification operator in storage domain δ
$\perp$	Failure symbol denoting invalid values
Gb	Garbling scheme
Ev	Evaluation algorithm for garbled circuits
$A \geq B$	A is as secure as B
$A \geq_{sec}^{model} B$	A is as secure as B in security model *model*, used especially to distinguish security from input-privacy notation
$A \geq_{priv}^{model} B$	A is as input-private as B in security model *model*
$Sys_1 \rightarrow Sys_2$	Ordered composition with data flow from Sys_1 to Sys_2
$\mathcal{F}_1 \vert \mathcal{F}_2$	Ideal composition of ideal functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$
$\mathfrak{M}$	Collection of simple machines
$[\mathfrak{M}]$	A completion of a collection that adds all connected buffers to the collection
$ports(\mathfrak{M})$	All ports that the machines in $\mathfrak{M}$ have
$free(\mathfrak{M})$	Free ports used to connect $\mathfrak{M}$ to machines outside it
$forb(\mathfrak{M})$	Ports inside $\mathfrak{M}$ that the environment Env is not allowed to have
$p?$	Input port named p
$p!$	Output port named p
$Conf(Sys)$	Set of all possible configurations of Sys
$(\mathfrak{M}_1, S_1) \vert\vert (\mathfrak{M}_2, S_2)$	Composition of structures $(\mathfrak{M}_1, S_1)$ and $(\mathfrak{M}_2, S_2)$
$Sys_1 \circ Sys_2$	Composition of systems Sys_1 and Sys_2
$\Pi_1 \langle \Pi_2 \rangle$	Composition of two protocols Π_1 and Π_2

1. INTRODUCTION

1.1. Proving Security

Security and privacy are properties that many consider to be of utmost importance to various aspects of their lives. For example, internet banking, voting, our homes, our income, or our cars all need some level of security. We simply need these things to be secure to reliably use them in our day-to-day lives. There is so much variety in what one can consider secure so it is pretty evident that no one clearly defined notion of security covers all these needs. Wherever the term is used, it is in some specific context. This context limits it to some degree, but even then, its actual meaning and details may differ for different people.

In the world of computer science, information security is one of the aspects of security that is often desired. Broadly, information security ensures that sensitive information is protected from unauthorized accesses, modification or deletion. For example, one's medical records should only be available for that person and their doctors. However, the correct level of access some person should have or the strength of the security mechanisms can vary a lot. For example, maybe no one should be authorized to delete the medical records or maybe they should not be stored at all after the data subject has passed away. Often, technology offers means to achieve some level of security but it is up to the application to define which flavour of security is needed.

Secure multiparty computation (MPC) is a tool that enables multiple parties to jointly compute on their inputs while only disclosing the output of the computation. This thesis is focused on the security of MPC, which already narrows the meaning of security to something quite specific but still leaves a lot of freedom to refine the notion. Many years of research into cryptography and secure multiparty computation have developed a common set of parameters and an understanding of different security properties and their relations. There are common frameworks for security proofs, vocabulary to define commonly used settings for these proofs, and known means to achieve some desirable security properties. However, in secure multiparty computation, one cannot separate the functionality from security. After all, it is often the most secure option not to compute anything and not to share any private information. So the common security definition for a secure multiparty computation protocol is an ideal functionality that specifies what the computation should do and which kinds of interactions are allowed. Anything that cannot be done with this functionality should also not be possible with a secure computation protocol. Such functionalities follow a common pattern where all parties give their inputs, the functionality computes the defined output and gives it to parties expecting the result.

While this approach gives an excellent common ground to define security, it does not take away the personal look at security. The protocol designer is at liberty to define the ideal functionality precisely as they see fit. The ideal functionality

for a complicated protocol such as voting may differ for different protocol designers. Hence, rather than considering the ideal functionality as something perfect, it should be viewed as a specification of both the strengths and flaws of the protocol. Then all that remains is to show that the real protocol is as secure as the one defined as ideal and that the ideal functionality is suitable for where the protocol should be used. A protocol is said to be secure if anything that can be learned from it could be learned from the ideal specification.

There are standard tools to prove that a protocol is as secure as an ideal specification. Commonly, these proofs are done in some framework that enables the prover to consider all possible contexts where the protocol might execute. The formal frameworks like universal composability or reactive simulatability that define these approaches are built so that the rest of the world is abstracted away from the concrete protocol. However, the available proof techniques ensure that the protocol remains secure in all contexts where the protocol can be used.

The overall idea of proofs that enable security under composition may be simple. Still, the existing frameworks for doing these proofs properly are complicated and filled with details that are often overlooked. Designing a protocol is fun, and the author already has an intuition about why the protocol is secure during the design. This intuition is usually enough to convince others, but it only constitutes a small factor of the overall formal proof. Hence, there is a significant decision either to go with an informal proof and summarise the intuition or to do the whole formal proof. The former is dangerous because it is still easy to overlook some details that may break the protocol. The latter however may not give any additional insights into the protocol, is often complicated and time-consuming to both write and verify, so the proofs are often overlooked in the publications.

Secure multiparty computation has reached a state where the computation frameworks are good enough to solve various real-life problems and more and more complex algorithms are built on top of the frameworks. In addition, there are many concrete protocols for secure multiparty computation and more will definitely emerge in the future. It would be beneficial if the algorithms could be defined in a general way, using some intermediate representation rather than the real frameworks as this would allow for more general adoption of these algorithms and easier upgrade of the underlying secure computation frameworks.

The goal of this work is to meet the protocol designer halfway. Firstly, this thesis specifies a natural structure for ideal functionalities that is suitable for many secure multiparty computation protocols. These functionalities can then be used to define the properties of secure multiparty computation frameworks and new algorithms. Secondly, a property called input privacy is introduced and formalised. A protocol is input-private if its execution does not leak any information about its inputs. Unlike a traditional secure protocol, the input-private protocol cannot give an output that provides information about the protocol inputs. Input privacy is sometimes achieved by simpler protocols than fully secure protocols with similar functionality. In addition, input privacy can be easier to prove. However, as

proven in this thesis, it is also easy to transform input-private protocols to secure ones in many cases.

Thirdly, an abstract protocol execution environment is derived from the formal specification of secure computation frameworks. This abstract execution environment is close to the world where the proofs based on intuition are described and the focus is on the public values seen throughout the protocol execution. The specification of the abstract execution environment also defines a series of properties that must be satisfied by the protocol or the computation framework. These properties ensure that a mostly intuition-based proof in the abstract world constitutes a formal proof with respect to the detailed specification of the secure computation framework. Hence, in many cases, intuition is sufficient, and there is no need to meddle with all the details of the proof of composable security. The abstract model also allows to decouple the real framework descriptions and security proofs from the proofs of the security of a new algorithm. Ideally, it can be used to build libraries of algorithms that use the security definitions for the ideal functionalities and data storage to define their requirements. Then any framework that meets these requirements can be used to implement the algorithms.

As a result of this thesis, many security proofs can be simplified and shortened. Therefore, privacy-preserving algorithm development can be faster and more enjoyable for the algorithm designer. In addition, the results of this thesis enable more efficient protocols for cases where input privacy is sufficient or where it is possible to use input-private components inside a composite protocol. The modular formalisation of secure computation and the abstract execution description also make it easier to give more general proofs of security for new algorithms. Rather than proving security for a specific secure computation framework, it is possible to specify the necessary properties of the underlying framework and prove that the new algorithm is suitable for all real frameworks that satisfy the specification. Hence, the main goal of the thesis is to enable the efficient development of new algorithms for secure multiparty computation while maintaining rigorous security. The results of this thesis bridge the gap between security intuition and proper formal proof. A more technical summary of the results is in Section 1.3.

1.2. Claims Proven in the Thesis

Claim 1: It is possible to formalise a notion of input-private cryptographic protocols that is composable and can be used as a building block to design secure multiparty computation protocols that have composable security.

Claim 2: Using input-private protocol components instead of secure can often give rise to more efficient protocols for secure computation. Furthermore, the security of the protocols can be derived from the composability results about input-private and secure protocols.

Claim 3: There exists an abstract execution environment that is equivalent to the more commonly used hybrid execution environment under reasonable

assumptions about secure multiparty computation protocols. The security proofs in the abstract execution environment can be translated to full formal proofs in the hybrid execution environment.

1.3. Summary of the Thesis

This work follows a long line of different formalisations of secure multiparty computation. The main goal is to build a theoretical framework that allows to do more modular and simpler security proofs while maintaining rigour. Especially, the focus is on the security proofs of algorithms designed for some secure computation framework with known security properties. Firstly, this thesis discusses input privacy and how input-private protocols can be extended to passively secure protocols. Secondly, this thesis considers how to extend a small set of protocols (*trusted core*) into a full-fledged secure multiparty computation framework with concise proofs. For example, to analyse the usual case where complex algorithms are built using a small core of basic arithmetic operations. This part defines an abstract execution model as a simplification of the secure computation framework formalisation.

The results of this thesis use the reactive simulatability framework (RSIM) as a basis for the formalisation. In the overall intuition, this framework and the commonly used universal composability (UC) framework are very similar. For universal composability, the framework is more flexible and there is also a restricted version intended for secure multiparty computation proofs. However, for the results here, RSIM provides more flexibility in discussing the components of a protocol execution than the secure computation version of UC while it avoids some more complex aspects of the full UC framework. More justifications for this choice can be found in Chapter 2. The definitions consider static and adaptive corruption models with both active and passive adversaries. The proactive model with a mobile adversary is not considered but some indications where the definitions need to be enhanced are noted in the text.

Secure multiparty computation formalisation. This thesis defines a modular formalisation of secure multiparty computation in Chapter 3. In modular execution, each operation, such as addition or multiplication, is a separate ideal functionality. These functionalities expect inputs in some secure data representation, for example, secret shares. Any such data protection scheme has to define functionalities to generate this representation from the input value and to reconstruct the value from the representation. The main privacy property of the storage is called *hiding*, meaning that it hides the value from any unqualified set of parties. The integrity property is called *modification awareness* to capture all cases from robust secret sharing to schemes without integrity protection. For example, additive secret sharing where all shares are summed to learn the secret value hides the secret from any group of parties less than the whole set. However, each party could modify the outcome by simply modifying its own share. However, additive

sharing can be enhanced with integrity checks to disallow modifications. Modelling these different properties and working with shared values is non-trivial, as the respective protocol functionalities must capture all modifications to the data.

Two different kinds of functionalities define interactive and local computations. Interactive *ideal functionalities* follow a pattern where they reconstruct all inputs, compute the desired functionality in plain and then apply the necessary protection mechanisms to all outputs. For example, the ideal multiplication functionality for shared values collects two secret shared inputs, reconstructs the values, multiplies the values, secret shares the result, and, finally, gives the shares back to the parties. Various interactions with the adversary can be added to this blueprint to accommodate different corruption models. *Local functionalities* represent the computations where a party or a set of parties do something with their representation of the data that has a meaningful effect on the value. For example, if all parties sum their shares of values that are additively shared then this is a local protocol for the addition of the values. It is local because there is no communication with other parties. Differently from the interactive functionalities, the output of the party depends on its inputs. If one would define an interactive functionality for addition, then all outputs would be random shares of the addition result. The secure computation framework is modelled as a collection of functionalities that work with a common setup and the same secure data representation.

Different data representations like encryption or secret sharing define *storage domains*. The intuition is that for each domain there is a protection functionality such as sharing or encryption that takes a public value into the domain. Inside the domain, each party has some representation of the data and there is an output functionality like reconstruction or decryption to return the protected value to the public domain. The ideal functionalities define which storage domains can be used to give inputs and where the outputs are. The computation functionalities and the respective storage domains together form *protection domains*. Data can be inserted into the storage domains, computed with using the functionalities and the results can be returned. Hence, a protection domain is a formalisation of a secure computation framework. The concrete protection domain also contains setup that in a way initializes the protection domain for a given execution. A representation of a value that is hidden only has a meaning within the instance of the protection domain where it has been created and can be operated with using the respective functionalities and help of other parties participating in the computations in that domain. For example, the setup with fixed parties and additive secret sharing with a set of protocols is a protection domain. A share created in its execution can meaningfully be used only in the further executions of this domain.

Input privacy. A computation protocol takes some input values from its participants, does some computations producing intermediate values and fixes some of the computed values as outputs. In secure multiparty computation, each party may have private inputs and it may receive private outputs that no other participant in the protocol knows. For example, a simple protocol executes some function $f()$

where the inputs x_i of the parties encode the input $x = \sum x_i$ as additive shares. Conceptually, the output of such protocol is $y = f(x)$ but each party i only learns y_i of the result $y = \sum y_i$ so that y_i alone does not reveal y. In the rest of this section, x_i denotes the inputs of the protocol and y_i denotes the outputs.

As described above, for an interactive ideal functionality there is dependency between x and y, but not between x_i and y_i. However, for a local protocol, there is a dependency between x_i and y_i. On the other hand, local functionalities, by definition, do not give their participants any new information about the values that other participants have. However, many interactive functionalities also ensure this property that we call input privacy. In different secure computation frameworks, it is noted that the computation until the publishing of the output does not reveal any information and the publishing is a special step for doing all verifications to ensure correctness and then reveal the output. Essentially, privacy until an explicit output is given is what one expects from a secure computation protocol. This thesis defines a version of input privacy that enhances this intuition with the idea that the intermediate secure representations that always remain private may, in a sense, be less secure than the ones used in the publishing step. For example, one may consider a framework that during the computation phase just stores all operations that need to be computed. The intermediate representation in this case is the code that is executed together with the private inputs. A publishing step may be the one that is responsible for first evaluating the result and then giving the numeric output back as expected. However, revealing the intermediate representation would leak all the private information stored as inputs to the operations.

Input privacy detailed in Chapter 4 is a unique security notion that applies to protocols where outputs and intermediate values, as seen by some protocol participants or the adversary, do not reveal any information about the inputs. For example, if all protocol outputs are secret shared, then the output that each party sees is just their share which, for the additive sharing example, is just a random value. Similarly, any tolerated adversary can see only a set of shares that does not define the value of the secret. The main idea is to consider protocols where each party knows its input initially, and after the protocol execution, it has not gained any insight into the private inputs or outputs of other parties. Hence, such protocols preserve the privacy of the private inputs. This thesis approaches this question in the specific setting of hiding output domains, meaning the cases where each party gets some representation of the output but does not learn the output.

A common security definition considers a case where an environment gives some inputs to the protocol, learns the output of the execution and tries to distinguish real and ideal executions. At the same time, the environment communicates with an adversary that can affect the protocol execution. The definition of input privacy is similar but assumes that the environment never learns the outputs from the protocol. This formalises the intuition that the steps before the output do not reveal information that is not revealed by the ideal private protocol. As a consequence, if the outputs of all parties of an input-private protocol are revealed,

then this full output of the input-private protocol may leak information about the inputs. For example, a protocol where each party adds 1 to their additive share is input-private as there is no communication. Still, if a party later reveals this output to another party, it also reveals its exact input. Note that in such protocol it is expected that pooling the individual outputs y_i reveal the value y but in this case, they also reveal the exact values of x_i. On the other hand, a secure protocol adding the number of participants to a secret value would only reveal the output y and the secret input x but not the values x_i held by each of the participants in the beginning of the protocol.

According to the definition of interactive ideal functionalities, such correspondence between inputs x_i and outputs y_i should not occur in a secure protocol. However, input-private protocols can be finished with a secure protocol to obtain a passively secure composed protocol. The thesis defines a notion of *ordered composition* used to define what it means to end one protocol with another. Essentially, it is the case where the second protocol can take the outputs of the first as its inputs and the execution of the first is not affected by the execution of the second protocol. For example, if addition results are later multiplied with some secret value using the multiplication functionality. The input privacy property is also composable itself, meaning that input-private components can be used as sub-protocols in larger algorithms so that the final protocol for the algorithm is still input-private for passive adversaries. Hence, many input-private components can be combined into more extensive input-private protocols and then transformed into passively secure protocols.

The input privacy definition ensures that the view of the parties that cannot reconstruct the value y does not leak information but does not ensure that the actual encoded output given to the parties is y. As such, this definition does not capture correctness in the sense that the protocol may give a different output than the ideal protocol that was used to prove input privacy. Hence, to define the actual functionality of the protocol or any composed protocols, the correctness has to be established in addition to input privacy. In Chapter 4 on input privacy, the correctness requirement is generalised to the *predictable output* requirement. The idea is that it is possible to predict the output of the composed protocol simply from the inputs of the input private component. This showcases that the second protocol does not reveal the values it receives from the first component and the predictor defines the expected functionality.

The benefit of using input-private components instead of secure ones is that they may be more efficient than secure protocols, and input privacy can be simpler to prove. Hence, both the development and the final protocols could be more efficient. In particular, if some input-private component is more efficient than the respective ideal functionality then one can use the input-private components until the actual output of the protocol is reached and then turn the representation in the storage domain to the secure representation before publishing. It is future work to generalise the input privacy results to the case of active adversaries.

Algorithm security and the abstract model. The security of complex cryptographic protocols is often discussed in the hybrid execution model, where the real protocol components have been substituted with their respective ideal functionalities. For a protocol for a dot product, the natural components are functionalities to multiply and then add the multiplication results. In a hybrid model, the inputs are secret shared, then given to the multiplication functionalities and the shared results are given to the addition functionalities. Possibly, the protocol might start with secret sharing some plain values and finish with reconstructing the result. When considering possible adversarial interactions with such protocol, the adversary can usually control the timing of all message exchanges and modify the messages sent by the corrupted parties and interact with the functionalities. The abstract execution model proposed in this work simplifies the hybrid execution model for cases where new algorithms are implemented using existing secure multiparty computation protocols. The ideal functionalities considered in this thesis may take both values and secure representations of data as inputs and give the same as outputs. Hence, a hybrid execution also has to specify the data representation and it can be modified between the ideal functionality calls. This differs from a more common model of secure computation known as the arithmetic black-box where the intermediate values are not modifiable but adds flexibility to consider lightweight ideal functionalities. The abstract model removes the need to consider concrete representations of secure data and limits the possible actions of the adversary by specifying properties that the data representation and ideal functionalities have to fulfil. For example, the abstract model version of the dot product specifies that the data representation is hiding and does not allow the adversary to modify it without detection. Otherwise, the structure of the protocol remains the same as in the hybrid model. Hence, a protocol considered in the abstract model can be realized by various concrete schemes that satisfy the properties of the storage domain and implement the desired ideal functionalities for these storage domains.

In the *abstract model*, the secret values are stored in a memory that is split into different storage domains with varying measures of protection and adversarial access. A value is untouchable if the underlying secret sharing is hiding and robust. At the same time, the adversary can destroy values protected by some verification scheme and introduce controllable changes in secret values without integrity protection. The ideal functionalities that usually are defined for a concrete data representation can in the abstract model be defined based on the properties of the storage domains. For example, an ideal functionality can be defined for a verifiable storage domain for integers modulo p and then have to be able to work with destroyed values where the verification does not succeed. In the abstract model, the adversary sees all published values – both the intermediate values made public in the algorithm to optimise the program flow as well as any final public outputs. Furthermore, the adversary sees all values available to the corrupted parties. For protocols with dynamic scheduling, the adversary can stall some operations and control the timing of parallel functionalities. The final security analysis of the pro-

tocol in the abstract model focuses on the limited modifications the adversary can make, the public values it sees throughout the execution, and the timing controls. For example, in the dot product protocol, the adversary can affect if the protocol proceeds or stalls at some operation and the adversary may abort the protocol execution. However, if no values are published then it does not have access to any values. All actions that the real adversary can do by modifying the messages of the corrupted parties are carried out as operations on the values in the storage domain. Specifically, they can either modify, destroy or not change these values at all depending on the properties of the storage.

The thesis proves that the abstract model is equivalent to the hybrid execution model under some natural restrictions for secure multiparty computation protocols. These restrictions appear when considering the transformation from the hybrid to the abstract model as done throughout Chapter 5. The discovered restrictions make various natural assumptions about secure multiparty computation protocols explicit. For example, the assumption that all participants are executing the same algorithm, branching is done based on public variables, or the functionalities are defined for all sorts of corrupted inputs. As such, the equivalence result is interesting from a theoretical viewpoint. It also paves the way for simple and intuitive formal proofs of security of the protocols executing complex algorithms composed of small building blocks. A security proof in the abstract model can be translated into a full proof in the hybrid execution model. In addition, the equivalence result opens a way to characterise the properties of single operations and storage domains needed to ensure the security of new algorithms. However, there is future work to make the results applicable to more general classes of functionalities and study the implications of some of the restrictions.

1.4. Thesis Outline and Contributions of the Author

Chapter 2 introduces the main properties and methods of secure multiparty computation and serves as the background for the main results. It also introduces the underlying formalisation called reactive simulatability and the visual notation proposed for and used throughout the thesis. In addition, the chosen formalism of reactive simulatability is compared with the more commonly used universal composability to stress why this choice is a natural one for the requirements of the following chapters. This chapter is partially based on the comparison of secure computation methods from [5].

Chapter 3 introduces the formalisation of programmable secure multiparty computation frameworks. This chapter is based on the formalisation used in the papers [30, 96] that are part of this thesis and serves as a common basis for Chapters 4 and 5 that are the main contributions of this thesis. Chapter 3 introduces secure storage domains, the specification of functionalities used in secure multiparty computation protocols and how small functionalities are combined to form secure computation frameworks. This chapter also compares the given formalisa-

tion and other formalisations of secure multiparty computation.

Chapter 4 defines input privacy, and demonstrates the composability of this notion. The main body of this chapter is based on [30] and shows Claim 1. Section 4.8.2 is based on [125] and demonstrates an application of the composition of input-private and secure protocol. Overall, Section 4.8 shows Claim 2.

Chapter 5 derives the abstract execution environment from the commonly used hybrid execution model. The derivation also proves the equivalence of the abstract and hybrid models under specific assumptions about the executed protocol and the secure computation framework. This chapter is based on [96] and shows Claim 3.

Chapter 6 concludes the thesis and provides ideas for future work.

The contributions of the author of the thesis in each of the publications are as follows.

[30] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From Input Private to Universally Composable Secure Multi-party Computation Primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 184–198. IEEE Computer Society, 2014

The author wrote the initial version of the proof of security of the composition of private and secure functionalities. The original proof was significantly refactored in collaboration with the co-authors to arrive at the modular proof found in the paper and Chapter 4.

[125] Pille Pullonen and Sander Siim. Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2015

The author designed the proposed hybrid protocol together with the co-author Sander Siim. Sander was then responsible for implementing the proposed protocol, and the author of the thesis was responsible for the security proof. The generalised version of this algorithm and a proof of security for the new algorithm that follows the same ideas as the original proof appear in Section 4.8.2 of the thesis.

[5] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and Performance of Programmable Secure Computation. *IEEE Security & Privacy*, 14(5):48–56, 2016

The author of the thesis was responsible for coordinating the paper writing and finalising the paper. The main findings of the paper, such as the efficiency evaluations, were carried out by the co-authors but were systematised for the paper by the author in close collaboration with Dan Bogdanov. The author of the thesis participated in the DARPA PROCEED program and

European Union PRACTICE program, which served as the source material for the paper. This paper is reflected in Chapter 2 of this thesis, and the overall understanding of programmable secure computation has strongly affected the formalisation in Chapter 3.

[96] Sven Laur and Pille Pullonen-Raudvere. Foundations of Programmable Secure Computation. *Cryptography*, 5(3):22, 2021

The author of this thesis worked in close collaboration with her co-author and thesis supervisor, Sven Laur, to derive the results and definitions of this formalisation of secure computation and the transformation to the abstract execution model. The author of the thesis was responsible for putting together the final version of the publication. The main formalisation presented in this paper appears in Chapter 3. Chapter 5 extends the equivalence result between the hybrid and abstract execution in more detail than the original publication.

2. PRELIMINARIES

This chapter introduces the main concepts required by the rest of this thesis. However, it is possible to skip this chapter or sections of it initially and read them when they are referenced by the other chapters. Firstly, Section 2.1 introduces secure multiparty computation. It considers both the desired properties as well as different technologies for achieving secure multiparty computation. This introduction serves as a basis for the formal approach to describing secure computation protocols in Chapter 3. This section helps to give context to the contributions of the thesis.

Secondly, in Section 2.2 this chapter introduces the overall idea of simulation-based proofs and universal composability. Section 2.2 introduces and compares the commonly used universal composability framework [42] and the reactive simulatability framework [9,122]. The latter is the formalism used throughout the rest of the thesis. Section 2.2.2 introduces the new visual notation used to illustrate and define the protocol details in the rest of the thesis. The reactive simulatability framework in Section 2.2.5 and its visualisation in Section 2.2.2 are the basis of the formalisations of this thesis as introduced in Chapter 3, other frameworks are included for comparison.

2.1. Secure Multiparty Computation

This section introduces secure multiparty computation schemes and properties. It gives context to the formalisation in Chapter 3 and various details of schemes and properties will be referenced by concrete examples in the following parts of the thesis. Note that many properties are mainly described for general background and in order to illustrate why the formalisation in Chapter 3 needs to allow for various details.

In secure multiparty computation, a set of n mutually distrusting parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ wish to compute some joint outcome on their private inputs. The overall security intuition is that nothing other than the output of the computations should be revealed to the parties and the output of the computations should be correct. These make up the privacy and correctness requirements. In addition to privacy and correctness, there are different properties one might expect from a protocol for secure multiparty computation. The most common security properties are discussed in Section 2.1.1. There are also many different methods that secure computation can be based on. The major technologies are linear secret sharing (Section 2.1.4), garbled circuits (Section 2.1.5), and homomorphic encryption (Section 2.1.6). The different methods can be combined as discussed in Section 2.1.7. There are also additional technologies and goals for secure computation, such as oblivious RAM [74] or function secret sharing [36] that are not discussed in this thesis. In addition, there are different schemes that implement specific functionalities, such as private set intersection or private information retrieval. However, the focus of

this thesis is on computation techniques that can be programmed to compute any algorithm, and hence, the special purpose protocols are out of scope.

Programmable secure computation means that the computation method defines ways to compute a wide range of functions and algorithms. Commonly these algorithms are specified as boolean or arithmetic circuits. The algorithms can give either public outputs known to all parties or private outputs to some parties. In addition, the computation can either focus on a function with one round of outputs or be reactive. For example, evaluation of arithmetic circuits means that there can be arbitrary computation, but the output wires give the computation output at the end of the execution. Reactive computations are such that there can be many rounds of computation that share the state, and any input can affect the outputs of all the following rounds. In addition, the parties can use intermediate results to decide which computations to do next. The latter setting is the most interesting when considering developing new algorithms. Overall, reactive functionalities still evaluate circuits, but the outputs of the circuit are still protected and can be opened with a special publishing round, whereas the outputs can be used as inputs to further computations. However, the most common way to define such reactive computation is the specification of an arithmetic black-box discussed in Section 2.1.8. In practice, the computations can either be defined as circuits or also as programs. For any program, the main difficulty is handling conditional statements, which can either be computed by evaluating all cases and obliviously choosing the right outcome or by only allowing branching on public values. Either the computation evaluating all branches or the computation outside of branching can then be described as circuits.

The participants $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of secure multiparty computation protocols can have different roles. There are computing parties that participate in the main computations of the secure multiparty computation protocols. Input parties are the ones giving their private data as input to the computations, and result parties are the ones receiving the computation output. In the simplest case, each party has all three roles. Most of the theory about secure computation either focuses on this case or simply only considers the computing parties. However, in many practical use cases like the ones discussed in [4], these roles will be carried out by different participants. For example, input parties are the ones that hold the original data, computing parties are participants with a suitable infrastructure to set up and execute the protocols, and result parties are data analysts.

2.1.1. Adversary Models and Flavours of Security

This section summarizes the different aspects of security of secure multiparty computation protocols and serves as background for the rest of the thesis. In secure computation, security is defined with respect to an adversary. The adversary is a single party that tries to interfere with the protocol and its participants. Most notably, the adversary usually controls some parts of the network communi-

cation and can corrupt parties. The corrupted parties are under the control of the adversary and the adversary can see all their view of the protocol and may be able to change their actions. Hence, this adversary can cover both external observers as well as collusion attacks where several participants of the protocol are working together to undo the security properties of the protocol. It can also be considered as an abstraction of the case of many independent bad participants and the model with a single entity controlling them is stronger. More formal security definitions are given in Section 2.2.

In terms of adversary actions, one can consider either passive (also known as semi-honest or honest-but-curious) or active (also known as malicious) adversaries. The former follows the protocol description but tries to derive new knowledge from its view in the protocol. The latter, on the other hand, actively tries to break the protocol and learn the secrets of the other parties. In order to stop an active adversary, the protocol has to have some mechanisms to find all actions that may cause leaks or faults in the protocol. In the information-theoretic setting, there is no limit to the actions of an active adversary, and in the computational setting, the adversary is required to work in polynomial time. Other than this timing limit, the active adversary is allowed to do arbitrary actions. Sometimes also a covert adversary [6, 7] is considered that also might arbitrarily deviate from the protocol but is worried about getting caught cheating. Hence, in order to stop a covert adversary, it is sufficient to have a reasonably big probability of noticing any cheating behaviour.

The adversary can corrupt parties of the computation and play their roles. The allowed corruption models can either be static or adaptive. For static corruption, the adversary chooses the parties to corrupt before the execution of the protocol. For adaptive corruption, it can choose parties to corrupt during the execution. Furthermore, it is possible to also consider mobile corruption [114], where the adversary may corrupt a party and later drop its control over the corrupted party. Mobile adversary is used to model cases where adversary is discovered and the system is cleaned. However, the mobile corruption model is not commonly considered for secure multiparty computation. Dealing with mobile corruption would add extra complexity to the protocols. It would also complicate analysis because it is not easy to distinguish between an external attacker that can be kicked out from the system and a party that is corrupted itself and should then be fully dropped from the computations. A common approach to dealing with such an adversary is to periodically refresh the representation of the private data so that the data the adversary has seen beforehand becomes unusable. This thesis focuses on static or adaptive adversaries.

There are many different commonly used properties that a secure protocol may or may not have. For the protocol outputs, it is necessary to consider who and under what conditions learns the outputs, as thoroughly discussed in [51]. A protocol is said to be fair if all result parties either get the output or none of them does. However, if a protocol has guaranteed output delivery, then all output parties al-

ways get the needed output. On the contrary, a protocol has security with abort if the adversary can decide if the protocol gives an output at all or may simply finish with abort. In addition, for security with abort, the adversary sometimes gets to see the output before it can decide to abort the protocol, and, at other times, the decision has to be made blindly. Regarding the protocol inputs, the protocols often ensure the independence of inputs, meaning that it should not be possible for an input party in a real protocol to give an input that depends on the input of some other input party. For example, it should not be possible for a party to always set its input to be equal to the input of another party using the information that it sees about the other input during the protocol execution.

A secure protocol does not have to be secure against all possible attacks and adversaries. Rather, the protocol specifies the properties that it satisfies and the type of adversaries it tolerates. In addition, the protocols limit how many parties the adversary is allowed to corrupt before the security properties are broken. In one case, the protocols either expect an honest majority meaning that more than half of the parties are honest or tolerate a dishonest majority, where the expectation is that security is achieved as long as one party remains honest. In the more general case, the protocol can specify exactly which subsets of parties can be corrupted together. These sets form an adversary structure $\mathcal{A}$. All reasonable adversary structures are such that if a set P of parties is in the structure, then any set R such that $R \subseteq P$ must also be in the adversary structure. Meaning that if the adversary can corrupt the set P, then it can also corrupt any subset of this set. The most common adversary structures are threshold structures where the adversary can corrupt any set of t or fewer computing parties.

Security definitions of the protocols also include different assumptions regarding the setup of the world where the protocol is executed. For example, for networking, it is common to assume secure point-to-point network channels or the existence of secure broadcast or multicast. In practice, this means that these properties have to be achieved using some specific protocols. It is also common to expect other functionalities to exist, for example, public key infrastructure or some common source of randomness. In the computational security case, the underlying hardness assumption also constitutes an important security detail. For example, if some mathematical problem is broken, then the protocol is not secure any more. On the other hand, protocols with statistical or perfect information theoretic security do not depend on any such assumptions, but there is some fixed parameter defining the success of the adversary.

2.1.2. Feasibility of MPC

There are various results concerning which settings or setup models are suitable for which MPC protocol. This section summarizes the main results known about possible MPC protocols and serves as a general background for the thesis. Overall, it is possible to achieve secure computation with different security models and

adversary structures. Hence, the formalisation appearing in Chapter 3 has to be generic enough to be able to consider all these different settings.

In the information-theoretic model with secure point-to-point channels, perfect security can be achieved for any functionality with guaranteed output delivery if and only if the adaptive adversary corrupts less than half of the parties for the passive security (honest majority setting) and less than a third for active security [24, 25, 50]. However, in the statistical security and assuming a broadcast channel, active security can be achieved for honest majority [127]. It is impossible to achieve information-theoretic security in the dishonest majority setting [24]. In the computational security model with trapdoor one-way permutations, active security can be achieved for the honest majority for both static [75, 76] and adaptive [45] corruption model. Moreover, computationally secure computation without guaranteed output delivery or fairness is also possible with a dishonest majority assuming the broadcast channel and existence of trapdoor permutations, e.g. using the GMW construction [75]. In particular, two-party computation is only possible in the computational setting.

The threshold results can be generalised to an adversary corrupting sets in an adversary structure as done in [82, 83]. They consider conditions Q2 and Q3 where, respectively, no two or three sets in the adversary structure cover the set of all parties. In this case, any functionality can be securely computed with information-theoretic security if and only if the adversary is in Q2 for the passive security case and Q3 for the active security case. In the computational setting, Q2 still allows secure computation with guaranteed output delivery [53].

2.1.3. Private Protocols

In some cases, the security of secure multiparty computation protocols can be divided into two parts. The first is the correctness of the protocol independently of the adversarial actions, and the second is the privacy of the protocol. A notion of privacy called input privacy is a focus of this thesis in Chapter 4, where it is shown how to transform private protocols to those secure against passive adversaries. However, there are other treatments and definitions of privacy and its connection to security in the literature. This section gives an overview of definitions related to input privacy.

Intuitively, the protocol is private if it does not leak too much information. Or, in a more common case, everything that can be deduced from a corrupted view can be deduced from the input and output of the corrupted party. Some secure multiparty computation protocols can be seen as containing two phases: first evaluation and then verification and publication of the result. In such cases, it is often interesting to consider privacy for the first part. Such privacy, in the case where there are no public outputs yet, is known as input privacy [30] (as in Chapter 4), active privacy [118] or active security with weak privacy [70]. These approaches give different concrete definitions and uses for this property but cap-

ture the same overall idea. The assumption is that the intermediate values do not leak anything about the inputs other than what is known purely from the inputs of the corrupted parties. This property, in turn, is closely related to security against additive attacks [70], which states that an active adversary can change the output of the computation by a known modifier but cannot learn other information. Using a protocol that is weakly private for an active adversary with a circuit that is resilient against additive attacks gives a secure protocol [70].

The privacy property can be defined for common classes of adversaries, such as passive or active adversaries. An interesting case between these is the definition of defensible privacy [78] where the adversary may modify the protocol messages but, in the end, can produce a valid input and initial randomness (also known as the defence) that would also produce the same messages honestly. Such proof guarantees that the actions appear to be honest and that nothing other than the defined output is leaked because otherwise, it would also leak to that passive adversary. Defensible privacy can be easily extended to security against covert adversaries if the knowledge of defence can be proven without revealing the inputs.

Privacy, especially against active adversaries, is an interesting building block when designing protocols that are secure against active corruption. Essentially, it is possible to combine an actively private protocol with another protocol that checks the correctness of the private part. This idea has been used by [64, 80] to define ways to compile passively secure protocols with active privacy or defensible privacy to actively secure protocols.

2.1.4. Secret Sharing

This section introduces secure computation that is based on secret sharing. It is mostly background for the secure computation formalisation but some of the following examples use concrete protocols based on additive secret sharing. Secret sharing was proposed independently by [27, 129] and, starting with [24, 50], it has been used in numerous ways to achieve secure multiparty computation. In secret sharing, a secret value is split into several shares, and each share is given to one participant. For a (t, n)-threshold scheme, the secret is split into n parts, and t of them are necessary to reconstruct the secret. Hence, the secret sharing scheme consists of two algorithms – sharing and reconstruction.

A secret sharing scheme is perfectly hiding if having less than t shares reveals nothing about the secret value. A secret sharing scheme is linear if the sharing and reconstruction operations are linear transformations. All linear secret sharing schemes allow participants to compute their shares for the linear combinations of shared elements without interaction. A linear secret sharing scheme is called multiplicative if each party can take its shares of a and b and compute c_i such that the multiplication result ab can be computed as a linear combination of c_i [53]. A multiplicative linear secret sharing scheme is a good basis for secure multiparty computation because the multiplicative property can be extended to a

proper multiplication protocol using other properties of the linear secret sharing scheme. The general multiplication protocol, sometimes also called the Maurer's multiplication protocol, is defined in [53] as follows. First, the parties compute c_i, where

$$\sum_i r_i c_i = c$$

is the multiplication result. Then each party $\mathcal{P}_i$ secret shares their c_i to shares c_{ij} and gives c_{ij} to party $\mathcal{P}_j$. Then the multiplication result is combined to properly secret-shared form by each party $\mathcal{P}_j$ separately computing their part of the linear combination as

$$c_j = \sum r_i c_{ij} \ .$$

Note that Maurer's protocol is a generalisation of the Gennaro-Rabin-Rabin multiplication protocol [71] for Shamir's secret sharing [129].

If a secret sharing scheme is not multiplicative, then it may still be used for secure computation, but dedicated protocols must be designed for multiplication and other necessary operations. A recent ongoing overview of secure computation using secret sharing can be found in [67].

Current schemes for secure multiparty computation often use additive secret sharing. In additive secret sharing, a secret x is shared to x_i such that

$$x = \sum x_i$$

and each party $\mathcal{P}_i$ has x_i. Additive secret sharing is mostly considered in finite fields or rings, but there are also variations of it over integers [58]. Additive secret sharing is not multiplicative. Hence, each scheme has to define a suitable multiplication protocol. It is most common to use precomputation-based multiplication protocol using Beaver triples [17].

A formal definition of secret sharing algorithms and properties for secret sharing are given in [22]. The focus of [22] is on computational robust secret sharing, but the framework also considers both computational and information-theoretic secret sharing with and without robustness. Intuitively, a secret sharing scheme is robust if honest parties can always reconstruct the secret. Overall, there are two properties to be considered – privacy and recoverability. The necessary procedures to define these properties are in Figure 1 and 2 respectively for secret sharing defined by two algorithms Share and Rec for sharing and reconstruction. In such game-based definitions, the adversary A interacts with the defined components and tries to achieve some goal that breaks the desired property of the used scheme.

The overview of the privacy game is in Algorithm 1. The game starts with an initialisation step that creates an empty set of corrupted parties T. Then the adversary can secret share some values and is given access to the CORRUPT functionality to corrupt some parties and see their view. In the end, the adversary makes its final guess about which secret was shared. This determines if A wins

Algorithm 1: Privacy game

$T \leftarrow \emptyset$
$(S^0, S^1) \leftarrow \mathsf{A}(1^k)$
$b \xleftarrow{\$} \{0,1\}$
$\mathbf{S} \leftarrow \mathsf{Share}(1^k, S^b)$
$b' \leftarrow \mathsf{A}^{\text{CORRUPT}}()$
return $b == b'$

Algorithm 2: CORRUPT(i)

$T \leftarrow T \cup \{i\}$
return $\boldsymbol{S}[i]$

Figure 1: Functions used in the secret sharing privacy game.

or loses the game. The adversary wins the game and breaks the security if the game returns True. The games use a security parameter k. This can be omitted for cases of information-theoretic security, and often for secret sharing, the security parameter is the same as the size of the set of possible secret values. Note that, in both privacy and recoverability, it is only interesting to consider adversaries that are limited to corrupting only a subset of parties. The variable $\mathbf{S}$ contains the state of the game, and the set T contains the indices of corrupted parties. The corruption call adds the party i to the set of corrupted parties as the union $T \cup \{i\}$ of the previously corrupted parties T and the set $\{i\}$. The success of the adversary is measured as twice the probability of winning minus one, specified as

$$2Pr[\text{Privacy game returns True } \& \ T \text{ is less than the bound}] - 1 \quad .$$

For a threshold scheme, the bound is $|T| < t$, but for schemes with more complex access structure, the bound is defined so that the set of the shares seen by the adversary is less than required to reconstruct the secret. The exact bound depends on the scheme that is considered. Secret sharing scheme has perfect privacy, if the success of the adversary is 0. For a computational secret sharing, the probability has to be negligible for any efficient adversary.

Algorithm 3: Recoverability game

$T \leftarrow \emptyset$
$S \leftarrow \mathsf{A}(1^k)$
$\mathbf{S} \leftarrow \mathsf{Share}(1^k, S)$
$\mathbf{S}' \leftarrow \mathsf{A}^{\text{CORRUPT}}()$
$j \leftarrow \perp$ or index of some trusted party
return $\mathsf{Rec}(1^k, \boldsymbol{S}'[T] \cup \boldsymbol{S}[\overline{T}], j) \neq S$

Algorithm 4: CORRUPT(i)

$T \leftarrow T \cup \{i\}$
return $\boldsymbol{S}[i]$

Figure 2: Functions used in the secret sharing recoverability game.

In the recoverability game in Figure 2, the adversary can choose the value S that is shared and can learn the shares of corrupted parties using the CORRUPT functionality. Finally, the adversary can choose new shares $\mathbf{S}'[T]$ for the corrupted parties T and the reconstruction operation Rec is executed using the honest shares

in $\mathbf{S}$ and the corrupted shares from $\mathbf{S}'$. The adversary wins if it can make the reconstruction algorithm output something else than the initially shared value S. The value $j \in \overline{T}$ points to an honest share and is used in cases where some share can be trusted to be correct. The value j can also be left unspecified. The use of j helps to characterise both reconstruction by an external party that cannot trust any shares and reconstruction by some party that has one share and knows that its share is correct. The corrupted shares $\mathbf{S}'$ can also contain missing values. The success of the adversary is measured as the probability that the recoverability game returns True, or as

$$Pr[\text{Recoverability returns True } \& \ T \text{ is less than the bound}] \ .$$

Concrete variants of secret sharing are defined by specifying a limit on this success. A scheme has perfect robustness, if the success is 0. On the other hand for statistical robustness, there can be some small probability of success.

Such definitions consider secret sharing in isolation, just focusing on the sharing and reconstruction algorithms. For the cases of secure computation using secret sharing, it is important to also consider shares that are obtained through computation protocols and not directly from the sharing algorithm. In addition, sharing schemes used for secure computation often have some setup and parameters that are shared between the shares of different values. Chapter 3 defines hiding and modification awareness properties as the extensions of the privacy and robustness properties for the case where there can be computation, some setup, or the shares may be generated incorrectly. In many cases, the hiding and modification awareness properties can be derived from privacy and recoverability.

The previous definitions from [22] do not cover verifiable secret sharing or cheating detection. There are two variations of verifiability depending on whether the share or the value is verifiable. A share in a secret sharing scheme is said to be verifiable if each party can verify the correctness of the share that it receives. The important difference in definitions of the secret sharing scheme is that, for verifiable secret sharing, the dealer who creates the shares is assumed to be the adversary who may not always create correct shares. On the one hand, such a view is possibly better suited for considering secure computation than assuming an honest dealer. On the other hand, current secure computation schemes often consider the case where individual shares are not verifiable, but the value encoded in the shares is. For example, the combination of additive shares and message authentication used by the SPDZ framework [59] allows one to verify the secret but not individual shares. For use cases in secure computation, where verification of the output is crucial, the previous definition of privacy still holds. For recoverability, it would be reasonable to define the winning condition as a successful reconstruction of a value other than the initially chosen value. However, making the reconstruction output a failure is not considered a successful attack. Both variations of verifiability can be considered as part of the modification awareness definition in Chapter 3.

2.1.5. Garbled Circuits

This section introduces garbled circuits as they are one of the computing methods that has to be covered by the formalisation in Chapter 3. Garbled circuits are also used in Section 4.8.2 for an example of using input private components in a composed protocol. The core of the garbled circuit idea [21, 101, 137] is the two-party scheme between a garbler and an evaluator. The garbler creates the garbled version of the circuit that they want to evaluate and sends it to the evaluator. It also sends the encodings of its inputs to the evaluator, and the evaluator uses oblivious transfer to learn the encodings of its inputs. Oblivious transfer is a special two-party protocol where one party has two messages, and the other receives one of them. The receiver does not learn any information about the other message and the sender does not learn which message was transferred. The evaluator then uses the encodings and the description of the garbled circuit to arrive at either the plain or encoded output of the circuit. Both participants are in the role of the computing party and – usually – both are also input and result parties. However, the basic idea can be extended and combined with other techniques to allow for more input and result parties. For example, it is easy to only give the output to the evaluator. It is also possible for the evaluator not to learn the real output if the circuit outputs a ciphertext that can only be decrypted by the real result party. The main privacy and security guarantees of the garbled circuits protocol require a secure garbling scheme and an input encoding that hides the inputs. The security of the oblivious transfer protocol ensures that only the desired output can be decoded.

Note that there are also various approaches to obtain secure computation against active adversaries with garbled circuits, for example [79, 100, 102, 110, 111]. In these cases, the following security properties are still relevant, but there are added properties that are needed to ensure that the garbling has been carried out correctly. This thesis only requires formal garbled circuit security definitions for passive security as they are needed in Chapter 4. However, the formalisation of the secure data representation for MPC in Chapter 3 generalises the ideas and would also applicable to garbling schemes when the focus is in the mapping between wire labels and actual values.

The main body of the garbled circuit approach is the garbling scheme Gb used to encode some function f. The garbling algorithm gives out the garbled circuit F, the rules for encoding the input e and for decoding the output d. A plain input x is translated to the garbled input X using the rules in e, then the garbled circuit F is evaluated on X to learn the garbled output Y that can be translated to plain output y using d. In a correct scheme, $y = f(x)$. The circuit f can be defined in various ways, but it has several important parameters independently of the concrete definition. Consider the specification from [21], where each gate has two inputs. The circuit has n inputs, m outputs, and q gates. For each gate, function A defines its first input wire and B defines the second input wire. Finally, function G determines the functionality of each gate. This specification and the

following security definitions will be especially relevant later in Section 4.8.2. However, they are important to generalise the properties of data protection used in MPC to the properties considered in Chapter 3.

The following summarises some of the security definitions of garbled circuits from [21]. The privacy property defines that running the garbled circuit protocol does not reveal more information than just seeing the output of the computation. The privacy property is a version of the main security property idea of MPC protocols. Obliviousness characterises the case when the output of the protocol hides the actual output value, and the protocol does not reveal any new information to the participants. The obliviousness property resembles the property that will be defined as input privacy of secure computation protocols in Chapter 4. In addition to these properties, the definitional framework of [21] also considers authenticity that is required for verifiable computation, but this property is not addressed in this work.

The idea of the privacy property is that any party obtaining the garbled circuit F, the garbled input X, and the output decoding information d should not learn anything other than what is revealed by just knowing the final output y. However, in many cases, there could be something more revealed by the functionality. This is considered to be the side information $\Phi(f)$ of the functionality f. For example, the side information can be used to capture flavours where the circuit f is either known or hidden.

Both privacy and obliviousness are defined in two ways in [21] – once based on indistinguishability and once based on simulatability. These notions are shown to be equivalent for some common side information functions Φ. Components of the simulatability-based definitions (`prv.sim` and `obv.sim` respectively for privacy and obliviousness) are summarised in Figure 3 as these are considered to be the stronger ones in case the two flavours do not coincide. In addition to the main definitions of the garbled circuit components, the games use a security parameter k, encoding function $\mathbb{E}$ and a simulator Sim. Each game starts with the initialisation phase that fixes b and finishes with checking if the adversary guessed correctly as shown in Algorithm 5. The actual garbling operation is different for the privacy and obliviousness games, and these are shown in Algorithm 6 and 7, respectively. The adversary is successful at breaking either privacy or obliviousness of the garbling scheme if it correctly identifies the challenge bit in the respective game. In both of these games, the success is measured as twice the probability of the game returning True minus one,

$$2Pr[\text{Garbling game returns True}] - 1 \quad .$$

A garbling scheme has `prv.sim` or `obv.sim` security for a side information function Φ, security parameter k and simulator Sim if the success in the respective game is negligible.

In the case of privacy, the adversary indeed obtains all the information about the garbled circuit - the circuit F, the input X and the decoding information d. The

Algorithm 5: Garbled circuits security game structure

$b \xleftarrow{\$} \{0,1\}$
$(f,x) \leftarrow \mathsf{A}(1^k)$
$b' \leftarrow \mathsf{A}(\textsc{Garble}(f,x))$
return $b == b'$

Algorithm 6: $\textsc{Garble}(f,x)$ for `prv.sim` game

if $x \notin \{0,1\}^{f.n}$ **then**
  return $\perp$
if $b = 1$ **then**
  $(F,e,d) \leftarrow \mathsf{Gb}(1^k,f)$
  $X \leftarrow \mathbb{E}(e,x)$
else
  $(F,X,d) \leftarrow \mathsf{Sim}(1^k,f(x),\Phi(f))$
return (F,X,d)

Algorithm 7: $\textsc{Garble}(f,x)$ for `obv.sim` game

if $x \notin \{0,1\}^{f.n}$ **then**
  return $\perp$
if $b = 1$ **then**
  $(F,e,d) \leftarrow \mathsf{Gb}(1^k,f)$
  $X \leftarrow \mathbb{E}(e,x)$
else
  $(F,X) \leftarrow \mathsf{Sim}(1^k,\Phi(f))$
return (F,X)

Figure 3: Components of the games for defining `prv.sim` and `obv.sim` security of a garbling scheme with garbling algorithm Gb, encoding algorithm $\mathbb{E}$, simulator Sim, security parameter k, length of input $f.n$ and side information function Φ.

only difference in the obliviousness case is that the decoding information is withheld from the adversary. Note that the simulator of the obliviousness game does not necessarily know the circuit, and hence, the encoded circuits and the outputs of the functions may differ. Thus, if the decoding information was made available to the adversary, then distinguishing the simulated and real circuits in the obliviousness game could be very simple. On the other hand, oblivious circuits also give no guarantees about what else the adversary may learn when it can decode the output. Notably, many use cases of garbled circuits are such that the output of the circuit is indeed public and the function d is trivial. Such schemes are not oblivious, but they can be private, as nothing is broken in the privacy game by also adding d, even if it is trivial. The obliviousness property is closely related to the input privacy property of secure computation that is developed in Chapter 4 and Section 4.8.2 explores the connection between the two properties.

Many garbling schemes hide at least some parts of the function that is evaluated even though it is not always required or necessary. Likewise, the topology of the functionality is usually revealed to allow for efficiency. Common optimisations also reveal other information. For example, the technique known as free-XOR [91], which makes it easier to garble XOR gates, reveals which gates of the circuit are indeed XOR gates. The analysis in [19] shows that side information functions Φ leaking either the topology or the location of XOR gates

are efficiently invertible and therefore [21] establishes that, for such schemes, the simulatability- and indistinguishability-based definitions coincide.

2.1.6. Homomorphic Encryption

This section introduces the main idea of using encryption as a basis for secure computation as this is one method that is covered by the formalisation in Chapter 3. Homomorphic encryption is an encryption scheme that allows computations on encrypted values. For example, there exists an operation that can be done with the ciphertexts such that the result is a ciphertext containing the encryption of the sum of the plaintexts inside the initial ciphertexts. Some homomorphic encryption schemes that are sometimes also called partially homomorphic or semi-homomorphic, support one such operation. For example, additively homomorphic encryption like Paillier encryption [115] allows only to compute addition with the ciphertexts. Fully homomorphic encryption [72,73] allows both addition and multiplication and can therefore be used to compute any arithmetic circuit. Somewhat homomorphic schemes like [35] allow doing both addition and multiplication but only a limited number of one of these operations.

There are many ways how homomorphic encryption can be used to achieve secure multiparty computation. The most straightforward is the two-party case of computation outsourcing, where an input party sends the ciphertexts to the computing party that uses the homomorphic properties to compute and return an encrypted result. However, both fully homomorphic and additively homomorphic encryption can be used in settings where all parties are computing parties and input parties. Some fully homomorphic encryption schemes, such as [37] also support distributed decryption [59], which makes them easy to use for multiparty computation. The idea is then that no party knows the secret key, and hence, all ciphertexts can be public and computations on them can be done publicly. Any ciphertext can be decrypted only if enough parties participate in the distributed decryption process.

An encryption scheme is specified by three algorithms – KeyGen for generating the keys, Encrypt for encrypting and Decrypt for decryption. The components needed to define the security of the encryption scheme are given in Figure 4 as defined in [72]. For semantic security (or chosen plaintext attacks (CPA)), the game generates a new keypair and gives the public key to the adversary. Then, the adversary can choose two messages m_0 and m_1 and the game that chooses and encrypts one of them. Finally, the adversary wins if it manages to correctly guess which of the two messages was encrypted. The success is measured as

$$2Pr[\text{CPA game returns True}] - 1$$

and the scheme has CPA security if this success is negligible.

A homomorphic encryption scheme also has an algorithm Evaluate that takes as input the public key, input ciphertexts and the computation specification and

produces the ciphertext of the computation result. The encryption scheme is correct if $\mathsf{Decrypt}(sk, \mathsf{Encrypt}(pk, m)) = m$ for any keypair (pk, sk) generated by KeyGen. The homomorphic evaluation is correct if the output of the evaluation can be decrypted to the same result as the evaluation would give on the plaintexts. A homomorphic encryption scheme is circuit-private if the distribution of the evaluation output is the same as the output of $\mathsf{Encrypt}$ on the circuit output value. This is formalised through the circuit privacy where the adversary specifies a computation C and a vector of input ciphertexts $\mathbf{c}$ and has to afterwards guess if the result it gets was generated through evaluation or encryption. Note that circuit privacy is achievable only if the evaluation also randomises the result somehow and the evaluation is not a deterministic function on the input ciphertexts or if the encryption scheme itself is deterministic. Circuit privacy enables secure computation where the computing party hides the algorithm used to compute the output. The success of the adversary in a circuit privacy game is measured as

$$2Pr[\text{Circuit privacy game returns True}] - 1$$

and the scheme has circuit privacy if this success is negligible.

Algorithm 8: CPA game

$(pk, sk) \leftarrow \mathsf{KeyGen}(1^k)$
$(m_0, m_1) \leftarrow \mathsf{A}(pk)$
$b \xleftarrow{\$} \{0,1\}$
$r \leftarrow \mathsf{Encrypt}(pk, m_b)$
$b \leftarrow \mathsf{A}(r)$
return $b == b'$

Algorithm 9: CCA1 game

$(pk, sk) \leftarrow \mathsf{KeyGen}(1^k)$
$(m_0, m_1) \leftarrow \mathsf{A}^{\mathsf{Decrypt}(\mathsf{sk},\cdot)}(pk)$
$b \xleftarrow{\$} \{0,1\}$
$r \leftarrow \mathsf{Encrypt}(pk, m_b)$
$b \leftarrow \mathsf{A}(r)$
return $b == b'$

Algorithm 10: Circuit privacy game

$(pk, sk) \leftarrow \mathsf{KeyGen}(1^k)$
$(C, \mathbf{c}) \leftarrow \mathsf{A}(pk)$
$b \xleftarrow{\$} \{0,1\}$
if $b = 1$ **then**
    $r \leftarrow \mathsf{Evaluate}(pk, C, \mathbf{c})$
else
    $r \leftarrow$
    $\mathsf{Encrypt}(pk, C(\mathsf{Decrypt}(sk, \mathbf{c})))$
$b \leftarrow \mathsf{A}(r)$
return $b == b'$

Figure 4: Security games of homomorphic encryption with security parameter k.

Chosen ciphertext (CCA) security can also be defined for homomorphic encryption, but it is difficult to achieve. The chosen ciphertext security game extends the semantic security game with the Decrypt query that allows the adversary to decrypt any ciphertext of its choosing. In the CCA1 security definition, these queries are only allowed before receiving the challenge ciphertext and in CCA2 security, they are always allowed with the exception that the challenge r cannot be decrypted using DECRYPT. However, CCA2 security is not achievable for homomorphic encryption in general since it is easy to derive a new ciphertext from the

challenge that can be decrypted using DECRYPT. For example, the new ciphertext could be just a re-randomised version of the challenge or a simple function computed from the challenge ciphertext. The latter is especially true if the scheme has circuit privacy and it is not possible to distinguish the fresh ciphertext from the computation result.

2.1.7. Combining Secure Computation Techniques

Previous three sections introduced concrete computation schemes. It is common for practical implementations of secure computation frameworks to combine several different computation techniques. For example, this has been done by TASTY [81, 90] and ABY [60, 117] frameworks and extensions [49, 109] of the ABY idea. In this thesis, an example of such a combination is presented in Section 4.8.2 that combines garbled circuits and additive secret sharing. This combination is motivated by the possibility to easily extend the set of protocols available to the user using automated tools to generate circuits for the garbled circuit method. Another driving force for combining different techniques is efficiency since different operations are efficient for the different techniques, and also, the computation technique of choice might depend on the actual deployment based on the available network and computational power of the participants. For example, these tradeoffs between protocol properties are discussed in [5].

The combination of different computation techniques is also very well supported by the popular online-offline paradigm first introduced in [57]. It is common that the online phase uses quite lightweight computations, for example, using secret sharing and could run with several different offline phases. This is well illustrated by the SPDZ protocol [59] that uses additive secret sharing in the online phase but, over its evolution summarised in [113], has had offline phases using somewhat homomorphic encryption [59], oblivious transfer [88], and additively homomorphic encryption [26].

The main component required for any combination of secure computation methods is a way to transform data representation from one technique to another. For example, to derive shares from ciphertexts or generate a ciphertext of the value from shares. For garbled circuits, the secure representation is less explicit. However, it is easy to define a circuit that outputs a ciphertext or takes either ciphertext or shares as input. Shares can also easily be given as output if the circuit supports private outputs to different parties. However, also the wire labels used in the garbled circuit can be considered to be a secure representation that could be directly translated into other representations. The conversion becomes more complicated in the active security model, where the data representation contains some authentication mechanism. However, conversions are still possible, as discussed in [128].

2.1.8. Arithmetic Black-Box

This section describes an arithmetic black-box view of formalising secure computation as it is needed for comparison with the formalisation of this thesis as given in Chapter 3. The arithmetic black-box (ABB) is a way of describing a secure computation functionality first proposed in [55]. It specifies secure computations like a secure general-purpose computer, where every party can send its inputs privately, and all parties (or a set of parties large enough to always contain at least one honest party) together can issue commands. In addition, only the command that outputs values can be used to return anything from the internal state of the ABB. No values other than explicitly published outputs are leaked from the ABB during the execution.

The internal state of ABB is a mapping from public handles to private values. Each party learns the handle corresponding to its input when it inputs the value and all corresponding computation commands are given using the handles. If a suitable set of parties specify the same operation with the same handles, then the functionality performs the operation on the values mapped to these handles and stores the output with a new handle. A command is only valid if the functionality has a value associated with the given handle. The publishing operation returns the value mapped to the handle used in the publishing command. An ABB can be invoked many times and it keeps its internal storage between the invocations. Hence, it is a stateful reactive functionality.

In general, the main ideal functionality of ABB for secure computation ($\mathcal{F}_{ABB}$) is commonly described as follows.

- On input (`input`, id, i) from honest parties wait for input (input, id, x) from party $\mathcal{P}_i$ and store x in memory with handle id.
- On input (`linear combination`, $\mathbf{c}$, $\mathbf{id}$, id_o) where $\mathbf{id} = (id_1, \ldots, id_k)$ and $\mathbf{c} = (c_0, c_1, \ldots, c_k)$ from the honest parties, retrieve values x_i corresponding to id_i from memory and compute $z = c_0 + \sum c_i x_i$. Store z with handle id_o.
- On input (`multiplication`, id_1, id_2, id_3) from the honest parties retrieve x_1, x_2 corresponding to id_1, and id_2 from memory and compute $x_3 = x_1 \cdot x_2$. Store x_3 with handle id_3.
- On input (`output`, id) from honest parties retrieve x corresponding to id from memory and send x to all parties.

Depending on the adversarial model, $\mathcal{F}_{ABB}$ is accompanied by additional capabilities. Commonly, each input is also reported to the adversary so that the adversary has some knowledge about the used handles and the computed operations. In the case of security with abort, the functionality allows the adversary to give the abort signal at any time and the signal is sent to all parties. As such, ABB is well suited to describe reactive functionalities and is clearly programmable. All conditional choices are made outside of the ABB based on the output values.

ABB defines a monolithic ideal functionality $\mathcal{F}_{ABB}$ for secure computations where the inputs and outputs are public values. There are no explicit shares for

parties and every interaction uses the handles of the values. Everything that is not explicitly public remains in the secure internal state of the functionality $\mathcal{F}_{ABB}$. By now, this approach is a common way to define the ideal functionalities of secure computation applications and engines. For example, it is used in [26, 59, 64, 103, 132].

2.2. Simulation Based Security

The security definitions of protocols commonly compare the protocol description to a so-called ideal functionality or the specification of the protocol. The proof is then carried out in a simulation paradigm, meaning that the proof proposes a simulator that translates a real-world adversary into an adversary against the ideal specification. This approach to security proofs originates from [75] but has been refined to several different frameworks that consider various details of the protocols. The overall idea of these proofs, as well as two concrete frameworks that apply this style to achieve universal composition, are introduced in this section.

Sections 2.2.2 and 2.2.5 introduce the visualisations used in the thesis and the core framework of the definitions. These are needed to follow the contributions of the thesis, the other sections as a general introduction to the topic. Section 2.2.1 and 2.2.3 give a more general overview before the concrete formalisations. Universal composability in Section 2.2.4 is given for comparison as this is the more commonly used formalisation. Finally, section 2.2.6 compares some aspects of the concrete formalisations and their applicability to this thesis.

2.2.1. Protocol Composition

The simulation-based proof can either prove security in a standalone or some composition setting. A standalone setting is one where only one instance of a single protocol is executed at a time. Hence, the adversary has no arbitrary information to use and has to work only with the information in the execution of the given protocol. It is more realistic to consider protocols in a composition setting where other protocols may be running either sequentially [40,112] or concurrently [11,40,116] with the protocol at hand and the adversary may play a part in some of those executions. Overall, if a protocol Π realises some functionality $\mathcal{F}$ with composable security, then Π will be indistinguishable from $\mathcal{F}$ in any context where it is run. This thesis considers the universal composition setting with no limit on the order or the existence of protocols outside the protocol of interest. There are different formalisations of such composition.

The main difference between universal and concurrent or sequential composition is the role of an environment that represents the world outside of the concrete protocol and its participants. The environment interacts with the protocol and chooses the inputs as well as receives all outputs.

The most commonly used formalisation is the universal composability framework by Canetti [41,42] introduced in Section 2.2.4. However, this work relies on

the reactive simulatability framework [9, 122] introduced in Section 2.2.5. Other frameworks for composable security include [10, 34, 39, 84, 94, 95, 106–108]. The concurrent general composition is equivalent with a version of universal composability as shown in [98], and there are cases where standalone security proofs also imply security under composition [61, 93].

2.2.2. Visual Representation of Protocol Structure

This thesis uses the following new visual representation of the protocol components. The protocols are formalized in the reactive simulatability framework, the components outlined here are specified in Section 2.2.5. The main elements of the visualization are summarised in Figure 5. The protocols are considered to consist of machines and network buffers. The concrete formalisation used in the thesis is the reactive simulatability framework introduced in Section 2.2.5. This section introduces the visual notation that can be used more generally for different formalisations of protocols. The machines can represent participants as well as functionalities. In the visual representation, machines are denoted by boxes with rounded corners (▢), and buffers are denoted by bullets and arrows. All computing nodes communicate with the outside world through ports denoted as square boxes at the border of machines. Input ports at the border of a simple machine are white (□) and output ports are grey (■). Ports are omitted if their direction is clear from the direction of the arrow denoting the connected buffers. Network communication is controlled by some participant through a clocking port. The clock signal is the signal given to the machine that is currently active and can be passed from one machine to the other to start the other machine. Ports for sending out clocking signals are denoted by bullets to visually separate communication and clocking ports. The master scheduler, who gets the control whenever no other machine is active, also has a special master clock in port, but it is not denoted explicitly.

Basic buffers leak nothing about transferred messages and have three ports: an input, output and clocking port. These stand for confidential and authentic network communication. These ports are not explicitly drawn since they are always connected to respective ports on some machine. An arrow with a bullet (—○→) denotes buffers that transport data in the direction of the arrow. The undirected component is used to denote the party that clocks the buffer. Dedicated notation is used for buffers clocked by a sender or a receiver, as illustrated in Figure 5. A grey box with a white circle (◘→) symbolises that the sender has full control over the clocking of the given buffer. A white box with a white circle on the receiver side (→◘) shows that the receiver has full control. An arrow without a buffer clocking symbol can be used to show buffers with undefined clocking in generic figures. If a buffer does not have an explicit clocker marked in the picture, then it is clocked by the master scheduler. In some cases, it is important for one machine to give control to another without any meaningful message. In this case, the

clocking connection ●—○ can be used between two machines with the grey bullet on the side of the clocker. A leaky buffer can be used to denote networking that leaks some information to the clocking party. A message in a leaky buffer has two components, a leaking tag t and a secret message m. A leaky buffer is denoted by —⊙→ and its idea is that it is clocked twice by the clocking party. First, the party can ask to receive the part of the message that is leaked and then it can clock the message to the recipient. If there are many messages then the clocking party chooses which to clock. The formal construction is discussed in Section 2.2.5.

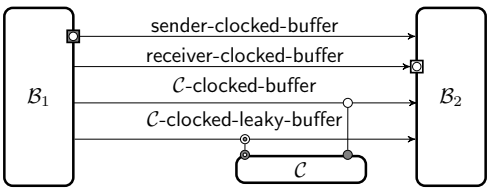


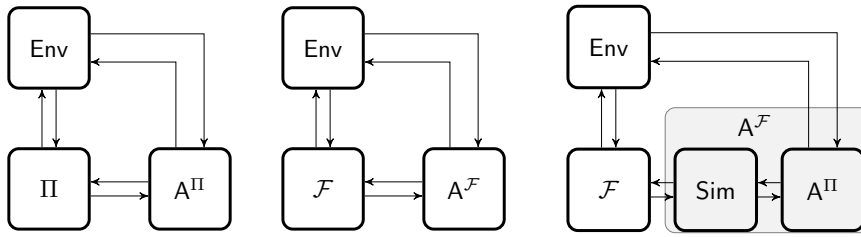
Figure 5: Notation for differently clocked buffers for sending data from machine $\mathcal{B}_1$ to machine $\mathcal{B}_2$ where some buffers are clocked by $\mathcal{C}$.

Machines, their ports, and buffers can be labelled with names if needed. Very often, it is reasonable to depict a part of a collection and not a complete collection. In this case, a dashed rectangle is drawn around the collection and the ports from machines outside of the collection that are used to connect to this collection are drawn to the border of the rectangle.

2.2.3. Protocol Equivalence

This section discusses some of the core ideas of simulation based proofs and protocol composition, for a more thorough introduction see [99]. The goal of this section is to have a general representation of the ideas and common aspects of the concrete frameworks of UC and RSIM introduced later. The notation used for protocol composition is specific to this thesis but represents general existing ideas. Independently of the concrete formalisation of the security proof, the overall goal of either standalone or composable simulation-based proof is to prove the indistinguishability of the real protocol and the ideal specification. For that, the interfaces of the real and ideal protocol have to match. However, this is usually not the case, as the real protocol commonly expects several rounds of interactions with a party, and the ideal functionality requires only the inputs of the party. Hence, commonly it is shown that there exists a simulator that interfaces with the adversary against the real protocol and the ideal functionality. The real, ideal and simulated executions are illustrated in Figure 6. The role of the simulator is to act as the protocol for the adversary and extract the inputs that the corrupted party has to the ideal functionality. In this case, it is actually shown that the real protocol is indistinguishable from the ideal protocol combined with the simulator. On the other hand, the real adversary and the simulator can be considered to be a new adversary against the ideal protocol. However, in the more general case, it

is possible to show the indistinguishability of any two protocols, not just the real and ideal ones.



(a) Real execution. (b) Ideal execution. (c) Simulation.

Figure 6: Execution of protocol Π, ideal functionality $\mathcal{F}$ with environment Env, adversary A^{Π} against the real protocol and $\mathsf{A}^{\mathcal{F}}$ against the ideal functionality, and a simulator Sim.

A protocol in real life is defined by a set of parties interacting with each other. In a formal representation, each party or other building block of the protocol is some kind of a computing machine and they are connected using network buffers. A collection of machines in a finite set of such machines and buffers connected to these machines. Each machine defines an interface that it uses to communicate with others. Each desired connection is defined as a port. Buffers shown in Figure 5 connect the input port of one machine to the output port of another machine. In order to compose protocols, they have to have matching interfaces where some ports of one machine can be connected to ports of another machine. A composition is well-defined if the respective ports of the two protocols are connected and there are no conflicts in the names of the port (e.g. machines are not allowed to have an output port with the same name). A composition of two protocols Π_1 and Π_2 is denoted as $\Pi_1\langle\Pi_2\rangle$. Note that the resulting collection is equivalent to $\Pi_2\langle\Pi_1\rangle$ and the notation is symmetric. However, in some cases, it is beneficial to give extra focus to the outer collection. It is also possible to shorten $\Pi_1\langle\Pi_2\langle\Pi_3\rangle\rangle$ to $\Pi_1\langle\Pi_2,\Pi_3\rangle$.

In the very general approach to security definitions, there exists a distinguisher $\mathcal{D}$ that interacts with two protocols Π_1 and Π_2 that have identical interfaces. The goal of $\mathcal{D}$ is to output the guess about which protocol it is interacting with. Let $\mathcal{D}\langle\Pi_i\rangle$ denote the distinguisher running the Π_i. Such a combination is meaningful if the interfaces are compatible and the composition is well-defined. More formal details of what compatibility means are given in the reactive simulatability description in Section 2.2.5. In the best possible case, Π_1 and Π_2 are perfectly equivalent and

$$Pr[\mathcal{D}\langle\Pi_1\rangle = 1] = Pr[\mathcal{D}\langle\Pi_2\rangle = 1]$$

for any distinguisher that matches the interfaces of Π_i. However, in a cryptographic proof, this is commonly unattainable and even unnecessary as the protocols are reasonably secure only in some specific contexts or with specific param-

eters. Moreover, it is sufficient if the probabilities are almost the same rather than exactly equal. Hence, indistinguishability can be defined as follows.

Definition 1 (Indistinguishability)**.** Two protocols Π_1 and Π_2 are indistinguishable if for any distinguisher $\mathcal{D}$ there exists a negligible function $\mu(\cdot)$ such that for every protocol input and security parameter $k \in \mathbb{N}$

$$|Pr[\mathcal{D}\langle\Pi_1\rangle = 1] - Pr[\mathcal{D}\langle\Pi_2\rangle = 1]| \leq \mu(k) \ .$$

This is denoted as $\mathcal{D}\langle\Pi_1\rangle \equiv \mathcal{D}\langle\Pi_2\rangle$.

The definition can be refined for a class of distinguishers rather than all possible. For computational indistinguishability, the protocols and the distinguisher are limited to be polynomial time in the security parameter. In the following, protocols that are indistinguishable are sometimes also called equivalent.

For simulation based proofs, the distinguisher is usually made up of two components - the environment Env and the adversary A as the protocol context in Figure 6a and Figure 6b. A protocol Π_1 is said to be as secure as Π_2 if there exists a construction $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\forall \mathsf{A}_1 \in \mathbb{A}_1, \forall \mathsf{Env} \in \mathbb{E}: \quad \mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$$

where $\mathbb{A}_1$ and $\mathbb{A}_2$ are the classes of adversaries and $\mathbb{E}$ is the class of allowed environments. For black-box simulation, the construction ρ is the simulator. $\mathbb{A}_1$ and $\mathbb{A}_2$ can be specified to consider the exact corruption model, such as passive or active adversaries and static or active corruption. Also, other restrictions, such as polynomial runtime, can be set on $\mathbb{A}_1$ and ρ. $\mathbb{E}$ defines the context in which the protocol is running and in which security is evaluated. Env acts as the distinguisher in Definition 1 and can similarly be restricted consider different flavours of security. The following alternates between this definition and the definition where the adversary construction is less implicit and simply denotes $\rho(\mathsf{A}_1)$ as A_2. The choice is made based on which is more natural in a given situation. The following sections discuss this setup and the following security definition and composition theorem with the concrete details of universal composability and reactive simulatability frameworks.

Definition 2 (Security)**.** Let Π_1 and Π_2 be collections with an identical interface for Env, and let $\mathbb{E}$ be the set of compatible environments. Let $\mathbb{A}_1, \mathbb{A}_2$ be the set of compatible adversaries. Then Π_1 is as secure as Π_2 (denoted as $\Pi_1 \geq \Pi_2$), if there exists a construction $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$$

for all $\mathsf{A}_1 \in \mathbb{A}_1, \mathsf{Env} \in \mathbb{E}$.

Protocols are often built from subprotocols, and especially this work mainly considers the composition of various protocols. Hence, it is crucial to be able to define and prove the security of a composition. The following is a version of the

composability theorem stated in the terms used in this thesis. Concrete versions of this theorem are discussed for the UC and RSIM framework in the following sections.

Theorem 1 (Secure two-protocol composition)**.** *Let* Π_e, Π_1, Π_2 *be three collections such that collections* $\Pi_e\langle\Pi_1\rangle$ *and* $\Pi_e\langle\Pi_2\rangle$ *are well-defined and have an identical interface for* Env. *Let* $\mathbb{E}$ *be the subset of compatible environments and let* $\psi : \mathbb{E} \to \mathbb{E}^*$ *be a natural construction* $\psi(\mathsf{Env}) = \mathsf{Env}\langle\Pi_e\rangle$. *If there is a construction* $\phi : \mathbb{A}_1 \to \mathbb{A}_2$ *that proves* $\Pi_1 \geq \Pi_2$ *for the set of environments* $\mathbb{E}^*$ *then this construction is also a proof for* $\Pi_e\langle\Pi_1\rangle \geq \Pi_e\langle\Pi_2\rangle$ *for the set of environments* $\mathbb{E}$.

The idea of ψ is simply to form a new environment from the initial environment Env and the protocol Π_e. The following sections consider more details of the security definition as well as the composability results in the universal composability and reactive simulatability frameworks in Section 2.2.4 and Section 2.2.5, respectively.

2.2.4. Universal Composability

This section introduces the universal composability framework for comparison with reactive simulatability and the MPC formalisation of this thesis. The universal composability (UC) [41, 42] framework provides a formal framework for describing cryptographic protocols and analysing their security. The strength of the framework is the composition theorem stating that any protocol proven secure in this framework remains secure when run in composition with arbitrary protocols. As common in simulation-based proofs, the security is defined with respect to an ideal functionality, and the proof requires defining a simulator that translates a real-world adversary to the ideal-world adversary. Overall, the simulator has two tasks: managing the timing of the protocol and translating the messages between the two interactions. In more detail, the setup consists of an environment (Env), the protocol Π and an adversary A. The environment represents everything that can happen in the world outside of the concrete protocol execution. Specifically, it provides inputs to the protocol, collects the outputs and interacts with the protocol during the execution. It also interacts with the adversary in an arbitrary way. A protocol Π is said to UC-securely implement a functionality $\mathcal{F}$, if, for any adversary A against Π, there exists an ideal adversary $\mathcal{S}$ against $\mathcal{F}$ such that no environment Env can distinguish between interacting with $\mathcal{S}$ and $\mathcal{F}$ or A and Π.

Informally, the composition theorem, which is the concrete formalisation of Theorem 1 for the UC framework, states the following. Consider a protocol $\Pi_e\langle\mathcal{F}\rangle$ that operates in a hybrid model where parties can communicate as usual and also have access to some ideal functionality $\mathcal{F}$. Let Π be the protocol that UC-securely implements $\mathcal{F}$. Then $\Pi_e\langle\Pi\rangle$ is the composed protocol where Π is used instead of $\mathcal{F}$. According to the UC theorem, in these cases, $\Pi_e\langle\mathcal{F}\rangle$ and $\Pi_e\langle\Pi\rangle$ are indistinguishable. As a special case if $\Pi_e\langle\mathcal{F}\rangle$ UC -securely implements a func-

tionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model then $\Pi_{\mathsf{e}}\langle\Pi\rangle$ UC-securely implements $\mathcal{G}$. Note that Π_{e} can run many instances of Π.

The main composition results hold for the cases where the internal states of composed protocol instances are independent of each other, at least for the honest parties. However, it is common to share some state, for example, using the same cryptographic keys in different instances. In such cases, if any information about the keys leaks in one protocol instance, it can easily be used to attack other instances. Hence, analysing one instance independently from others with the shared state requires extra care to account for the possibility of the joint state being revealed somewhere else. A shared state can be considered if the set of instances sharing the state is considered as one big protocol. However, this could defeat the purpose of the modular approach allowed by the composition theorem, as one can no longer simply work with a simple representation of the protocol. The version of universal composability with the joint state was introduced in [47] to handle some joint state and shared randomness. The modular formalism of this thesis explicitly considers setup as one part of the secure computation formalism and considers composing protocols in this specific context rather than a more general shared setup. Hence, shared state composition results are not explicitly needed.

The main UC framework is very generic to capture various possible details of different cryptographic protocols. In addition, it has significantly changed over time to both fix bugs (e.g. definition of polynomial time) and add more expressibility. A simpler variant of universal composability (SUC) for secure multiparty computation was defined in [44]. The main simplification compared to the generic UC is that the MPC version assumes that all participants are fixed ahead of time. In addition, all communication channels are authenticated while still leaving the adversary in control of the timing. On the formal side, these restrictions simplify defining polynomial-time execution. A similar simplification also appeared in [136].

On the networking side, SUC defines all machines in a star network with a router machine in the centre and all other machines connected only to the router. The adversary controls the message timing via the router machine. Note that the adversary sees all the messages, but they are authenticated. Hence, the adversary cannot change the messages in the router. To model private messages, the messages can have two parts: a public header and private content. In such a case, the adversary sees only the header that has to contain all the information expected to be available to the adversary, for example, the length or type of the message. The MPC formalisation in Chapter 3 also uses a similar approach to network messages.

As usual, the adversary is allowed to corrupt parties and control their actions (either semi-honestly or maliciously). Similarly to UC, SUC also defines the environment Env as an interactive distinguisher for the executions. The composition is defined as replacing the program code sending the message to the ideal functionality with the code for the real functionality. Hence, in SUC, the subroutines

are executed internally, whereas, in UC, they are executed as separate interactive Turing machines. The set of machines running in the SUC framework is always fixed beforehand. Hence, the polynomial-time execution is guaranteed if each separate machine runs in polynomial time.

SUC also limits the adversary's capabilities. Each party is either fully corrupted or not corrupted. A corrupted party is under the control of the adversary (either actively or semi-honestly). The simplifications do restrict the types of protocols that can be expressed in SUC compared to UC. However, SUC suffices for secure function evaluation as well as secure reactive computations. In addition, the simplifications only affect the types of protocols that can be considered. It is shown in [44] that any protocol proven secure in the SUC model can be automatically transformed into a protocol secure in the UC model.

2.2.5. Reactive Simulatability

This section describes the reactive simulatability framework that is the basis of the formal definitions in this thesis. Reactive simulatability [9, 122] (RSIM) is another way of defining the conditions that need to hold for one protocol to securely implement another protocol or functionality. Note that there are slight variations in different versions of RSIM [8, 9, 120–122], and the versions used here [9, 122] consider the composition of a fixed number of parties and protocol instances with asynchronous communication. In the overall intuition, there are many similarities with the universal composability framework. However, there are differences in the formal details. Still, in many cases, also this formalisation is called by the name universal composability. Overall, the RSIM idea stems from definitions of one system implementing another in the distributed systems world and adds a confidentiality layer to these. Reactive means that the environment and the system can interact multiple times, and asynchronous means that the adversary may control delays in the message delivery.

In RSIM, the protocol interacts with the honest users represented by Env and the adversary A that are essentially only distinguished by the purpose of the communication. Honest users and the adversary can interact arbitrarily. A protocol is secure with respect to universal reactive simulatability if, for every adversary against a protocol Π, there exists an adversary $\mathcal{S}$ against the specification $\mathcal{F}$ such that, for all honest users Env, the view of the honest user is indistinguishable in the two settings. Notably, similarly to the UC definition, the following introduction focuses on the case where the adversary against the ideal functionality does not depend on the honest users and is derived only from the adversary against the real system which is called the *universal* reactive simulatability. It is also possible to define *general* reactive simulatability where the derived adversary can depend on both the initial adversary and the honest user but this is not used here. In addition, if the ideal adversary uses the real adversary without knowing its internals as in Figure 6c then this is called *black-box* reactive simulatability. This section

introduces the RSIM framework in its original form and connects it to the notions and notations used in Section 2.2.3 as this thesis uses both variations depending on the level of detail that is needed.

Specification and Notation. A participant or functionality in the RSIM framework is modelled as a *simple machine* specified by tuple $(Name, Ports, States, \sigma, Ini, Fin)$. It is a probabilistic state transition machine with the name *Name*. *Ports* is a sequence of ports of that machine used to connect to other machines via buffers. *States* is a set of states and σ is a probabilistic state transition function. *Ini* and *Fin* are the sets of initial and final states. When a machine starts, it reads inputs on its input ports and takes the next transition producing the new state and the messages to the output ports. In addition, the length of the input that it reads is bounded by the state, and the longer parts of the input will be ignored.

Data transfer from one machine to another goes through a buffer that connects an output port $p!$ of the first machine with an input port $p?$ of the second machine. A *buffer* is a dedicated machine that has exactly three ports: clock-in port, buffer in-port and buffer out-port. Messages written by other machines are written in the buffer and then clocked out from there to deliver them to the recipients. The shorthand p^c (complement of p) denotes a port that is connected to p via a buffer. For a set S of ports, S^c can be used similarly.

Commonly a protocol is made up of several machines interacting with each other as well as the honest user Env and the adversary A. As an illustrative example, consider the most common setting in the RSIM framework depicted in Figure 7. In this configuration, a collection $\mathfrak{M}$, composed of several subsystems and parties, is under the influence of an adversary A and an environment Env, which uses the collection to obtain an output. In addition, there is a buffer denoted by an arrow. The collection can have any number of machines. Figure 7 considers two machines M_1 and M_2 explicitly, but for any other machines M_i, the connections are similar. Finally, the adversary A schedules all buffers in Figure 7.

The execution of the machines is controlled by clocking and inputs. Exactly one machine is active at any time. The execution goes according to the following rules.

- A machine is clocked when it receives an input, and then it performs its next state transition and may write messages to its output buffers.
- Any machine can clock at most one of the output buffers by sending an input to a clock-in port of the buffer.
- Upon a clocking input, the buffer releases the value indicated by the clocking party and the recipient of the value is clocked.
- If no machine is scheduled, then the control goes to a machine dedicated as the master scheduler.

The execution ends when the master scheduler reaches a final state. In other words, the execution continues as long as the master scheduler clocks the next steps. The master scheduler cannot do anything useful with other machines if

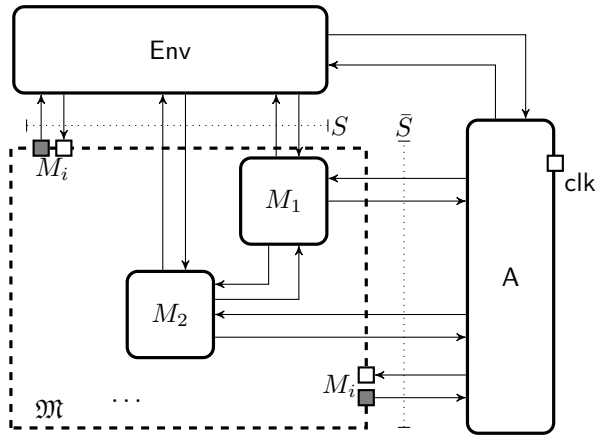


Figure 7: Canonical interface between adversary A, honest user Env and system consisting of the collection $\mathfrak{M}$ of machines M_i with the dedicated ports S and adversary ports $\overline{S}$. A also has a master clock-in port clk that is omitted from most figures.

they have reached their final state. However, the master scheduler can finish the execution when other machines are not yet in the final state.

A finite set of machines is called a collection $\mathfrak{M}$. Each machine in a collection must have a unique name and unique ports. The connections between machines can be derived from the port names. Writing $ports(\mathfrak{M})$ stresses that the ports belong to the collection $\mathfrak{M}$ of machines. The set $ports(\mathfrak{M})$ can be further split into the ports that connect machines in the collection $\mathfrak{M}$ and the set of free ports $free(\mathfrak{M})$ that can be accessed from outside, for example by A or Env. A collection is closed if only *master clock-in port* is free. The master clock-in port is an input port that is used to activate the collection. Commonly, a collection only contains simple machines. If the set of machines also includes all the buffers connected to the ports (other than the master clock-in port), then it is called a *completion* $[\mathfrak{M}]$. The set of buffers that connect machines inside the collection with each other is defined by the ports of the machines. Similarly, the set of buffers that connect the machine in a collection to the machines outside of the collection is clearly defined. Hence, a collection uniquely defines a completion and vice versa.

A *structure* $(\mathfrak{M}, S)$ is collection of simple machines $\mathfrak{M}$ together with a set of ports $S \subseteq free(\mathfrak{M})$. Let $\overline{S} = free(\mathfrak{M}) \backslash S$. Ports in S are for communicating with the environment Env, and ports in $\overline{S}$ are for communicating with the adversary A as shown in Figure 7. To avoid confusion with communication, Env cannot have ports belonging to S or ports used to send messages between machines in $\mathfrak{M}$. These *forbidden* ports are denoted by $forb(\mathfrak{M}, S)$.

A system *Sys* is defined as a set of structures. These structures commonly correspond to different sets of statically corrupted parties. However, this thesis focuses on the more generic case of adaptive corruption that can be modelled by systems with exactly one structure. The downside is that the machines are

more complicated and accept corruption requests during their work. Essentially, each machine can then either work in the honest or corrupted mode. In such a setting, static corruption can be modelled by limiting the class of adversaries to the ones that only corrupt parties before any other action has happened in the protocol. Hence, for current purposes, $Sys = \{(\mathfrak{M}, S)\}$ and the following RSIM overview refers only to this case. In this work, the adversary has full control over the protocol and message scheduling but does not see the messages of uncorrupted parties. The interface $\overline{S}$ specifies which messages the adversary can affect and which machines it can corrupt.

A *configuration of a system* is a tuple $(\mathfrak{M}, S, \mathsf{Env}, \mathsf{A})$ where Env is a simple machine without forbidden ports $forb(\mathfrak{M}, S)$ and the completion $[\mathfrak{M} \cup \{\mathsf{Env}, \mathsf{A}\}]$ is closed with A being the master scheduler. A shorthand $Conf(Sys)$ denotes the set of all valid configurations for a system Sys. Note that, as for most collections, the structure is implicitly clear from the description and in such cases, the set of configurations is derived from the set of all compatible adversaries and environments as in Definition 2.

Security definitions. Security in the RSIM framework is defined by contrasting two systems Sys_1 and Sys_2, where Sys_2 is the system that is known to be secure, and Sys_1 is the designed system. Both systems must have the same interface S towards the environment Env, and there must exist a way to convert a valid adversary A_1 against $(\mathfrak{M}_1, S) \in Sys_1$ to a comparable adversary against $(\mathfrak{M}_2, S) \in Sys_2$.

A *run* of a closed collection is a sequence of steps of the machines where machine N received input v_{in} from an input port $c_{\mathsf{in}}?$, went to state s writing v_i to output port $c_i!$ and possibly clocking the channel c_{clk}. Denote the steps of a run as $(N, c_{\mathsf{in}}, v_{\mathsf{in}}, s, ((c_1, v_1), \ldots, (c_n, v_n)), c_{\mathsf{clk}})$ A *view* of a set of parties $M_0 \subseteq \mathfrak{M}$ is a subset of a run that corresponds to the steps relating to machines in M_0 as is denoted by $view(M_0)$. Note that the view does not contain port names, and therefore, it is allowed to rename ports and buffers when composing systems.

Definition 3 (Simulatability)**.** Let systems Sys_1 and Sys_2 with identical sets of free ports be given. Sys_1 is perfectly as secure as Sys_2 if, for every configuration $conf_1 = (\mathfrak{M}_1, S, \mathsf{Env}, \mathsf{A}_1) \in Conf(Sys_1)$ where $ports(\mathsf{Env}) \cap forb(\mathfrak{M}_2, S) = \emptyset$, there exists a configuration $conf_2 = (\mathfrak{M}_2, S, \mathsf{Env}, \mathsf{A}_2) \in Conf(Sys_2)$ such that the environments have coinciding views

$$view_{conf_1}(\mathsf{Env}) = view_{conf_2}(\mathsf{Env}) \quad .$$

If A_2 does not depend on Env, then this property is called universal simulatability. Perfect security is denoted as $Sys_1 \geq_{sec}^{perf} Sys_2$.

Definitions for computational and statistical security can be obtained similarly by restricting the set of plausible configurations and by varying the requirements on the views. Computational security $Sys_1 \geq_{sec}^{poly} Sys_2$ requires computational indistinguishability of views created by all polynomial configurations $Conf(Sys_1)$.

For statistical security $Sys_1 \geq_{sec}^{stat} Sys_2$, statistical indistinguishability of views for all configurations $Conf(Sys_1)$ is required. The running-time of A_2 must be polynomial in the running-time of A_1. Universal simulatability is in line with the security definition in Definition 2, where the adversary A_2 depends only on A_1.

Black-box simulatability means that $\mathsf{A}_2 = \mathsf{Sim} \cup \mathsf{A}_1$ must be the *combination* of a simulator machine Sim and the adversary A_1. The original machines and the buffers between them can be thought of as the submachines of the combination. For black-box simulatability, the *simulator* Sim depends only on $\mathfrak{M}$ and the set of ports S. It uses A_1 without knowing its internals. As the master scheduler A_1 is in the combination, then the combined machine A_2 becomes the new master scheduler. Some of the internal buffers are clocked by A_1, others by Sim that also uses the clock signals by A_1. Another way to consider black-box simulatability is to consider the simulator as part of the system so that the original system and the second one with the simulator are equivalent.

The definition can also be restricted to some classes of adversaries. For example, in terms of Definition 2, the set of all compatible adversaries could be limited to compatible adversaries with some known properties.

Definition 4 (Simulatability for a class of adversaries)**.** Given systems Sys_1 and Sys_2, Sys_1 is perfectly as secure as Sys_2 for a class $\mathcal{A}$ of adversaries if, for every configuration $conf_1 = (\mathfrak{M}_1, S, \mathsf{Env}, \mathsf{A}_1) \in Conf(Sys_1)$ where $\mathsf{A}_1 \in \mathcal{A}$ and Env has no forbidden ports $(ports(\mathsf{Env}) \cap forb(\mathfrak{M}_2, S) = \emptyset)$, there exists a configuration $conf_2 = (\mathfrak{M}_2, S, \mathsf{Env}, \mathsf{A}_2) \in Conf(Sys_2)$ with $\mathsf{A}_2 \in \mathcal{A}$ such that

$$view_{conf_1}(\mathsf{Env}) = view_{conf_2}(\mathsf{Env}) \ .$$

This is denoted as $Sys_1 \geq_{sec}^{perf,\mathcal{A}} Sys_2$.

Security of compositions. The main result of the RSIM framework assures that if one system is proven to be as secure as the other, then it can be used to replace the other one in any composition without compromising the security of the composition. A composition of systems in the adaptive adversary case used in this thesis means that the structures in the systems are connected through their complementing ports, and each system has one structure.

Definition 5 (Composability and composition)**.** Structures $(\mathfrak{M}_1, S_1), \ldots, (\mathfrak{M}_n, S_n)$ are *composable* if they have compatible port layouts:

$$\forall i \neq j: \ ports(\mathfrak{M}_i) \cap forb(\mathfrak{M}_j, S_j) = \emptyset \ ,$$
$$\forall i \neq j: \ S_i \cap free([\mathfrak{M}_j])^c = S_j^c \cap free([\mathfrak{M}_i]) \ .$$

Their *composition* $(\mathfrak{M}_1, S_1)||\ldots||(\mathfrak{M}_n, S_n)$ is a structure consisting of all machines $\mathfrak{M} = \mathfrak{M}_1 \cup \ldots \cup \mathfrak{M}_n$ that has the interface $(S_1 \cup \ldots \cup S_n) \cap free([\mathfrak{M}])$ for communicating with the environment.

For any systems Sys_1 and Sys_2, let $Sys_1 \circ Sys_2$ denote the system of all valid compositions $(\mathfrak{M}_1, S_1)||(\mathfrak{M}_2, S_2)$ for $(\mathfrak{M}_1, S_1) \in Sys_1$ and $(\mathfrak{M}_2, S_2) \in Sys_2$. Structure $(\mathfrak{M}_2, S_2) \in Sys_2$ is composable with Sys_1 if there exists $(\mathfrak{M}_1, S_1) \in Sys_1$ such

that $(\mathfrak{M}_1, S_1)||(\mathfrak{M}_2, S_2) \in Sys_1 \circ Sys_2$. Hence, when composing systems, only composable structures of the systems are composed. The system composition leads to the following theorem regarding the security of the composed system, where it is assumed that the corresponding structures in the two systems are identified by the set of ports intended for the environment. For the cases where each system is represented by one structure and the sets of dedicated ports is implicit, the composition can also be denoted as $\mathfrak{M}_1\langle\mathfrak{M}_2\rangle$ or $\mathfrak{M}_2\langle\mathfrak{M}_1\rangle$. In addition, the rules regarding matching ports in composable structures can be simplified to say that the composition is well-defined, as was done in Theorem 1.

Theorem 2 (Secure two-system composition)**.** *Assume that there are three systems Sys_1, Sys_1', Sys_2 such that $Sys_1 \geq_{sec} Sys_1'$. If, for every structure $(\mathfrak{M}_2, S_2) \in Sys_2$ that is composable with $(\mathfrak{M}_1, S_1) \in Sys_1$ and for every structure $(\mathfrak{M}_1', S_1) \in Sys_1'$ the composition $(\mathfrak{M}_1', S_1)||(\mathfrak{M}_2, S_2)$ exists and satisfies $ports(\mathfrak{M}_1') \cap S_2^c = ports(\mathfrak{M}_1) \cap S_2^c$, then $Sys_1 \circ Sys_2 \geq_{sec} Sys_1' \circ Sys_2$.*

Note that the theorem holds for various restricted cases of simulatability such as statistical, perfect and (for polynomial-time Sys_2) computational security as well as universal and black-box simulatability and different restricted classes of adversaries.

Time Complexity. One of the complicated details of such formalisations is the definitions of the running times of the machines needed for computational security. It is important to establish which machines and collections or compositions run in polynomial time. Each machine can be thought of as a Turing machine with work tape keeping the state of the machine, communication tapes, output tape and other special tapes if needed.

A machine is said to be polynomial time if its number of steps is polynomial in the security parameter (represented as the initial content of the work tape). The notion where the time can also depend on the content of the input tapes is called weakly polynomial. However, a combination of weakly-polynomial machines may have a running time that is exponential in the security parameter. As a result of this definition, a collection runs in polynomial time if all its machines are polynomial time, and a view of a polynomial-time collection is polynomially bounded. A system is polynomial time if the machine collections in all its structures are polynomial time. A configuration is polynomial time if the structure, honest user, and adversary are all polynomial time.

Note that transparent machines like a corrupted party simply forwarding its communication to and from an adversary is not polynomial and is only weakly polynomial. This is because a corrupted party may read an unbounded number of input bits, especially those coming from an adversary. However, if the rest of the system ensures that the size of the inputs is bounded by the expected inputs in the uncorrupted case, then the collection will still run in polynomial time. The following formalisation of secure computation in Chapter 3 satisfies this as it will be shown that the adversary has only limited meaningful interactions with each cor-

rupted party, and therefore the systems will be polynomial time. In addition, the RSIM model allows to include bounds on the input that the machine reads which can also be used to ensure that corrupted machines can not exceed polynomial time.

Leaky Buffers. The following formalisation of secure multiparty computation frameworks extends the RSIM buffers with a leaky buffer. Such buffers are used to consider communication that is mostly secure but where the adversary may learn some metadata about the communication. Each message in such a buffer is a tuple (m,t) where m is the protected message and t is the message tag that can be seen by the adversary. This achieves a similar effect as the metadata that leaks from the router in the SUC model.

Leaky buffers can be constructed from the standard buffers and a simple machine in the RSIM framework. The construction in Figure 8 has a message tagging machine $\mathcal{T}$ and three buffers for input, output and leakage, respectively. The machine $\mathcal{T}$ accepts only pairs of strings (m,t) as inputs from b_1 buffer. When clk_1 is clocked then $\mathcal{T}$ receives (m,t) from the buffer b_1. The pair (m,t) is written to an output buffer b_2, and the annotation t is written to the sender-clocked buffer b_3 and clocked. There are two ports for clocking the leaky buffer, but the assumption is that all ports clk_1, clk_2 and type belong to the same machine. The first port clk_1 determines when the annotation arrives at $\mathcal{T}$ and, therefore, to b_2 and b_3. The second port clk_2 controls when and in which order message pairs (m,t) are written to the port out and is analogous to the clocking of the regular RSIM buffer.

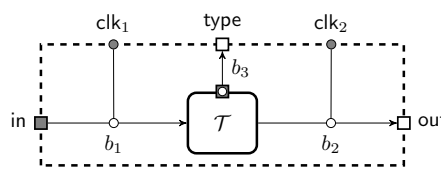


Figure 8: Construction of a leaky buffer from a simple machine and regular buffers.

Leaky buffers are used for buffers that are under some amount of adversarial control. Note that all regular buffers could be replaced with leaky buffers by giving both clock signals and the leak port to the party clocking the message and using a dummy tag t in order not to leak additional data. However, the following chapters of the thesis use all types of buffers to make it explicit where the leaking tags are needed. The graphical notation for the leaky buffers adds a dot to the regular buffer notation as shown in Figure 5 for a leaky buffer clocked by $\mathcal{C}$. They are clocked by the adversary or the simulator unless explicitly denoted otherwise.

2.2.6. Comparison of Simplified UC and RSIM

In the very general view, both universal composability and reactive simulatabiity frameworks define their own detailed version of the security Definition 2 and the

security of composition as generalised in Theorem 1. It is hard to compare the two frameworks in detail because both have different versions of concrete definitions and have evolved over time. A summary of the comparison of universal composability, simplified universal composability, reactive simulatability, and the formalisation of MPC that appears in Chapter 3 is given in Table 1. The following of this section focuses on RSIM and SUC. Some more comparisons with the formalisation of this thesis can be found in Section 3.7.

In the big picture, SUC and the adaptive adversary version of universal RSIM are quite similar to each other. In all cases the security definition (version of Definition 2) can be adapted to use different security models like computational and information-theoretic. Notably, the composition theorems (version of Theorem 1) also give similar guarantees. However, for SUC and UC, the composability theorem is only stated for polynomial-time adversaries and environments. For RSIM the composition theorem explicitly considers all versions of computational, statistical and perfect security where these parties are not restricted. In the UC model and RSIM the machines can define different corruption rules, for example, is UC a party can be partially corrupted or become uncorrupted. In SUC and the model of this thesis if a party is corrupted then all its control is given to the adversary.

First of all, both SUC and RSIM formalisms divide the protocol to essentially the environment (or honest user), the adversary, and then the protocol at hand. Both models work with a fixed number of parties and give the adversary control over the timing of the execution. Both SUC and RSIM disallow the dynamic generation of new machines during the protocol execution. However, for SUC, the communication with the environment differs from the rest of the communication model and its timing is not under the control of the adversary (or passing through the router). This control is not relevant for corrupted parties as the adversary controls their timing. However, for honest parties the environment can read the output as soon as an honest party has computed it which is more in line with the general UC setup.

In the SUC model, there is the restriction that two copies of the same protocol in the hybrid model call different copies of their ideal sub-protocol. This is ensured by using suitable session identifiers and it assures that the two copies of the protocol do not share the state through the ideal functionality that they use. The formalism in Chapter 3 is built to support the same idea that the ideal functionalities can be separated for the different protocol runs.

One notable difference is the approach to networking management. The SUC model uses a star network with a router that is under adversarial control, and RSIM gives similar capabilities with the clocking control over the buffers. The public metadata that the adversary can read in the router can also be added to RSIM buffers, as done in Section 2.2.5. Note that, despite of the central router, the adversary in SUC cannot forward the messages to other parties than the intended recipient. Hence a router could be thought of as an interface to a collection of independent buffers. Note that the authenticity of the network channels with

Table 1: Comparison of the properties of different protocol formalisations.

Property	UC	SUC	RSIM	This thesis
Security model	Computational, information-theoretic	Computational, information-theoretic	Computational, information-theoretic	Computational, information-theoretic
Parties, functionalities	Can be added dynamically	Fixed before execution	Fixed before execution	Fixed before execution
Network configuration	Point-to-point communication channels	Star network with an adversary-controlled router	Point-to-point buffers with a known party clocking them	Point-to-point buffers with a known party clocking them
Network security	Insecure, authentic and confidential all possible	Authentic and confidential with well-defined leaks	Insecure, authentic and confidential all possible	Authentic and confidential with well-defined leaks
Composing	Separate functionality is called	Separate ideal functionality is called, for a real protocol the code of the protocol is inlined to the calling party	Separate functionality is called	Separate functionality is called
Corruption	Possible to define corruption rules	Party is corrupted fully	Each machine can define corruption rules	Party is corrupted fully

potential confidentiality has been defined as part of SUC, whereas RSIM can also cover insecure network channels. However, the MPC formalisation given in Chapter 3 of this thesis focuses on the case of RSIM with authenticated and (partially) confidential channels.

In RSIM and UC, the subroutine that is composed is a separate machine (or a collection of machines). The SUC model defines it by inlining the code of this machine to the calling machine. Note that in RSIM, either the calling machine or the master scheduler may be responsible for starting the subroutine. Hence a similar effect to inlining in SUC can be achieved by giving the calling machine control over the clocking of the buffer to the subroutine. In SUC, all channels between machines are controlled by the adversary.

From the computational perspective, both formalisms define machines as probabilistic interactive Turing machines with small differences in the details of the machine setup. In SUC, the polynomial-time machine is defined as polynomial in its input from the environment and the security parameter. In RSIM, the inputs from the environment are not differentiated from the inputs on other ports and polynomial time is defined only based on the security parameter with the addition that a machine reads only an expected length of the inputs (the length is a function of the state of the machine). In both cases, since the number of machines is fixed, then also the running time of a system is polynomial if all machines are.

SUC was defined for formalising secure multiparty computation and also defined some things that one needs to specify separately in RSIM. For example, all calls to ideal functionalities begin with a session identifier. Respectively, the functionality runs several copies of its code, one for each received session identifier. Chapter 3 will define similar behaviour for ideal functionalities later. Similarly, the formalisation of this thesis assumes that each party is either corrupted or honest. Especially if a party is represented by multiple machines, then one must assume that either all of these machines are corrupted or none are. Hence, parts of Chapter 3 that sets up the formalism of secure multiparty computation can be seen as defining a specialised RSIM framework similar to the relationship between UC and SUC.

This thesis uses the RSIM framework over SUC because of the flexibility and the clear structure of keeping the separate machines in composition. In addition, in comparison to UC, it is acceptable to choose RSIM because its restrictions, compared to the expressibility of UC, match with the ones that are reasonable for secure computation anyway. Hence, choosing RSIM will not limit the formalisation in Chapter 3 compared to an analogous formalisation that could be established using UC as the underlying definitional framework.

3. FORMALISATION OF MPC

This chapter proposes a new formalisation of secure multiparty computation that can be used independently and is the basis for the main results of this thesis. An MPC framework is represented in this formalisation as a protection domain consisting of setup, data storage and computation functionalities. The formalism is based on the RSIM framework and certain assumptions that are generally made towards reasonable data protection schemes and secure computation functionalities. Section 3.1 gives a high level overview of the core ideas and references the following sections that formalise these ideas.

3.1. Protection Domain Formalisation Overview

An MPC framework has some means of secure data storage and some computation protocols. Parties can interact with the framework to execute some computations on their private inputs and to get outputs from the system. The protection domain look at MPC is intended to keep this modular view. A protection domain contains all necessary operations, such as setup, inputs, outputs, and computations. Parties interacting with the protection domain each know their inputs and they hold their view of the intermediate data. For example, each party knows its share of secret shared values.

The protection domain (formalised in Section 3.6) formalisation uses storage domains (Section 3.3) to generalise different means of secure data representations. Generally, a storage domain defines methods to protect and reveal data so that information is not disclosed to the participants before the storage domain explicitly does so. The following calls these operations sharing and reconstruction, but the same approach applies to other data protection means, such as encryption. The general properties required from the data protection schemes are discussed in Section 3.3. In short, the storage domain is hiding if it ensures the confidentiality of its contents. In secure computation, the stored values are used to derive new stored values, and there are places where the data representation may become corrupted or where it could be adversarially modified. Such properties are formalised through modification awareness and limited control. The modification awareness specifies that if the adversary modifies the representation of data held by the corrupted party, it knows how it will affect the stored value. Limited control is the reverse property that any allowed value modification can be achieved by changing only the corrupted view of the representation.

A protection domain often contains a setup functionality used for both computation protocols and the storage domain. The properties that a proper setup has to obey are discussed in Section 3.2. The overall assumptions are that the setup cannot be corrupted and always gives consistent setup data to all parts of the protection domain.

The security of secure computation is defined through ideal functionalities that

can be generic. A protection domain itself can be seen as a specific ideal functionality. However, it is more interesting to consider its components as separate functionalities to achieve a modular approach. Computation functionalities that are reasonable and intuitive have some simple properties. The overall structure is that they reconstruct the secret value, compute the desired operation in plain and then share the result. Such functionalities are fully specified in Section 3.4. This definition of ideal functionalities excludes the class of computations done without interaction. Such protocols cannot leak new information to the participants, but there has to be a way to consider them as building blocks of composed protocols. These functionalities are called local and their details are given in Section 3.5.

The protection domain is built from the previously listed components as discussed in Section 3.6. Overall, the idea is that a protection domain is a simple collection of the computation functionalities. The functionalities themselves contain the pieces of the storage domains to reconstruct and share secrets. The parties have an ongoing interaction with the protection domain to compute something. They get intermediate outputs from the functionalities and give these as inputs to the following functionalities.

Section 3.7 summarises this chapter and compares the MPC formalisation using protection domains defined in this thesis with other approaches of formalising secure computation frameworks.

3.2. Trusted Setup

Secure multiparty computation often uses some form of setup. In fact, achieving universal composability can be impossible without it [43, 123]. Even if it is not necessary from the theoretical perspective, a setup phase can help achieve more efficient protocols. For example, it is common to have setup to establish secure channels or distribute cryptographic keys. Depending on the needs of the protocol, the setup phase can give both public and private parameters to parties. For schemes that can run on various algebraic structures or different number of parties, the setup could specify the number of parties, the used data structures and the threshold of the protection scheme. These parameters are available to all participants. On the other hand, private parameters are intended only for one or some of the participants and include mostly private keys.

In the following formalisation, the setup is considered a separate functionality expected to give coherent parameters to all functionalities requiring them. The trusted setup or the ideal functionality of the setup phase is denoted as $\mathcal{F}_{\triangle}$. In practice, this is realised by some protocol $\Pi_{\triangle}$. The setup distribution is expected to be the first step in the protocol execution, and all parties that require setup receive it during the first activation of $\mathcal{F}_{\triangle}$. For any protocol, running setup is a one-time occurrence in a protocol. Also, all machines get the same public parameters, and all machines requiring the same private parameters get the same private parameters. In addition, public and private parameters match each other. For ex-

ample, if a public key is in the public parameters and the corresponding private key is in the private parameters of some party. No machine that requires setup information carries out other operations before receiving the setup.

Definition 6 (Trusted setup, $\mathcal{F}_{\triangle}$)**.** During the first activation, $\mathcal{F}_{\triangle}$ internally generates all necessary setup parameters. If some machine $\mathcal{M}_1$ expects setup, then $\mathcal{F}_{\triangle}$ has a sender-clocked port pair with this machine $\mathcal{M}_1$. $\mathcal{F}_{\triangle}$ writes the setup parameters to the buffer for $\mathcal{M}_1$ and clocks this buffer. $\mathcal{M}_1$ stores this data and immediately clocks a default response to $\mathcal{F}_{\triangle}$. Then $\mathcal{F}_{\triangle}$ can continue the same interactions with all machines $\mathcal{M}_1, \ldots \mathcal{M}_\ell$ until all of them have received and stored the setup data. Later, $\mathcal{F}_{\triangle}$ can be queried by the adversary to learn the private parameters of the corrupted parties. The setup is said to be *consistent* if all machines expecting the same setup information get the same information.

Note that the setup could also be defined as a machine that is polled by $\mathcal{M}_i$ whenever they need setup. The setup distribution in Definition 6 means that some corrupted machine $\mathcal{M}_i$ can stop the protocol by not giving the control back to $\mathcal{F}_{\triangle}$. However, the following description for secure multiparty computation also leaves many other such places where execution can be stalled. The strength of the current description is that the setup distribution can be executed once in the beginning and it is not necessary to consider it in the rest of the protocol description.

In a real protocol, the adversary would learn the setup information from the parties that it corrupts. For the purpose of this thesis, we only consider security models where the parties do not forget their parameters and these are not updated. Hence, the same effect can be achieved if the adversary gets these parameters from the setup functionality. This is convenient for cases where the protocol description is simplified to limit the party interaction with the adversary as used in Chapter 5. Note that the setup itself is not corruptible and always functions as expected. Some security models, such as the common reference string that is maliciously chosen [18] or where the choice is at least limited by the adversary [46] would also need a corruptible setup that is not discussed here. In addition, setup may need to be repeated to adapt this formalism to consider the proactive security model.

In Chapter 5, the set of machines changes as the proofs transform the hybrid execution model into the abstract execution model. The transformation formally also changes the machine $\mathcal{F}_{\triangle}$ since it needs to interact with different machines. However, it always operates according to the rules in Definition 6. As no machine does any other steps before all others have also received their setup, the exact order in which setup is distributed is irrelevant and the changes to $\mathcal{F}_{\triangle}$ to adapt to a changing set of machines are not discussed in the proofs.

In many cases, the setup functionality $\mathcal{F}_{\triangle}$ has two modes of operation – it either gives a genuine or fake setup. The latter is used to provide trapdoors for simulation. The real setup is used when interacting with the real protocol, and the fake one is used when interacting with the ideal adversary. This change is

not explicitly considered, as this thesis focuses on the ideal functionalities and the hybrid model. Still, it should be kept in mind that if, for some protocol, the fake setup is required, then this is the mode where the $\mathcal{F}_{\triangle}$ operates.

3.3. Security of Data Storage

A common component of secure computation frameworks is some representation of the securely held data that comes with methods to both protect the inputs of the user and publish the desired outputs. The secure storage can be based on secret sharing, encryption or the encoding functions of the garbled circuit. Frameworks that combine computation methods often also combine storage methods. The following defines a secure storage domain δ based on secret sharing terminology where the private data is first shared through a functionality $\mathcal{S}_\delta$ and can be reconstructed through a functionality $\mathcal{R}_\delta$. Storage domains can also use secure setup $\mathcal{F}_{\triangle}$, in which case both $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ receive the setup parameters. The functionality $\mathcal{S}_\delta$ distributes the input to shares received by all or a subset of the parties involved in the protection domain. The reconstruction functionality $\mathcal{R}_\delta$ receives shares and outputs either the value or a special failure symbol $\perp$ if the reconstruction fails for any reason. A storage domain is correct if the reconstruction of shares given by $\mathcal{S}_\delta$ always gives the same value as the input of $\mathcal{S}_\delta$. To illustrate, for homomorphic encryption, encryption is formalised as sharing and decryption as reconstruction. The setup is used to generate the necessary keys. For garbled circuits, the correspondence is less obvious and is defined based on the correspondence of wire labels and bit values. Sharing matches the bits to labels and reconstruction matches the outcome label back to bits. Note that the garbling scheme usually only gives reconstruction information for the wires designated as outputs.

The sharing and reconstruction functionalities $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ are defined as stateless in terms of their shared or reconstructed inputs. However, they store the setup that they receive before any other activations. Statelessness is suitable for many schemes, but there can be some, where the state is necessary or one needs to carefully consider how to define the functionalities. For example, in garbled circuits, the encoding depends on the concrete wire where the value is on. Hence, the input to the encoding functionality should be the wire and the value. Similarly, the decoding should take the wire and the encoding and can only be used to decode the outputs of the garbled circuit. There are also stateful encryption schemes where, similarly to the garbled circuits, the state or some necessary part of it can be considered as part of the input and output of $\mathcal{S}_\delta$ and input of $\mathcal{R}_\delta$. Treating the information regarding the state as an extra input means that these functionalities remain stateless.

To define the security properties of the storage domain, it is essential to remember that all such properties hold for certain adversaries or sets of corrupted parties. Let $\mathcal{A}_\delta$ be the adversary structure defining which sets of parties can be corrupted without losing the security properties of the storage domain. $\mathcal{A}_\delta$ is a

set of sets of parties, and each of these sets can be safely corrupted together. The following definitions are akin to secret sharing security definitions in [22]. The hiding property in Section 3.3.1 is a fairly straightforward extension of privacy. However, the integrity property called modification awareness in Section 3.3.2 in a generalisation of recoverability that is necessary to consider malicious actions in secure multiparty computation. The correspondence is discussed more in Section 3.3.3.

The definitions of hiding and modification awareness are given as indistinguishability games defined using collections of machines in the RSIM notation since this allows to use these results directly in the context of secure multiparty computation protocols that are also defined in RSIM here. In addition, the definitions are drawn up with the knowledge that it is necessary to simulate the shares of corrupted parties within security proofs of secure multiparty computation. Often, it is necessary for the simulator first to simulate the shares blindly and then adapt the shares not yet seen by the adversary so that they could be reconstructed to the desired value. This process has been called patching in [56].

3.3.1. Hiding

Intuitively, a storage domain δ is hiding if, as long as the adversary corrupts only a set of parties allowed by $\mathcal{A}_\delta$, it learns no information about the secret input. This is captured by Definition 8. In addition, the definition considers what happens if more parties become corrupted. Furthermore, it limits other unwanted behaviours of $\mathcal{R}_\delta$ and $\mathcal{S}_\delta$ that may leak something other than the private value, for example, the setup parameters or some previous values.

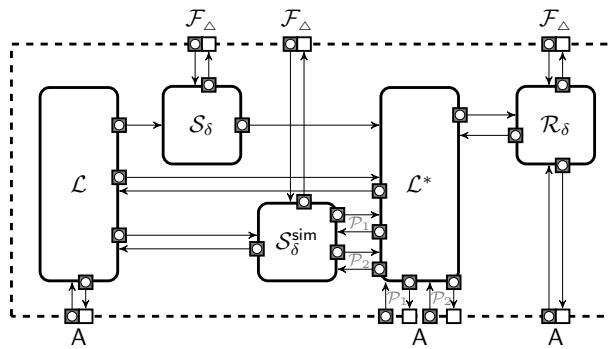


Figure 9: Configuration defining the hiding property of a storage domain with secret sharing functionality $\mathcal{S}_\delta$, share simulator $\mathcal{S}_\delta^{\mathsf{sim}}$, reconstruction functionality $\mathcal{R}_\delta$, adversary A, value memory $\mathcal{L}$, and share memory $\mathcal{L}^*$. Adversary and $\mathcal{L}^*$ ports are drawn for the two-party case. The grey labels show that duplicated buffers are intended for parties $\mathcal{P}_1$ and $\mathcal{P}_2$.

The collection in Figure 9 defines the hiding games. The functionality $\mathcal{S}_\delta$ is the secret sharing functionality, and $\mathcal{R}_\delta$ is the reconstruction functionality. A is the adversary and $\mathcal{F}_\triangle$ is the secure setup. The memory machines $\mathcal{L}$ and $\mathcal{L}^*$ store the

values and their shared representations, respectively. The ports for the adversary and $\mathcal{L}^*$ are drawn for the two-party case. Each party would be a separate machine and have its own connection to $\mathcal{S}_\delta$, $\mathcal{S}_\delta^{\mathsf{sim}}$ and $\mathcal{R}_\delta$ as well as the adversary. The machine $\mathcal{L}^*$ is an abstraction of the parties that keeps the shares of all parties in one machine. If there are more than two parties, there are also more buffer pairs for each currently duplicated communication channel. $\mathcal{L}$ stores a one-dimensional array $\mathfrak{s}$ where the adversary can write inputs it wants to have shared. $\mathcal{L}^*$ has a multidimensional array $\mathfrak{s}_*$ for the shares, $\mathfrak{s}_\circ$ for the temporary shares, and a one-dimensional array $\mathfrak{b}$ to define if the shares are generated by $\mathcal{S}_\delta$ or $\mathcal{S}_\delta^{\mathsf{sim}}$. The share simulator $\mathcal{S}_\delta^{\mathsf{sim}}$ is an efficient machine that has to generate shares to $\mathfrak{s}_*$ but can only query the respective value in $\mathfrak{s}$ once the adversary has corrupted a subset of parties not in $\mathcal{A}_\delta$ so that it breaks the hiding property. The existence of $\mathcal{S}_\delta^{\mathsf{sim}}$ needs to be shown to prove that the storage domain is hiding.

The hiding definition uses the structure considered in Figure 9 in two different manners. Game $\mathfrak{B}_0$ represents honest behaviour where all shares are generated by the real sharing functionality $\mathcal{S}_\delta$. In game $\mathfrak{B}_1$, the adversary can force share of the value in location ℓ to be generated by $\mathcal{S}_\delta^{\mathsf{sim}}$ by setting the value $\mathfrak{b}[\ell] = 1$. It is the goal of the adversary to distinguish simulated shares from the real shares.

Note that the description has separate buffers for parties but no explicit representation of the party that could be corrupted. In a protocol these buffers would be connected to machines representing a party and the adversary could only use them if a party is corrupted. For the current exposition, a party is considered corrupted in this collection if the adversary uses the buffer pair for the said party. $\mathcal{L}^*$ stores the set of corrupted parties.

Definition 7 (Hiding games, $\mathfrak{B}_0$ and $\mathfrak{B}_1$)**.** As the first step, $\mathcal{F}_\triangle$ distributes setup according to Definition 6. The adversary A can interact with the configuration in Figure 9 in the following ways. There are no interactions or steps carried out other than the ones described in the following as only A can activate the execution.

- A can send value x and location ℓ_i to $\mathcal{L}$. $\mathcal{L}$ stores $\mathfrak{s}[\ell_i] = x$. $\mathcal{L}$ sends ℓ_i to $\mathcal{L}^*$ that stores this.

  Note Each location ℓ_i can be set once.

- A can send commands to write a value b to $\mathfrak{b}[\ell] = b$ to $\mathcal{L}^*$ using the buffer pair for $\mathcal{P}_{i_k}$ before reading $\mathfrak{s}_*[\ell, i_j]$ for any $\mathcal{P}_{i_j}$.
  - Party $\mathcal{P}_{i_k}$ is marked as corrupted by $\mathcal{L}^*$.
  - If the adversary is playing in game $\mathfrak{B}_1$ then the value $\mathfrak{b}[\ell]$ is set so that $\mathfrak{b}[\ell] = b$.
- A can send commands to $\mathcal{L}^*$ to read $\mathfrak{s}_*[\ell, i_k]$ using the buffer pair for $\mathcal{P}_{i_k}$ if it has written $\mathfrak{s}[\ell]$ to $\mathcal{L}$.
  - Party $\mathcal{P}_{i_k}$ is marked as corrupted by $\mathcal{L}^*$.
  - If $\mathcal{L}^*$ has not received location ℓ from $\mathcal{L}$ then $\mathcal{L}^*$ does not do anything.
  - If the shares are not yet present in $\mathcal{L}^*$ ($\mathfrak{s}_*[\ell, i_k]$ is empty), then one of

the following happens.

  * If $\mathfrak{b}[\ell] = 0$ then $\mathcal{L}^*$ asks $\mathcal{L}$ to share the value in $\mathfrak{s}[\ell]$.
    $\mathcal{L}$ does that by sending the value to $\mathcal{S}_\delta$. $\mathcal{S}_\delta$ generates the shares and sends them to $\mathcal{L}^*$. $\mathcal{L}^*$ stores the share of party $\mathcal{P}_{i_j}$ to $\mathfrak{s}_*[\ell, i_j]$.
  * If $\mathfrak{b}[\ell] = 1$ then the query is sent to $\mathcal{S}_\delta^{\mathsf{sim}}$ using the buffer pair for $\mathcal{P}_{i_k}$. $\mathcal{L}^*$ also sends the set of indices of corrupted parties. If the adversary has corrupted more parties than allowed by $\mathcal{A}_\delta$ then $\mathcal{S}_\delta^{\mathsf{sim}}$ can query $\mathcal{L}$ to learn the value $\mathfrak{s}[\ell]$ and use it. $\mathcal{S}_\delta^{\mathsf{sim}}$ delta generates the necessary share and sends it to $\mathcal{L}^*$. $\mathcal{L}^*$ writes this to $\mathfrak{s}_*[\ell, i_k]$.

– Finally, $\mathcal{L}^*$ responds with the value $\mathfrak{s}_*[\ell, i_k]$.

Note In $\mathfrak{B}_0$, the value of $\mathfrak{b}[\ell]$ is always 0, and the real sharing functionality is always used even if the adversary tries to modify this value.

Note In $\mathfrak{B}_1$, the value of $\mathfrak{b}[\ell]$ is set by the adversary when it asks for the first share for a location ℓ. It cannot be overwritten with subsequent queries for the location ℓ.

Note $\mathcal{S}_\delta^{\mathsf{sim}}$ can use the public setup parameters and the private parameters of the corrupted parties to simulate the shares of the corrupted parties.

- A can ask $\mathcal{R}_\delta$ to reconstruct the value in location ℓ if it has stored a value in location ℓ in $\mathcal{L}$.
  – $\mathcal{R}_\delta$ requests the shares in $\mathfrak{s}_*[\ell]$ from $\mathcal{L}^*$.
    * If $\mathfrak{b}[\ell] = 0$ then $\mathcal{L}^*$ sends the shares in $\mathfrak{s}_*[\ell, i_k]$ to $\mathcal{R}_\delta$.
    * If $\mathfrak{b}[\ell] = 1$ then the shares in $\mathfrak{s}_*[\ell]$ cannot be used.
      If $\mathfrak{s}_\circ[\ell]$ is empty then then $\mathcal{L}^*$ first requests $\mathcal{L}$ to share $\mathfrak{s}[\ell]$ using their direct communication channel. $\mathcal{S}_\delta$ shares the value and $\mathcal{L}^*$ stores these shares to $\mathfrak{s}_\circ[\ell]$.
      $\mathcal{L}^*$ gives $\mathfrak{s}_\circ[\ell, i_k]$ to $\mathcal{R}_\delta$.

    Note If $\mathfrak{b}[\ell] = 1$ then the shares in $\mathfrak{s}_*[\ell]$ are simulated and not corresponding to the real value. Hence, the reconstruction must use new shares generated from the correct value.

  – $\mathcal{R}_\delta$ reconstructs the value and returns this to A.

Definition 8 (Hiding storage)**.** A storage domain δ is perfectly hiding if no adversary can distinguish configurations $\mathfrak{B}_0$ and $\mathfrak{B}_1$. A storage domain δ is hiding for a class of adversaries $\mathbb{A}$ if the advantage is negligible for any $\mathsf{A} \in \mathbb{A}$.

Many secret sharing schemes do not use private setup parameters. This means that different outputs of $\mathcal{S}_\delta$ are not correlated by any secret information and are independent because of the stateless nature of $\mathcal{S}_\delta$. In this case, the existence of $\mathcal{S}_\delta^{\mathsf{sim}}$ for a single sharing is sufficient to prove the hiding property. This is in line with the usual security definitions for secret sharing.

If secret parameters are used, it is important to consider the case where many sharings are generated. In addition, adversary A is also expected to know the setup

parameters of the corrupted parties and any other information that $\mathcal{S}_\delta^{\mathsf{sim}}$ can have from the setup functionality. Suppose the shares leak more information about the setup or previously shared values than is already known by the corrupted parties. In that case, it is not likely that the hiding property can be satisfied.

If the storage domain considers static corruption, then $\mathcal{S}_\delta^{\mathsf{sim}}$ can also know the set of corrupted parties beforehand and use this to generate shares for the corrupted parties using the setup parameters of the corrupted party. In case of adaptive corruption, the simulator $\mathcal{S}_\delta$ must carry out a procedure called patching in [56] to continuously create new shares as more parties become corrupted and make sure the shares correspond to the correct value. Dealing with mobile corruption requires updating the definition to consider the stages where the sets of corrupted parties can change and needs to add steps to suitably update the shares.

Note that this definition is in line with the perfect privacy of secret sharing schemes. Hence, it does not consider the case where some partial information about the secret may leak if enough parties become corrupted. On the other hand, this definition does not limit any leakage once too many parties are corrupted. A secure computation framework often contains several storage domains with different adversary structures. For example, there are public values known to all parties, values that are shared and values that are known to some fixed party. As soon as the adversary corrupts $\mathcal{P}_i$, it learns that party's public and local values, as it has corrupted more parties than allowed by the adversary structure of these storage domains. On the other hand, it does not get access to shared values unless $\mathcal{P}_i$ was the last party to be corrupted to go over the adversary structure of the shared values.

3.3.2. Modification Awareness and Limited Control

Modification awareness is an extension of the integrity property of secret sharing. In robust secret sharing, the situation is black and white. Either the share can contain the desired value, or the reconstruction fails. However, in some other schemes, it may be possible to also change the underlying value, at least to some extent. Modification awareness covers both of these properties and defines an extractor $\mathcal{E}_\delta$ that has to take the adversarial modifications and understand if they result in a change in the shared value, corrupt the value, or simply change the shares and keep the same value. Modification awareness is necessary because an active adversary may always change its shares. However, it has been observed that, in many cases, the adversary also knows how its changes affect the shared value. For example, [70] discusses that for typical schemes, adversarial actions result in additive changes to the final output.

A storage domain δ defines a modification function $\oplus_\delta$. Let $[\![x]\!] = (x_1, \ldots, x_n)$ be the secure data representation given by $\mathcal{S}_\delta$ so that $\mathcal{P}_i$ knows x_i. Let A be the set of indices of corrupted parties. Let $[\![\hat{x}]\!] = (\hat{x}_1, \ldots, \hat{x}_n)$ be the adversarial modification of this where $x_i = \hat{x}_i$ if $\mathcal{P}_i$ is not corrupted. The extractor $\mathcal{E}_\delta$ gets the

values $(x_i)_{i\in A}, (\hat{x}_i)_{i\in A}$ for the set of corrupted parties in A and it also gets the setup parameters known by these parties. The goal of $\mathcal{E}_\delta$ is to output a modification Δ to the value that is such that $\mathcal{R}_\delta([\![x]\!]) \oplus_\delta \Delta = \mathcal{R}_\delta([\![\hat{x}]\!])$. If the modification invalidates the secure representation of data and reconstruction is not supposed to work, then the modifier $\Delta = \perp$. The modification function can be any function with the additional constraint that $a \oplus_\delta \perp = \perp$ and $\perp \oplus_\delta a = \perp$ for any a that is a valid input to $\mathcal{S}_\delta$.

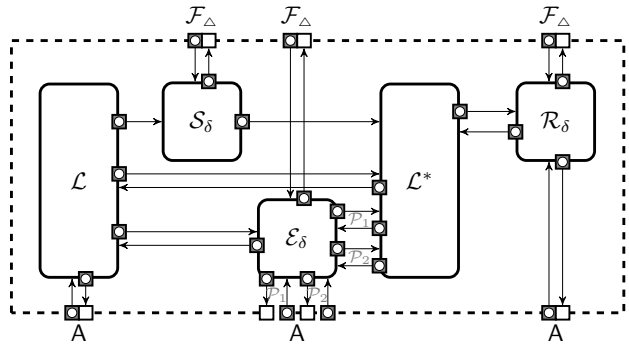


Figure 10: Configuration defining the modification awareness and limited control properties of a storage domain with secret sharing functionality $\mathcal{S}_\delta$, extractor $\mathcal{E}_\delta$, reconstruction functionality $\mathcal{R}_\delta$, adversary A, value memory $\mathcal{L}$, and share memory $\mathcal{L}^*$. Adversary and $\mathcal{L}^*$ ports are drawn for the two-party case and the parties are noted as the buffer labels.

To prove modification awareness, the prover must show that there is an efficient extractor $\mathcal{E}_\delta$ for the configuration shown in Figure 10 and the game defined in Definition 9. As before, $\mathcal{R}_\delta$ and $\mathcal{S}_\delta$ are the reconstruction and sharing functionalities. The memory machines $\mathcal{L}$ and $\mathcal{L}^*$ are slightly modified from the hiding property description. In particular, $\mathfrak{s}_*$ is the entire state of $\mathcal{L}^*$ and $\mathcal{L}^*$ allows A to write $\mathfrak{s}_*[\ell, i_k]$. Similarly, as before, a party $\mathcal{P}_{i_k}$ is considered corrupted if the adversary reads or writes $\mathfrak{s}_*[\ell, i_k]$.

Definition 9 (Modification awareness game, $\mathfrak{B}_2$)**.** The games start by $\mathcal{F}_\triangle$ distributing the setup according to Definition 6. The adversary A can interact with the modification awareness game in Figure 10 and cause the following actions. There are no other execution steps.

- A writes values to $\mathfrak{s}[\ell]$ by sending the value x and location ℓ to $\mathcal{L}$.
  - $\mathcal{L}$ sets $\mathfrak{s}[\ell] = x$.

  Note Each value can be set once.
- A reads values in $\mathcal{L}^*$ through $\mathcal{E}_\delta$ by requesting a location (ℓ, i_k) if it has set location ℓ in $\mathcal{L}$.
  - Party $\mathcal{P}_{i_k}$ is marked as corrupted by $\mathcal{L}^*$ meaning that it adds i_k to A.
  - If adversary reads an uninitialized $\mathfrak{s}_*[\ell, i_k]$ then $\mathcal{L}^*$ notifies $\mathcal{L}$ that shares the value in $\mathfrak{s}[\ell]$ using $\mathcal{S}_\delta$.

  - $\mathcal{L}^*$ sends $\mathfrak{s}_*[\ell, i_k]$ to $\mathcal{E}_\delta$ that sends it to A.
- A can modify each location ℓ by sending $\mathcal{E}_\delta$ the new values $\hat{x}_{i_k}$ for locations $\mathfrak{s}_*[\ell, i_k]$ for $i_k \in A$.
  - Each party $\mathcal{P}_{i_k}$ is considered corrupted, each i_k is added to A.
  - $\mathcal{E}_\delta$ uses the existing shares $\mathfrak{s}_*[\ell, i_k]$ for $i_k \in A$ in $\mathcal{L}^*$ and the modified values $\hat{x}_{i_k}$ received from A to compute Δ.
  - $\mathcal{E}_\delta$ updates the respective locations in $\mathcal{L}^*$ with the values $\hat{x}_{i_k}$ received from the adversary and sends the pair (ℓ, Δ) to $\mathcal{L}$.
  - $\mathcal{L}$ sets $\mathfrak{s}[\ell] = \mathfrak{s}[\ell] \oplus_\delta \Delta$.

  Note Each location ℓ can be modified once.
- The adversary can order $\mathcal{R}_\delta$ to reconstruct the value in $\mathfrak{s}_*[\ell]$.
  - $\mathcal{R}_\delta$ forwards this request to $\mathcal{L}^*$ that responds with $\mathfrak{s}_*[\ell]$ that is used as a reconstruction input by $\mathcal{R}_\delta$.
  - The reconstructed value is given to A.

  Note The adversary wins the game if the location ℓ is initialized, the received value is different from $\mathfrak{s}[\ell]$, and the set of corrupted parties is in $\mathcal{A}_\delta$.

Definition 10 (Modification aware storage domain)**.** The adversary A wins the modification game $\mathfrak{B}_2$ (Definition 9) if the set of corrupted parties is in $\mathcal{A}_\delta$, location ℓ is initialized, the adversary orders $\mathcal{R}_\delta$ to reconstruct the shared value in $\mathfrak{s}_*[\ell]$ and the reconstructed value is different from the value in $\mathfrak{s}[\ell]$ in $\mathcal{L}$. A storage domain δ is modification aware against a class of adversaries $\mathbb{A}$ if the probability of winning $\mathfrak{B}_2$ is negligible for any $\mathsf{A} \in \mathbb{A}$.

In practice, the adversary may change the shares many times. However, the game definition limits the changes to one modification that affects all corrupted shares simultaneously. The same effect could be achieved if $\mathcal{E}_\delta$ stores the initial valid shares and all the modifications and the difference Δ is computed only when the reconstruction order is received. The current definition is more straightforward since $\mathcal{E}_\delta$ is stateless.

For a stateless $\mathcal{E}_\delta$, continuously computing the modifier and discarding the previous state might not be possible as A may keep track of all of the state. For example, A could first invalidate the shares and then restore the shares to their previous value. The invalidation causes a situation where $\mathfrak{s}[\ell] = \perp$ and, by definition of $\oplus_\delta$, no later modification Δ could restore the initial value in $\mathfrak{s}[\ell]$. In addition, the current definition simplifies $\mathcal{E}_\delta$ since it can always assume that the initial shares are valid shares of some value.

Similarly to the definition of the hiding property, it suffices to show extraction for a single shared value if no private setup parameters are involved. For cases with private parameters, these parameters and a possible connection between different shares must be taken into account.

Note that modification awareness covers basic integrity for cases where the only possible outcome of the modifier Δ is either $\perp$ or the identity element for $\oplus_\delta$. This definition also covers correctness, meaning that the new output of $\mathcal{S}_\delta$ is always reconstructed to the same value as the input. If the storage domain is not correct, then the adversary could win in the modification awareness game even if it does not do any modifications.

If the secret sharing scheme is robust, then the only possible output for $\mathcal{E}_\delta$ is the identity element of $\oplus_\delta$. If the scheme is verifiable, then the possible outputs are the identity element and $\perp$. However, note that the adversary may still change the shares in $\mathfrak{s}_*$ even if the output is the identity element for the change in $\mathfrak{s}$. For example, an adversary might switch shares between parties. As before, the modification awareness definition does not give any guarantees when the adversary corrupts more parties than allowed by $\mathcal{A}_\delta$ and, in this case, any modification might be valid. For example, if an adversary corrupts a party, it can modify its local values. If it corrupts all parties, it learns all setup parameters and can easily create valid shares for any value.

Modification awareness considers finding a value modification based on the modification of the shares. Sometimes, it is also necessary to consider if the reverse is possible. For example, can the adversary or a reverse extractor find valid share modifications to change the value by Δ. This is defined as the adversary having limited control over the storage domain since it is natural to expect that, for each specific domain δ, the acceptable set of values Δ is limited. If the extractor $\mathcal{E}_\delta$ is a two-way extractor, then limited control and modification awareness can be considered in the same configuration in Figure 10.

The limited control game in Definition 11 follows the pattern of $\mathfrak{B}_2$ (Definition 9) with the difference that the adversary can send a modifier Δ for $\mathcal{L}$ through the extended extractor $\mathcal{E}_\delta$. $\mathcal{L}$ still sets $\mathfrak{s}[\ell] = \mathfrak{s}[\ell] \oplus_\delta \Delta$. However, the extended extractor $\mathcal{E}_\delta$ uses $(x_i)_{i\in A}$ from $\mathfrak{s}_*[\ell, i]$ and computes new shares $(\hat{x}_i)_{i\in A}$ that result in modification Δ.

Definition 11 (Limited control game, $\mathfrak{B}_3$)**.** The game starts with the setup distribution as in Definition 6. The adversary A can interact with the limited control game in Figure 10 and cause the following actions. There are no other executions.

- A writes values to $\mathfrak{s}[\ell]$ by sending the value x and location ℓ to $\mathcal{L}$.
  - $\mathcal{L}$ sets $\mathfrak{s}[\ell] = x$.

  Note Each value can be set once.
- A reads values with location (ℓ, i_k) in $\mathcal{L}^*$ through $\mathcal{E}_\delta$.
  - Party $\mathcal{P}_{i_k}$ is considered to be corrupted meaning that i_k is added to A.
  - If adversary reads an uninitialized $\mathfrak{s}_*[\ell, i_k]$ then $\mathcal{L}^*$ notifies $\mathcal{L}$ that shares the value in $\mathfrak{s}[\ell]$ using $\mathcal{S}_\delta$.
- A can modify each location ℓ by sending $\mathcal{E}_\delta$ the modifier Δ for location $\mathfrak{s}[\ell]$.
  - $\mathcal{E}_\delta$ uses the existing shares $\mathfrak{s}_*[\ell, i_k]$ for $i_k \in A$ in $\mathcal{L}^*$ and the modifier Δ

received from A to compute new shares $\hat{x}_{i_k}$.

- $\mathcal{E}_\delta$ updates the respective locations in $\mathcal{L}^*$ with the values $\hat{x}_{i_k}$ received from the adversary and sends the pair (ℓ, Δ) to $\mathcal{L}$.
- $\mathcal{L}$ sets $\mathfrak{s}[\ell] = \mathfrak{s}[\ell] \oplus_\delta \Delta$.

Note Each location ℓ can be modified once.

- The adversary can order $\mathcal{R}_\delta$ to reconstruct the value in $\mathfrak{s}_*[\ell]$.
  - $\mathcal{R}_\delta$ forwards this request to $\mathcal{L}^*$ that responds with $\mathfrak{s}_*[\ell]$ that is used as a reconstruction input by $\mathcal{R}_\delta$.
  - The reconstructed value is given to A.

  Note The adversary wins the game if the extractor fails, meaning that the outcome of $\mathcal{R}_\delta$ for location ℓ differs from $\mathfrak{s}[\ell]$ and the set of corrupted parties A is in $\mathcal{A}_\delta$.

Definition 12 (Limited control)**.** The adversary A wins the game $\mathfrak{B}_3$ (Definition 11) if the extractor fails, meaning that the outcome of $\mathcal{R}_\delta$ for location ℓ queried by the adversary differs from $\mathfrak{s}[\ell]$ and the set of corrupted parties A is in $\mathcal{A}_\delta$. A class of adversaries $\mathbb{A}$ has limited control over a storage domain δ if the probability that the adversary wins $\mathfrak{B}_3$ is negligible for any adversary $\mathsf{A} \in \mathbb{A}$.

Note that a storage domain can only have limited control if there is a straightforward way for the extractor $\mathcal{E}_\delta$ to define shares of $\perp$. Current versions of both modification awareness and limited control are studied for cases where there is a way for $\mathcal{S}_\delta$ also to define shares of $\perp$ so that $\perp$ is a valid input that the adversary can write to $\mathcal{L}$ if there is any way for the shares to become invalid. If invalid shares are possible, then $\mathcal{E}_\delta$ may write $\perp$ to $\mathcal{L}$ anyway. In such case, the reconstruction functionality $\mathcal{R}_\delta$ has to be able to output $\perp$. It is not required in these definitions for $\mathcal{S}_\delta$ to accept $\perp$ as an input. However, in most cases, the storage needs to be both hiding and modification aware and in these cases, it is most straightforward if $\perp$ is a valid input to $\mathcal{S}_\delta$.

3.3.3. Connections to Previous Definitions

The hiding property corresponds well with the semantic security (CPA security) of encryption schemes and the secret sharing privacy defined in Chapter 2. The main difference from the secret sharing privacy definition is that the definition of hiding explicitly considers the case where many values are shared. A similar effect of hiding many values is achieved in the definition of semantic security with public key encryption, as the adversary can encrypt any value as it knows the public key. The definition of the hiding property looks quite different from the semantic security but aims to achieve a similar effect without giving the adversary the setup information that is the generalisation of the public key in the semantic security case. The hiding definition either gives an encryption (or share or otherwise protected version) of a known value or simulates that value. In the semantic security game in Section 2.1.6, the adversary has to distinguish two val-

ues. For an encryption scheme, a natural simulation usually gives an encryption of some other value. Such simulation implies that the adversary should not be able to distinguish ciphertexts derived from two different values as in the definition in Section 2.1.6. Hence, if an encryption scheme has semantic security then there should also exist a suitable simulator $\mathcal{S}_{\delta}^{\mathsf{sim}}$ that proves that the encryption scheme is hiding as in Definition 8. There is no clear correspondence between the given definitions and chosen ciphertext attacks against encryption schemes. In the given definitions, the sharing functionality is part of the game. There is no straightforward way in the hiding game for the adversary to generate sharings and ask for their reconstruction.

The definitions of modification awareness and limited control are more complicated. Modification awareness extends the recoverability property of secret sharing schemes. For robust schemes, the definitions are essentially the same, with the exception that modification awareness also explicitly considers sharing multiple values. For homomorphic encryption, there is no explicit definition of a similar property. If the key and ciphertext are known, the ciphertext can always be modified. However, at least simple modifications, such as adding a known value to the encrypted value, are covered by the modification awareness definition. There could be modifications where the modification function is not easy to define. However, in reasonable use cases, either the adversary is passive and does not do modifications, or the authenticity of encryption or ciphertext contents should be somehow verified. Such verification would then limit the allowed modifications.

Connecting the definitions with those of the security of garbled circuits is more complicated as these definitions are not directly about the secure representation. The obliviousness property of garbled circuits implies that the output of the garbled circuit has to be hiding, as the simulator does not learn the expected output of the circuit. In addition, the privacy and obliviousness properties imply that other wire encodings have to be at least somewhat hiding if the whole circuit information is not given to the simulator. In both games, the adversary only learns the wire encodings corresponding to its inputs. However, if the intermediate wire labels would leak the actual value on the wire, then it would be easy to distinguish the simulation and the real garbling for the cases where the simulation generates a wrong value to some wire due to using a wrong gate. Hence, at least the values must be hidden. There is no equivalent of modification awareness in the pure case of garbled circuits since the definitions expect passive adversaries that do not try to do any modifications.

3.4. Ideal Functionalities

The ideal functionality is essentially the protocol specification and, overall, does not have to obey any explicit rules. It is a definition rather than anything concrete and could, in fact, define any desired properties and functionalities. However,

there is also a common intuition about the shape of the secure multiparty computation protocol and what it means for it to be secure. This section defines ideal functionalities based on this common understanding, and, as such, these could be considered the standard lightweight ideal functionalities.

Overall, there are two main variations on how to define an ideal functionality for an MPC protocol. In the first, the ideal functionality considers the whole computation and its inputs and outputs are plain values that are either public or belong to some party. In the second, the ideal functionality corresponds to some specific operations done during the computations and the whole computation is a composition of many functionalities. We call the latter *lightweight* ideal functionalities as they often correspond to lightweight operations such as basic arithmetic as opposes to more complex programs. The arithmetic black-box (ABB) defined in Section 2.1.8 fits into the first category as it defines the whole functionality from public inputs to public outputs. However, individual commands inside the ABB can be easily mapped as lightweight ideal functionalities. The second is often used in research papers considering passive security and designing new algorithms assuming some existing primitives. However, it is more complicated for cases where the representation of the secret information contains some private setup and the inputs or outputs may be corrupted. In this thesis, the ideal functionality $\mathcal{F}$ is considered to be the lightweight functionality. The overall secure computation consisting of these lightweight functionalities is called the protection domain (specified in Section 3.6). Comparison of the protection domain approach and ABB appears in Section 3.7.

Each lightweight ideal functionality follows a common work pattern. It receives all inputs, reconstructs the encoded values, computes the desired functionality with the plain values, shares the output, and sends each participant their part of the output. Note that public inputs and outputs are a special case where the reconstruction or sharing is an identity function. By definition, such an ideal functionality produces a fresh output encoding that depends only on the value of the output and not on the input encodings. This property is similar to the circuit hiding property of homomorphic encryption introduced in Section 2.1.6 and ensures that the outputs do not reveal information about the computation. In addition, for functionalities that output many values, the sharings of these values are not correlated. This structure means, for example, that an output of a functionality cannot contain all input shares or the output of some deterministic computation with these shares. Also, no protocol input can be copied to the output of the functionality.

An ideal functionality may be queried many times throughout the execution of the protocol. In such cases, it is important to distinguish protocol instances. Protocol instance tags t are used to communicate which instance an input or an output belongs to. We assume that the functionalities are used within one protection domain (described in Section 3.6 meaning that it is in one secure computation protocol and uses the same setup for all its instances. In principle, this can be extended to allow the setup to be instance specific but this is not considered here.

Aforementioned intuition about the ideal functionality is sufficient in many cases throughout this document. The following considers the detailed construction of the functionality and the different interactions that it can have with the adversary.

3.4.1. Canonical Ideal Functionalities

The structure of a lightweight ideal functionality $\mathcal{F}_p$ is depicted in Figure 11. This section specifies some more properties of these components that together form a canonical ideal functionality. The machines $\mathcal{R}$ and $\mathcal{S}$ combine the $\mathcal{R}_\delta$ and $\mathcal{S}_\delta$ functionalities of each storage domain used by this ideal functionality. Note that this includes domains for public values. The machines $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$ manage the timing of the execution of $\mathcal{R}$ and $\mathcal{S}$, respectively. The functionality $\mathcal{F}_p^\circ$ computes the desired function using the values.

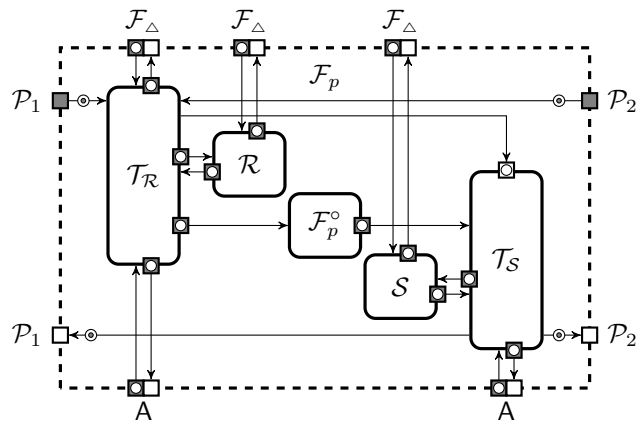


Figure 11: Structure of the ideal functionality $\mathcal{F}_p$ with connections for two parties and setup $\mathcal{F}_\triangle$. The ideal functionality contains a reconstruction timer $\mathcal{T}_\mathcal{R}$, reconstruction machine $\mathcal{R}$, the computation functionality $\mathcal{F}_p^\circ$, sharing timer $\mathcal{T}_\mathcal{S}$ and sharing functionality $\mathcal{S}$.

The machine $\mathcal{T}_\mathcal{R}$ collects all inputs from parties $\mathcal{P}_i$. When receiving an input, it always stores the input and checks if it has sufficient inputs to run $\mathcal{R}$. If it has all inputs, then $\mathcal{R}$ is activated. If there are not sufficient inputs, then $\mathcal{T}_\mathcal{R}$ stops. $\mathcal{R}$ always reconstructs the value using the necessary $\mathcal{R}_\delta$ and returns the result to $\mathcal{T}_\mathcal{R}$ immediately. Upon receiving the value, $\mathcal{T}_\mathcal{R}$ writes the information about the protocol instance to the buffer to $\mathcal{T}_\mathcal{S}$ if necessary and writes and clocks the received value to $\mathcal{F}_p^\circ$. $\mathcal{F}_p^\circ$ computes the desired functionality and gives the result to $\mathcal{T}_\mathcal{S}$. $\mathcal{F}_p^\circ$ also specifies the storage domain δ for each output. $\mathcal{T}_\mathcal{S}$ forwards the storage domain and value to $\mathcal{S}$ that uses a suitable $\mathcal{S}_\delta$ to produce the shared representation of the value. $\mathcal{T}_\mathcal{S}$ then clocks the input channel from $\mathcal{T}_\mathcal{R}$ to get the instance tag and uses the tag to write the outputs to the buffers for relevant parties $\mathcal{P}_i$. Adversary controls the timing of the output buffers.

Often, the ideal functionalities $\mathcal{F}_p$ are simple one-round machines that collect inputs and compute the output of the desired function. For example, this is the case for all ideal functionalities implementing arithmetic functions. However, in general, the ideal functionality may have many rounds of communication with the parties and then $\mathcal{F}_p^\circ$ may also be stateful. In addition, a functionality can be executed many times with different instances. Hence, an instance tag is necessary to keep track of the execution. Instance tags t are included by the parties $\mathcal{P}_i$ when they send their inputs, used by $\mathcal{T}_\mathcal{R}$ to group the right shares and then used by $\mathcal{T}_\mathcal{S}$ when the outputs are sent out so that the result parties know which instance was computed. The instance tag t is communicated from $\mathcal{T}_\mathcal{R}$ to $\mathcal{T}_\mathcal{S}$ through their direct buffer. The tags are necessary since the timing of the execution is under the control of the adversary that clocks the inputs and outputs of $\mathcal{F}_p$, and the execution timing depends on when all parties have sent their inputs. Hence, the timing cannot be used to distinguish protocol instances from each other.

The concrete corruption model for the ideal functionality depends on the allowed interactions with A. However, a reasonable common restriction is that corruption must only directly affect the inputs and outputs of the corrupted parties. This is considered in Definition 13. Further details of possible corruption are discussed in Section 3.4.2.

Definition 13 (Standard corruption mode)**.** An ideal functionality $\mathcal{F}_p$ has a standard corruption mode if

1. all output values given to $\mathcal{P}_i$ are generated by $\mathcal{S}$ from the output of $\mathcal{F}_p^\circ$ and distributed by $\mathcal{T}_\mathcal{S}$
2. and the adversary cannot learn anything about the shares of the honest parties other than revealed by the outputs to the corrupted parties.

Definition 14 (Canonical ideal functionality)**.** Functionality $\mathcal{F}_p$ is in canonical form if

1. it is a collection of $\mathcal{T}_\mathcal{R}$, $\mathcal{R}$, $\mathcal{F}_p^\circ$, $\mathcal{T}_\mathcal{S}$ and $\mathcal{S}$ with the internal structure specified previously and illustrated in Figure 11,
2. has a standard corruption mode (Definition 13),
3. always outputs $\perp$ if any input reconstructed by $\mathcal{R}$ is $\perp$,
4. and only gives outputs to parties that have submitted any input.

Note that the condition that the adversary cannot learn anything other than the public outputs about the honest shares is there to ensure some common sense in the functionality while not restricting adversarial communication too much. In usual cases the adversary either does not learn anything or can lean the inputs or outputs of the corrupted parties from the functionality and these are allowed by the standard corruption mode. However, in some cases the functionality might reveal more information, such as some leaked conditions. In these cases, extra care is needed to verity that such a functionality is indeed with standard corruption or otherwise extra analysis of the protocol is needed to ensure that this functionality is a suitable candidate. The abstract model for protocol analysis derived in

Chapter 5 only considers canonical ideal functionalities.

Note that it is possible to define ideal functionalities that go from one storage domain to another and also give outputs to parties that are not giving input data. However, the definition of canonical ideal functionality expects that these parties have sent some message as input. This could be some constant message in their local protection domain and could be ignored by $\mathcal{F}_p^\circ$, but it is necessary because it ensures that the party is not surprised by messages sent from $\mathcal{F}_p$. Any ideal functionality that does not have this property could be modified to the one that has as long as the local domains can be considered. For example, an ideal functionality for sharing an input would collect a public input value from one party and a token from other parties. Then the public value would be passed through $\mathcal{R}$ and returned as the same value, $\mathcal{F}_p^\circ$ would be the identity function and finally, $\mathcal{S}$ would execute the functionality $\mathcal{S}_\delta$ for the storage domain δ that is required for the output.

The machines $\mathcal{S}$ and $\mathcal{R}$ may require the setup parameters to execute their operations, including both the public and private parameters of all parties. $\mathcal{T}_\mathcal{R}$ only requires public setup parameters, for example, to know the number of parties in the computation or the modulus of the computation. If any such information is necessary for $\mathcal{F}_p^\circ$, $\mathcal{T}_\mathcal{R}$ includes it when sending the inputs to $\mathcal{F}_p^\circ$.

The assumption is that if any input reconstruction fails and $\mathcal{F}_p^\circ$ gets $\perp$ as an input, it always outputs $\perp$. This means that $\mathcal{S}$ must have a defined way to generate shares for $\perp$ if the ideal functionality fails silently. For example, if the secret value is authenticated like in the SPDZ [59], then no single party notices that the output is actually undefined. If the failure is checked, then the failure notification can be sent publicly.

3.4.2. Corruption Modes

The previous section defined standard corruption mode in Definition 13 as the case where the outputs of the functionality do not reveal direct information about the input shares and the adversary can see the corrupted shares. There are still various ways to interact with the adversary that fall within this limitation. Note that for canonical ideal functionalities, only $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{S}$ could communicate with the adversary. Depending on the adversarial model, the adversary could communicate with either of those or both.

In the case of passive corruption, there are two possible ways to define the ideal functionality and adversary communication. First, it is possible to consider the case where the ideal functionality reveals the corrupted input shares to the adversary through $\mathcal{T}_\mathcal{R}$ and outputs through $\mathcal{T}_\mathcal{S}$. Secondly, it is possible that the ideal functionality has no connection to the adversary. This depends on whether it is more convenient to assume that the adversary sees the inputs in some other manner before they are given to the protocol. For example, the adversary may interact with the corrupted party that gives all its view to the adversary. However, sometimes it may be more convenient not to consider the parties at all and as-

sume that the ideal functionality knows which parties are corrupted and gives the corrupted inputs and outputs to the adversary itself. This thesis uses the variation with direct communication in Chapter 4 and the model with the communication with the parties in Chapter 5.

The same two patterns with regards to the inputs can be considered for the active adversary. However, an active adversary might also want to change the corrupted inputs and could therefore send updates to $\mathcal{T}_{\mathcal{R}}$. In the definition of a protection domain in Section 3.6, the assumption is that all modifications of the input are done before the inputs arrive at the ideal functionality. Throughout this thesis, the assumption is that the active adversary communicates with the parties to modify the inputs.

In most cases, the output behaviour of ideal functionalities with active adversary requires some communication between $\mathcal{T}_{\mathcal{S}}$ and the adversary. The following explains how some common definitions can be mapped to the canonical ideal functionalities. If the ideal functionality is robust, meaning that the adversary cannot affect its execution, then it has no communication with the adversary. However, if the functionality has security with abort, then the adversary can instruct $\mathcal{T}_{\mathcal{S}}$ to send abort notifications to all parties instead of the shares. If the computation is fair, then the adversary can abort without seeing the outputs of corrupted parties. Hence, $\mathcal{T}_{\mathcal{S}}$ does not give it access to shares. In case of unfair protocols, the adversary can learn the corrupted shares from $\mathcal{T}_{\mathcal{S}}$. Similarly to the failure in the input reconstruction, the abort can be either silent or announced to all parties. In the former case, the $\mathcal{T}_{\mathcal{S}}$ must generate new shares of $\perp$ that stand for the silently failed protocol. If the protocol has a selective failure possibility, then the adversary could choose this as an option, and the functionality would fail if the failure conditions were met. In the covert security model, there is some possibility that the adversary can successfully cheat. In this case, the adversary can signal the ideal functionality that it wants to cheat and the functionality has to decide if it succeeds. If cheating fails, then the ideal functionality sends the abort message to all parties and also indicates the corrupted parties or the cheating party, as in this case, the honest parties detect cheating. Suppose cheating succeeds, then in the worst case, the functionality breaks and leaks its data to the adversary similarly to the case of corrupting more parties than allowed for the hiding property. It also allows the adversary to define the outputs of the honest parties. However, it is also possible to limit the effect the adversary can have in case of successful cheating and only allow these actions and data leaks. Note that addressing the case of corruptible ideal functionalities where the adversary can choose the outputs of the corrupted party means that the functionality that shares the output should be enhanced to allow for generating valid shares for the honest parties based on the value and the corrupted shares.

Each ideal functionality defines how many corrupted parties are allowed before its security breaks. In more general, ideal functionalities define their own adversary structures. These always depend on the structures of the used storage

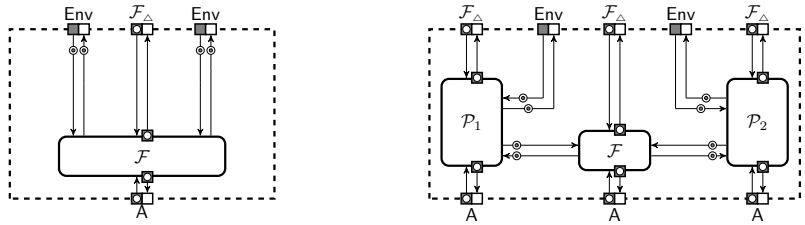
domains but can be more restricting. In case the adversary corrupts more parties than allowed by the functionality, $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{S}}$ can be used to learn the input and output values of the protocol. $\mathcal{T}_{\mathcal{R}}$ notifies the $\mathcal{T}_{\mathcal{S}}$ in case the corruption bounds have been broken since $\mathcal{T}_{\mathcal{S}}$ does not have the setup information and may not be able to decide it without the setup. If the ideal functionality does not have direct communication channels with the adversary, then the fact that it is broken can mean that $\mathcal{T}_{\mathcal{S}}$ sends all outputs to all parties. On the other hand, if the adversarial structure is the same as the hiding property of the storage domain, then simply giving all outputs as usual means that the corrupted parties can restore the input and output values from their interactions with the functionality. Note that the latter is the only formalisation where the ideal functionality does not need to know that the corruption bound has been reached.

If the ideal functionality has communication with the adversary, then the previous description assumes that the ideal functionality has knowledge of which parties are corrupted. In the case of static corruption, this can be considered as part of the setup or each party can start their interaction by declaring themselves as corrupted. In adaptive corruption, the functionality is notified when new parties become corrupted. Overall, a party is definitely corrupted if the adversary requests its information from $\mathcal{T}_{\mathcal{R}}$ or $\mathcal{T}_{\mathcal{S}}$. However, in such cases, different functionalities may consider different parties to be corrupted, and usually, this would overcomplicate keeping track of the corrupted parties. In the following, the discussion about the adversary in Section 3.6.4 resolves this by defining a coherent adversary that always corrupts all instances of a party at once. In this case, it is best to assume that the adversary would also send the corruption messages to all ideal functionalities that the party participates in. Adapting this ideal functionality definition to the mobile adversary would need to define the behaviour of the functionality in case the sets of corrupted parties reduces. However, proactive security is not explored in this thesis.

Note that the detailed structure of the ideal functionality also forces a restriction on the allowed adversary. Firstly, a party is considered corrupted when its information is asked from either $\mathcal{T}_{\mathcal{S}}$ or $\mathcal{T}_{\mathcal{R}}$. Secondly, the adversary is expected to know the inner structure or the ideal functionality. It is easy to also hide the structure with an additional multiplexing machine that mediates the adversary and the functionality communication to hide the $\mathcal{T}_{\mathcal{S}}$ and $\mathcal{T}_{\mathcal{R}}$. Hence, depending on the concrete level of detail that is needed for the discussion, the ideal functionality is depicted as having either one or two buffer pairs with the adversary.

3.4.3. Two Ways to Use an Ideal Functionality

A real protocol is carried out by parties that are sending messages from one to another. The ideal functionality either replaces the parties and all their interactions, or it can be seen as parties executing one call to the ideal functionality instead of communicating with each other. These two possibilities are illustrated in Fig-



(a) Direct use of $\mathcal{F}$, collection $\mathfrak{D}$. (b) Using $\mathcal{F}$ through parties $\mathcal{P}_i$, collection $\mathfrak{I}$.

Figure 12: Two alternatives for using a two-party ideal functionality $\mathcal{F}$.

ure 12 for the case of a two-party protocol. The parties $\mathcal{P}_i$ are simple machines that receive their inputs and use them as inputs to $\mathcal{F}$. The results that they receive from $\mathcal{F}$ are given back to Env. The ideal functionalities $\mathcal{F}$ are executed in a usual security context where the inputs come from an environment Env and protocol execution may be affected by an adversary A. In addition, parties $\mathcal{P}_i$ and the functionality $\mathcal{F}$ may receive some setup information from $\mathcal{F}_\triangle$. There are leaky buffers to and from the environment. In the second case, there are also leaky buffers between the parties and the functionality. These buffers are clocked by the adversary and may leak some metadata about the communication to the adversary.

These two ways are mostly equivalent but may suit different settings. In this work, Chapter 4 focuses more on the direct use in collection $\mathfrak{D}$ in Figure 12a and Chapter 5 on the setup with explicit parties as collection $\mathfrak{I}$ in Figure 12b. Chapter 4 focuses on merging protocols that correspond to some ideal functionalities to new protocols corresponding to some new ideal functionality and the role of the parties is not necessary for this discussion. On the other hand, Chapter 5 considers the ways how the parties use the ideal functionalities to execute different algorithms. Hence, it is straightforward to consider the parties and their code and the several ideal functionalities that may be used by the parties explicitly. In fact, Chapter 5 also proves one case where these two collections $\mathfrak{D}$ and collection $\mathfrak{I}$ are equivalent (Lemma 25).

3.5. Local Functionalities

The definition of ideal functionalities that always requires freshly shared outputs ignores an important class of computations that can take place and are commonly considered secure. These are functionalities where each party computes something on their shares without adding fresh randomness. For example, for linear secret sharing schemes, all linear combinations can be computed in this manner. These operations are local in the sense that no network communication is necessary to compute them. A storage domain usually defines which local computations also have a meaningful interpretation on the shared values.

Definition 15 (Meaningful local operation)**.** A local operation $\mathcal{G}_q$ implements a

function g_q if for any input $x_1,\ldots,x_s$ we have $(y_1,\ldots,y_t) = g_q(x_1,\ldots,x_s)$ where $x_1,\ldots,x_s$ are the values reconstructed from the input shares of $\mathcal{G}_q$ and $y_1,\ldots,y_t$ are the values reconstructed form the output shares of $\mathcal{G}_q$. A local functionality implementing some function is said to be meaningful and its meaning is g_q.

In practice, each participant carries out some part of the local functionality $\mathcal{G}_q$. Hence, $\mathcal{G}_q$ is a collection of machines $\{\mathcal{G}_{q,j}\}_{j\in\mathcal{J}_q}$ where $\mathcal{G}_{q,j}$ is the actual functionality used by party $\mathcal{P}_j$ and $\mathcal{J}_q$ is the set of parties that need to execute this functionality in order to get the meaningful result. In addition, the components $\mathcal{G}_{q,j}$ may differ for different parties, and the results of the local functionality become meaningful when all relevant parties in $\mathcal{J}_q$ have executed their component. $\mathcal{G}_q$ is completed if all relevant parties have executed their part. Similarly to the ideal functionalities, instance tags can be used to keep track of multiple queries to the local functionality. Especially, for local functionalities the instance tag helps to track which calls to $\mathcal{G}_{q,j}$ make up one execution of $\mathcal{G}_q$ as we can assume all parties in $\mathcal{J}_q$ performing the same computation use the same tag.

Note that another important difference between local and ideal functionalities is that local functionalities are asynchronous, whereas, by definition, ideal functionalities need inputs from multiple parties before any party gets an output and therefore they synchronize the execution of the parties. In fact, local computations stand for the internal computations of a participant and therefore are not under the direct control of the adversary. Depending on the context, the functionality can either be inlined to the description of the party, or it can be considered as an external functionality where party $\mathcal{P}_j$ clocks the input to $\mathcal{G}_{q,j}$ and the local functionality immediately performs the computation and clocks the response message back to $\mathcal{P}_j$. Note that in RSIM the party can clock this message and in effect the receiving machine is activated so $\mathcal{G}_{q,j}$ can perform the computation and give control back to the party.

It is possible for the local functionality to implement a randomized function. In principle, allowing a randomized g_q means that any local operation could be meaningful. Note that in practice it is only reasonable to discuss local functionalities where the function g_q is known and useful. For some functionalities, it is possible that all $\mathcal{G}_{q,j}$ are deterministic but g_q is still randomised and the randomness in the output of $\mathcal{G}_q$ depends on the distribution of the inputs. Such cases need careful analysis in the context where they are used. If the same local functionality is executed again with the same inputs, then it always gives the same output, as its randomness depends on the distribution of the input shares, which has not changed. However, the intuition about implementing a random function g_q indicates that the same input may give different outputs. Hence, any such functionalities can be used under the assumption that there are no executions with the same inputs. In most cases where the function is non-deterministic, it is reasonable to expect that at least some of the functionalities $\mathcal{G}_{q,j}$ are also non-deterministic. The following expects mostly deterministic local functionalities.

Note that there exist protocols that correspond neither to ideal nor local func-

tionalities. For example, a combination where some outputs are computed by an ideal functionality, and others are computed by a local computation from some inputs does not belong to either class. In addition, there are functionalities where the outputs are computed using local operations, and some ideal functionalities are used before, but all their outputs are published and used in the local computations to derive the outputs. For example, a functionality that takes $[\![x]\!]$ and $[\![y]\!]$, publishes y and then outputs $[\![xy]\!]$ computed using local operations. In such cases, the output shares are not independent of the input shares as needed for the ideal functionality, and neither is the functionality local, as communication is needed to publish a value. However, in most cases, the protocol can be structured to be either one or the other or a simple combination of the two types of functionalities. The following description of secure multiparty computation uses these two types of functionalities.

3.6. Protection Domains

A protection domain is a collection of storage domains and computation protocols $\Pi_1, \ldots, \Pi_k$ operating with data in these storage domains. In other words, the storage domain is the name of the secure computation framework. It defines its parameters through the parameters of the storage domains, and the computation capabilities are defined based on the set of protocols.

Overall, the protection domain can be thought of as a single functionality $\mathcal{F}_{\mathsf{pd}}$. In a modular case, each protocol (e.g. a command that can be given to $\mathcal{F}_{\mathsf{pd}}$) Π_p implements some ideal functionality $\mathcal{F}_p$. The ideal functionality, in turn, contains the sharing and reconstruction functionalities for its input and output storage domains. A functionality may accept inputs and give outputs in several different storage domains.

The overall function of the protection domain is meaningful only when the different functionalities operating with the same storage domain have the same setup. This restricts the $\mathcal{F}_{\triangle}$ to give matching parameters for several functionalities that work with the same domains. Hence, for a given storage domain δ, the machines $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ in all $\mathcal{F}_p$ that use them must receive the same setup parameters. For brevity, such a collection with the ideal functionalities receiving consistent setup is denoted as $\mathcal{F}_{\mathsf{pd}}$ (the ideal protection domain) and is implemented by Π_{pd}, which is an analogous collection of protocols with consistent setup. It can be said that the concrete setup defines the instance of the protection domain.

This thesis focuses on considering the properties of modular protection domains while staying as abstract as possible regarding the exact functionality or programs that are executed. The main restriction is that the protection domain consists of several canonical ideal functionalities. For algorithm design as well as the purposes of this work, it is most natural to focus on the hybrid execution where the concrete protocols are represented by the ideal functionalities. The idea is that if some new functionality is proven secure in the hybrid execution model,

then thanks to the composition theorems, the ideal functionalities can be replaced by the concrete protocols that implement these functionalities in order to achieve real composed protocols. Hence, most of the discussion focuses on the parties interacting with the ideal functionalities $\mathcal{F}_p$ that satisfy the properties described in Section 3.4.

The storage domain usually contains functionalities to give inputs and reconstruct stored values to plain outputs. A party can send inputs to the input protocol and all necessary parties get their representation of the secure data. The functionalities in a protection domain are called with inputs in suitable storage domains. In this formalisation, the representation of the secure data moves in and out of the protection domain. Parties $\mathcal{P}_i$ execute some computation and interact with the protection domain to do so. The parties each hold their own view of the stored value. Hence, the context is that there is an outer environment $\mathsf{Env}_{\mathsf{pd}}$ (more details in Section 3.6.2) that interacts with the parties and the parties then call out specific functionalities of the protection domain with their inputs and receive the outcomes. The adversary can corrupt the parties and interact directly with the ideal functionalities (more details in Section 3.6.4 and Section 3.4.2). This setup is illustrated in Figure 13 for modular protection domains (Definition 16) where $\mathcal{F}_{\mathsf{pd}}$ is marked with the grey box.

The setup $\mathcal{F}_{\triangle}$ can be thought of as part of the protection domain. However, for most uses, the setup functionality has to remain the same even if the details of the protection domain are changed. Hence, it is often more convenient to consider the setup as a separate functionality that is used by the protection domain. In such cases, the setup can be seen as a fixed part of the environment in which the protection domain executes.

3.6.1. Modular Protection Domains

It is usually more convenient to think of the computation framework as the sum of its components rather than the full functionality $\mathcal{F}_{\mathsf{pd}}$. This is only correct if the protection domain is modular, as illustrated in Figure 13.

Definition 16. A protection domain $\mathcal{F}_{\mathsf{pd}}$ has modular representation with lightweight ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ if collections $\mathcal{F}_{\triangle}\langle\mathcal{F}_{\mathsf{pd}}\rangle$ and $\mathcal{F}_{\triangle}\langle\mathcal{F}_1, \ldots, \mathcal{F}_k\rangle$ are indistinguishable for any party interacting with them.

A modular protection domain can therefore be thought of as the collection $\mathcal{F}_{\triangle}\langle\mathcal{F}_1, \ldots, \mathcal{F}_k\rangle$ that uses canonical ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$. There is one functionality for each operation that the protection domain can execute, each of the functionalities can be called many times using tags to separate different instances. In many cases, it is reasonable to decompose these functionalities as shown in Figure 14. The universal sharing machine $\mathcal{S}_u$ is a collection of all $\mathcal{S}_\delta$ functionalities from inside the $\mathcal{S}$ machine in $\mathcal{F}_p$. $\mathcal{S}_u$ has k port pairs, one for each $\mathcal{F}_p$. A query to p-th pair is sent internally to the respective $\mathcal{S}_\delta$. Such a machine is straightforward to construct for stateless $\mathcal{S}_\delta$ as their use from different func-

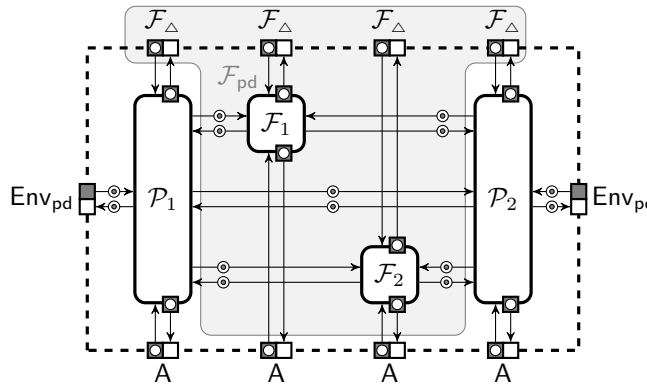


Figure 13: Usage of the modular two-party protection domain $\mathcal{F}_{\mathsf{pd}}$ with functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$.

tionalities $\mathcal{F}_p$ does not interfere with other calls from other functionalities. The universal reconstruction machine $\mathcal{R}_u$ is a similar collection of all reconstruction functionalities $\mathcal{R}_\delta$ for the ideal functionalities. As a simplification, $\mathcal{R}_u$ and $\mathcal{S}_u$ both have a single port pair connecting them to $\mathcal{F}_\triangle$, and they distribute the setup among internal machines themselves.

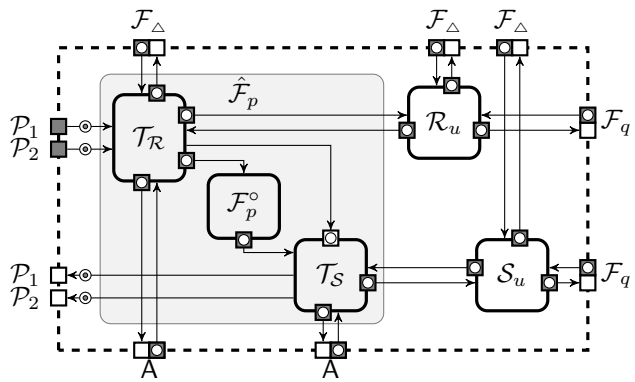


Figure 14: Canonical ideal functionality $\mathcal{F}_p$ split into functionality $\hat{\mathcal{F}}_p$ and separate machine $\mathcal{R}_u$ for universal reconstruction and $\mathcal{S}_u$ for sharing. $\mathcal{S}_u$ and $\mathcal{R}_u$ are shared with other canonical ideal functionalities $\mathcal{F}_q$.

The functionality $\mathcal{F}_{\mathsf{pd}}$ of the modular protection domain can therefore be also thought of as the collection $\hat{\mathcal{F}}_1,\ldots,\hat{\mathcal{F}}_k,\mathcal{R}_u,\mathcal{S}_u$. The collection $\hat{\mathcal{F}}_p$ represents the collection of $\mathcal{T}_\mathcal{R},\mathcal{T}_\mathcal{S}$ and $\mathcal{F}_p^\circ$. For modular ideal functionalities, it is easy to see that the collection $\mathcal{F}_1,\ldots,\mathcal{F}_k$ is indistinguishable from $\hat{\mathcal{F}}_1,\ldots,\hat{\mathcal{F}}_k,\mathcal{R}_u,\mathcal{S}_u$. If the setup is consistent and sends the same parameters to all functionalities and the machines $\mathcal{R}$ and $\mathcal{S}$ are stateless after the setup parameters have been fixed, then it is easy to see that the functionalities are modular. As such setup is reasonable, then the following mostly assumes that the protection domain is indeed modular and can be described as a collection of $\mathcal{F}_1,\ldots,\mathcal{F}_k$ or as a collection of $\hat{\mathcal{F}}_1,\ldots,\hat{\mathcal{F}}_k,\mathcal{R}_u,\mathcal{S}_u$

depending on which is more convenient for the given discussion.

Note that this description did not explicitly consider the local functionalities. For the purpose of this description, each local functionality $\mathcal{G}_{q,j}$ is inside $\mathcal{P}_j$. Later in Section 5.5.3 the local functionalities are made more explicit and separated from the parties as in that discussion using $\mathcal{G}_q$ instead of the collection of $\mathcal{G}_{q,j}$ becomes more relevant. For the general description of the protection domain here, it is sufficient to note that the parties can also carry out operations on their shares.

3.6.2. Suitable Environments

Security in the RSIM framework is defined with respect to an environment that tries to distinguish the two versions of the protocol. Overall, the environment is a very generic machine that does not have many reasonable restrictions. This is the case with the environment Env against the protection domain. Note that, for modular protection domains, some contents of the protection domain could also be considered as part of the environment when just analysing the security of some component $\mathcal{F}_p$. The following uses $\mathsf{Env}_{\mathsf{pd}}$ to specifically denote the environment that interacts with the whole protection domain. It gives inputs to the parties and receives their outputs. The inputs can also affect which computations the parties are running with the protection domain. In case the focus is on some functionality $\mathcal{F}_p$ inside the modular protection domain, then the rest of the protection domain and the parties interacting with $\mathcal{F}_p$ form Π_{e}. The full environment for the functionality is the combination $\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$.

The main restriction on $\mathsf{Env}_{\mathsf{pd}}$ is that the setup functionality $\mathcal{F}_{\triangle}$ gives consistent setup data to all components that require it. However, if the protection domain leaks too much information to the environment $\mathsf{Env}_{\mathsf{pd}}$, then it is usually impossible to prove the security of the protection domain or any specific protocol. For the specific protocol Π_{e}, it is as secure as an ideal functionality $\mathcal{F}$, if $\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\Pi_1,\ldots,\Pi_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\mathcal{F}\rangle$. However, if Π_{e} leaks the joint state like the setup parameters or share representation to $\mathsf{Env}_{\mathsf{pd}}$, then this is against the intuition of the protection domain meaning. The intuition is that $\mathsf{Env}_{\mathsf{pd}}$ uses the protection domain as a secure computation functionality and does not need to know its internals, only to give public inputs and receive outputs. Moreover, the generalized form of a security proof like derived in Chapter 5 would become impossible as the secure storage could not be abstracted away if the ideal functionality of the whole computation should still contain setup, which is usually not the case. For the purpose of this thesis, $\mathsf{Env}_{\mathsf{pd}}$ represents the world outside of the protection domain and the common interface between the protection domain, and $\mathsf{Env}_{\mathsf{pd}}$ considers plain values that are either public or belong to some specific party. In particular, shares or private setup parameters should not be given to $\mathsf{Env}_{\mathsf{pd}}$.

These restrictions are sometimes enforced by the structure of the protection domain. Specifically, the protection domain could include a protocol Π_{io} for input and output behaviour that only communicates plain values to input and result

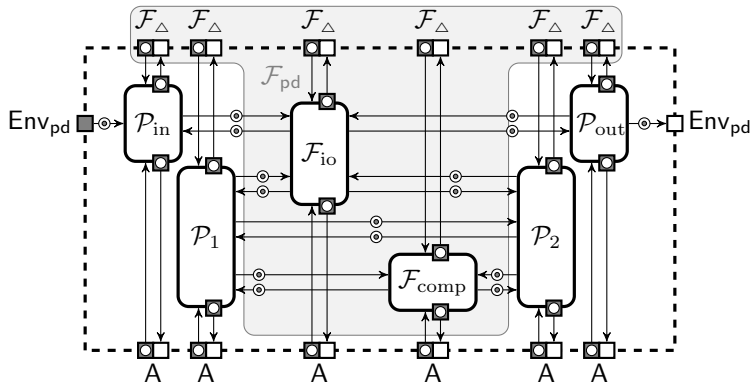


Figure 15: Usage of a protection domain with an input-output functionality $\mathcal{F}_{\text{io}}$ and the rest of the computation illustrated by $\mathcal{F}_{\text{comp}}$.

parties that are the only ones to communicate with $\mathsf{Env}_{\mathsf{pd}}$. Such structure is illustrated in Figure 15. The respective ideal functionality $\mathcal{F}_{\text{io}}$ can be thought of as a suitable collection of $\mathcal{R}_\delta$ and $\mathcal{S}_\delta$ functionalities for the storage domains δ used in the protection domain.

Depending on the use-case, the environment can also have a choice in the code that is executed. In the simpler case, the protocol can be fully specified in the Π_{e} and $\mathsf{Env}_{\mathsf{pd}}$ only sends the inputs. For example, if a certain application is considered as Π_{e}. In the more general case, the Π_{e} specifies a computation interface and $\mathsf{Env}_{\mathsf{pd}}$ sends both inputs and concrete commands. In that case, Π_{e} is defined as expecting new instructions from $\mathsf{Env}_{\mathsf{pd}}$ at certain points in its execution. For example, if the protection domain is a tool for secure statistical analysis that repeatedly gives query results to $\mathsf{Env}_{\mathsf{pd}}$.

When considering a specific protocol Π_p inside the protection domain, then the environment for this specific protocol is made up of the outer environment $\mathsf{Env}_{\mathsf{pd}}$ and the other protocol execution Π_{e} that is happening inside the protection domain that calls the protocol Π_p. Hence, for a specific protocol Π_p, the set of environments is made up of the setup and the combined environment as $\{\mathcal{F}_\triangle\} \times \mathbb{E}_{\mathsf{pd}} \times \mathbb{P}$. Here $\mathcal{F}_\triangle$ is the setup, $\mathbb{P}$ is the set of protocols running in the protection domain and $\mathbb{E}_{\mathsf{pd}}$ is the set of environments allowed for the set of protocols. The protocol $\Pi_{\mathsf{e}} \in \mathbb{P}$ encapsulates all other computations done in the protection domain and it calls Π_p. $\mathsf{Env}_{\mathsf{pd}}$ is the environment considered in the following protection domain security definition (Definition 17). For reasonable execution, $\mathsf{Env}_{\mathsf{pd}}\langle\mathcal{F}_\triangle\langle\Pi_{\mathsf{e}}\langle\Pi_1,\ldots,\Pi_k\rangle\rangle,\mathsf{A}\rangle$ is a well-defined closed collection. In the following, Env compatible with Π_p means $\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$ or $\mathsf{Env}_{\mathsf{pd}}\langle\mathcal{F}_\triangle\langle\Pi_{\mathsf{e}}\rangle\rangle$ that is the full environment against the protocol Π_p. This can be seen as always keeping the full context where the protocol is executing. However, the main goal is to define some properties of the interaction of the protocol Π_p (of $\mathcal{F}_p$) and the environment $\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$ so that the proofs are such that any suitable Π_{e} can be later used to call the designed protocol.

3.6.3. Security of a Protection Domain

The security of protection domains is considered in the usual real and ideal world paradigm. The ideal functionality of the protection domain is denoted as $\mathcal{F}_{\mathsf{pd}}$. It is common for the protection domain to be modular, as discussed in Section 3.6.1. The functionality is implemented by a series of one or more protocols $\Pi_1, \ldots, \Pi_k$ that implement parts of the functionality. Hence, in the most general case, the security of the protection domain should be done with respect to a series of protocols, as done in the following.

Definition 17. A list of protocols $\Pi_1, \ldots, \Pi_k$ with a shared setup $\mathcal{F}_\triangle$ is a secure protection domain for a class of protocols $\mathbb{P}$ and class of environments $\mathbb{E}_{\mathsf{pd}}$ if $\mathcal{F}_\triangle\langle\Pi_{\mathsf{e}}\langle\Pi_1, \ldots, \Pi_k\rangle\rangle \geq \mathcal{F}_\triangle\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_{\mathsf{pd}}\rangle\rangle$ for any $\Pi_{\mathsf{e}} \in \mathbb{P}$ and any environment in class $\mathbb{E}_{\mathsf{pd}}$.

A protection domain specifies the class of protocols $\mathbb{P}$ that can be computed securely. The security itself is parametrised by the class of environments $\mathbb{E}_{\mathsf{pd}}$ and also adversaries $\mathbb{A}$ as in the security definitions in Definition 2 and Definition 3. Since the protocols $\Pi_1, \ldots, \Pi_k$ in the protection domain are described by their ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ then the signature of the protection domain becomes $(\mathbb{E}, \mathbb{P}, \mathcal{F}_1, \ldots, \mathcal{F}_k)$. The used functionalities define the storage domains δ and the number of parties as well as some of the adversarial capabilities. The storage domains provide some limit on the adversary structure $\mathcal{A}$ but the definitions of the functionalities may limit it further. It is often the case that the adversary structure of the protection domain is defined based on all the restrictions of the storage and individual functionalities. The class of adversaries $\mathbb{A}$ is defined by the adversary structure and the expected behaviour of the adversary. Some more concrete details about the adversary behaviour in composed protocols are discussed in Section 3.6.4.

3.6.4. Adversaries Against Composed Protocols

In the real functionality, there is a machine for each party and these machines communicate with each other according to the protocol. For example, each party $\mathcal{P}_i$ could be a machine M_i in Figure 7. This communication goes over the leaky buffers meaning that the main message is transferred confidentially and cannot be changed in the buffer, but each message may have some public metadata that is seen by the adversary. In addition, the adversary is in charge of the timing of these channels. Each party $\mathcal{P}_i$ can receive a corruption message that makes it corrupted. For passive corruption, the party starts transferring all of its protocol view to the adversary. Actively corrupted machines fall under the control of the adversary and become transparent. This setup allows modelling adaptive corruption as well as static corruption. However, for static corruption, we can treat them as the class of adaptive adversaries that send the corruption messages before any other action is taken by the system and do not send new corruption messages afterwards.

Composing several MPC protocols means getting a machine for each party in

each of the sub-protocols. Formally, the adversary can corrupt machines independently, which clashes with the intuition that physical parties become corrupted. For example, in practice, one expects the adversary to either corrupt a physical entity, such as a piece of hardware that runs all the code for that party, or a person, such as an administrator managing the execution of one party. Hence, either all machines representing that party are corrupted, or none are. The following definition specifies that such behaviour is expected by coherent adversaries in the case where secure multiparty computation protocols are composed of several components.

Definition 18 (Generic adversary)**.** A class of adversaries $\mathbb{A}$ is generic if the only restriction on the adversary is the port compatibility with the protocol and environment.

Definition 19 (Coherent corruption)**.** We say that an adversary corrupts parties *coherently* if it always corrupts the machines corresponding to party $\mathcal{P}_i$ simultaneously in all systems in the composition. Such an adversary is called coherent. This class of adversaries is denoted as $\mathbb{A}_{\mathsf{c}}$.

The coherence property can be combined with other restrictions. For example, to consider either semi-honest or malicious coherent adversaries separately. Trivially, all static adversaries have coherent corruption and similarly, adaptive adversaries can follow coherent corruption. This work focuses on coherent corruption. This definition does not consider mobile corruption where the definition should be extended with the possibility to uncorrupt parties. However, a natural extension of the definition where uncorrupting means removing corruption from all machines representing the party could be considered. However, considering proactive security of the protection domain would also mean the need to extend the definition of the functionalities and storage to account for the removal of corruption.

Lemma 1 (Coherent corruption)**.** *The composed protocol $\Pi_{\mathsf{e}}\langle\Pi\rangle$ is as secure for the general class of adversaries $\mathbb{A}$ as it is for adversaries $\mathbb{A}_{\mathsf{c}}$ assuming that the protocols Π_{e} and Π have matching adversary structures and a party is considered corrupted as soon as any machine representing that party becomes corrupted.*

Proof sketch. Note that a party is considered corrupted if any machine representing that party in a composition becomes corrupted. Hence, corrupting a party in one protocol would also imply that the same party is corrupted in the other even if the adversary does not modify its execution actively. However, if the adversary breaks the adversary structure then both protocols are broken.

For any general adversary, we can create a corruption filtering machine that does two things:

- If the adversary corrupts any machine, then the filter corrupts all machines corresponding to the same machine (it gets control back from the machine after corrupting, so it can do it as one step before giving control back to the adversary).

- It maintains the list of machines that the general adversary has corrupted and allows the adversary to control these machines/see their views as expected. For the other machines, it blocks their inputs from the general adversary and commands them to act according to the protocol (e.g. semi-honestly).

Such a filter changes the general adversary to the coherent one while maintaining the same view as before for the generic adversary. Note that, as the protocols have matching adversary structures, then both can tolerate the same set of corrupted parties without breaking. Hence, any generic adversary can be turned into an equivalent coherent adversary. □

3.7. Comparison with other formalisations

The definition of the protection domain covers the MPC deployment model [4] that specifies that there can be separate input, result and computing parties. All of these parties are considered to be part of the parties interacting with the protection domain. They operate with values in different storage domains. The computing parties are using the storage domains where the private data is stored and computed with. The input parties get the plain inputs from $\mathsf{Env}_{\mathsf{pd}}$ and distribute them using the sharing functionalities for the desired secure storage domain. The result parties can either receive individual results or public results through specialised reconstruction functionalities for the storage domains. As the initial inputs are received from $\mathsf{Env}_{\mathsf{pd}}$ and also the final reconstructed results are sent back, then also $\mathsf{Env}_{\mathsf{pd}}$ is in the roles of the input and result parties.

3.7.1. Arithmetic Black-Box

The protection domain formalisation defined in this chapter can be seen as a more detailed look at the construction of the Arithmetic Black-Box model [55] described in Section 2.1.8. The protection domain and the parties interacting with it are indeed very similar to the ABB. In the big picture, in both cases, the environment interacting with the functionality can give inputs and can order specific computations in either ABB or the protection domain.

The protection domain formalism is very modular and targeted towards isolating parts of the formalisation as lightweight ideal functionalities and working with intermediate computations independently. ABB, on the other hand, is a single unit describing a secure computation functionality from inputs to public outputs. Note that ABB can also be seen as a specific lightweight ideal functionality taking public inputs and giving public outputs. Hence, a protection domain with just the ABB as an ideal functionality is a description of the same secure computation capability as the initial ABB.

It is easy to use an ABB as a formalism to define full executions of secure computation that start from inputs and finish with public outputs. However, there

is no straightforward mechanism to expand the set of commands inside the ABB description. Essentially, for an ABB, there is no explicit representation of secure data other than the handles used to interact with the values inside the ABB. Hence, there is no good way to discuss the representation of secure data or its movement between parties or protocols. Data is explicit only in the input and output operations of the ABB. In addition, the security of the framework is defined with respect to the full ABB description. Hence, if the framework is extended with a new operation, then the security of all of its components should be proven again with respect to the new ABB that contains the new operation. If the original security proof does not make any specific assumptions about the intermediate values in the computation, then the proof of most components may carry over very easily. However, if, for example, the proof uses some assumptions about the intermediate representation of data that is kept by all previous operations but not by the added protocol, then also the security proofs of the previous components have to be repeated to lift this assumption.

An important part of any security proof is ensuring that the simulation can take the outputs of the ideal functionality and adjust the simulated protocol so that it gives the same outputs. The protection domain approach with canonical ideal functionalities gives strict rules that protocol outputs are generated from the output value using the secret sharing functionality. This enables adding new functionalities to the protection domain as assumptions can be made about the protocol inputs but may complicate security proofs. For example, in order to prove that some protocol is as secure as a canonical ideal functionality, the proof has to be able to use the outputs of the ideal functionality as the outputs of the protocol. These values are in some secure storage domain and may be difficult to align. Whereas for an ABB only public values are outputs and a common simulation strategy assumes that any adjustments between the values in the ABB and in the simulation of the protocol have to occur when public outputs are given. The secure data representation is not exposed outside the ABB and there is no need to align this. The proof in the ABB still has to take into account all the possible operations that may produce the outputs in order to be able to show that their publishing can be simulated correctly. In a sense this is simpler for the protection domain case if we assume all functionalities give outputs with the same distribution as the secret sharing functionality. However, in both cases it is often the local functionalities that introduce different distributions and should be addressed separately. This is often overlooked in ABB based proofs when claiming that all shares can be simulated as random values and the difficulty is made explicit in the protection domain view.

The fact that a secure computation framework is as secure as an ABB does not immediately say that any algorithm description implemented using this is secure. If the algorithm does not publish any values, then its security can be derived directly. However, if there are any intermediate values published by the algorithm that are not considered to be the outputs of the algorithm, then the security has

to be evaluated independently. The same is true when using protection domains. Chapter 5 works on simplifying the protection domain formalism to show that, in many cases, it allows for quite simple modular security proofs of new algorithms for MPC frameworks.

3.7.2. Simplified Universal Composability

Simplified universal composability (SUC) is introduced in Section 2.2.4. Some of the similarities and differences between SUC and protection domain approaches are outlined in Section 2.2.6 as the restrictions posed by RSIM carry over to the protection domain approach. This section summarises the main aspects with a focus on the protection domain description.

The main property of SUC defined in [44] is that there is a fixed number of parties. The same restriction is there for the protection domain formalisation here. In addition, SUC considers only authenticated network channels. The buffers in the RSIM framework are not modifiable and any communication that is through a buffer or the leaky buffer constructed in Section 2.2.5 is also authentic. Protection domains also expect the communication channels to be secure, except for the metadata leaked by the leaky buffers. In both cases, the timing of the protocols is under the control of the adversary. For the protection domain, the communication between the domain and the environment is also under the control of the adversary, which is not the case for SUC. For the purpose of the security definitions, the exact model of the communication between the MPC formalism and the environment is not that relevant. For example, the adversary in SUC can get a similar control via direct communication with the environment, or the adversary against a protection domain formalisation can clock the communication to and from the environment so that it achieves the same timing as in SUC. Hence, if the interaction between the environment and the adversary is not restricted, then either model can be considered a special case of the other in this regard.

In SUC, there are separate copies for ideal functionalities if the same functionality is required multiple times. The same idea is carried by the definition of canonical ideal functionalities that do not share state over the executions of several instances. In SUC, all new operations that the party can do will be inlined to the party code. In the protection domain, one can consider new protocols either as part of the party in Π_e or the machine representing the party $\mathcal{P}_i$ can be split into several functionalities as will be useful in Chapter 5. Similarly, protection domain formalisation allows separating local functionalities from the party so that their timing is not under the control of the adversary. Hence, this formalisation allows more flexibility to play around with the representation of the computation. However, this means that the protection domain version is more complicated and needs the definition of the coherent adversary. Similarly to SUC, the protection domain formalism does not allow to uncorrupt a party or corrupting it only for some protocol instance. Hence, mobile adversaries (also known as proactive security) cannot

be considered in either of the frameworks.

The hybrid execution defined in SUC is, therefore, quite like the protection domain where the parties run some code and call ideal functionalities. Their formalism specifies a protocol calling one ideal functionality. However, it can be a complex functionality that internally runs all the functionalities used to specify the operations of the protection domain. Treating everything as one functionality also means that the properties that the current formalism requires from the setup are not necessary. The ideal execution in SUC is defined by the variation of parties running an ideal model protocol. This is in line with the variation considered here, where the parties run a code that simply calls the ideal functionality and returns the result. Note that similarly to our case, the local operations cannot be considered as separate ideal functionalities in the SUC model because the timing of ideal functionalities is under the control of the adversary.

4. FROM INPUT-PRIVATE TO UNIVERSALLY COMPOSABLE PROTOCOLS

Protocols for secure multiparty computation often end with publishing some value that is considered to be the output of the protocol. In practice, the protocol also has some representation of the intermediate values. To preserve security, the intermediate representation should not reveal information about the values in the protocol. In secure computation frameworks, many protocols do not have a published output at all. Instead, the output is only existing in some hiding storage domain. In such cases, the security of the protocol is directly linked to the hiding property when it comes to the outputs. However, to ensure that the protocol does not reveal anything about the inputs, it is still important to consider the intermediate computations and all sent messages. This chapter considers the input privacy property that is suitable for managing these cases. Some related work is discussed in Section 2.1.3.

This chapter defines the input privacy property and respective ideal functionalities. These ideal functionalities are like the canonical ideal functionalities (Definition 14), but the input-private protocols do not have to satisfy the restrictions that the canonical ideal functionality puts on the outputs of the protocols. As for the more common security property, a protocol can be proven to be as input-private as the defined ideal functionality. Secondly, this chapter defines a new flavour of protocol composition called ordered composition. The ordered composition is focused on the data dependency rather than the execution timing between the protocols. The two main results of this section are the results that say that the composition of two private protocols is a private protocol, and an ordered composition of input-private and secure protocols is secure if they have a jointly predictable outcome. These results hold for the case of black-box simulation, where the new simulator can only interact with the adversary and not replace it fully. While this is the common way how simulators are designed, it still remains a special case of the security definition that just requires the existence of an equivalent adversary.

Theorem 3 (Security of a composition of secure and private functionality, informal)**.** *The fully ordered composition of black-box input-private and black-box secure protocols with the jointly predictable outcome is black-box secure.*

Theorem 4 (Privacy composition, informal)**.** *The ordered composition of black-box input-private protocols is black-box input-private.*

These theorems are formalised for the passive security case as Theorem 6 and Theorem 5, respectively. Sections 4.1 to 4.6 build the definitions and results necessary to prove these theorems. Section 4.8 demonstrates how these results help to build more efficient practical protocols. The main observation is that there are protocols that are input-private and not secure but are more efficient than the fully secure protocol with the same functionality. Then, this protocol could be composed with a secure protocol to get a secure protocol that is more efficient than

the protocol from composing two secure protocols. Finally, Section 4.10 discusses how this approach might be extended to the case of active security.

Note that some of the definitions in this chapter are applicable in a more general case than that of the protection domains for secure multiparty computation described in Chapter 3. Hence, this section uses more of the notation of the RSIM framework (Section 2.2.5), especially that of the systems specifying different structures and service interfaces and considering the view of the machine in a special configuration. Specific remarks are made to show how the definitions used here differ from the MPC formalism in Chapter 3 and how they can still be mapped to the MPC formalism.

4.1. Input Privacy

Intuitively, a protocol preserves the privacy of its inputs if the protocol execution does not leak any information about private inputs of the parties. Input privacy is a useful notion for protocols where the output privacy is guaranteed by some other means. For example, the outputs are in some hiding storage domains. However, the real protocol may send many messages that are somehow leaking information about the protocol inputs. Hence, the definition of input privacy should explicitly show that the outputs of the protocol are not considered, but the rest of the protocol interactions are.

For the input privacy definition, consider an environment that does not use the values it gets back from the computation. It only distinguishes configurations based on the adversary's view of the protocol and its knowledge of the inputs given to the protocol. Formally, this is represented by separating the environment into two distinct parts and limiting their communication with each other and partially with the adversary. Hence, Env is modelled as a composition of two machines $\mathsf{Env} = \mathsf{Env}' \cup \mathsf{Env}_\perp$. Here, Env' gives the inputs to the system and communicates with A while only $\mathsf{Env}_\perp$ learns the outputs of the system. $\mathsf{Env}_\perp$ is not allowed to give any information to the adversary or to Env. Especially, it has no buffers to send messages to these machines. Other than that, the environments have to be compatible with the system. This privacy configuration $conf = (\mathfrak{M}, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{A})$ can be seen in Figure 16 where $\mathfrak{M}$ could be expanded similarly to Figure 7. The crucial component of this setup is that $\mathsf{Env}_\perp$ does not send messages to Env' or A directly. Note that if any machine, for example, Env' finishes execution without clocking another machine then the control goes to the master scheduler, in this case A.

The privacy configuration gives rise to the following input privacy definition that is analogous to the security definition.

Definition 20 (Input privacy)**.** System $Sys_1 = (\mathfrak{M}_1, S)$ is perfectly at least as input-private as $Sys_2 = (\mathfrak{M}_2, S)$ (denoted as $Sys_1 \geq_{priv}^{perf} Sys_2$) if, for every privacy configuration

$$conf_1 = (\mathfrak{M}_1, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{A}_1) \in Conf(Sys_1)$$

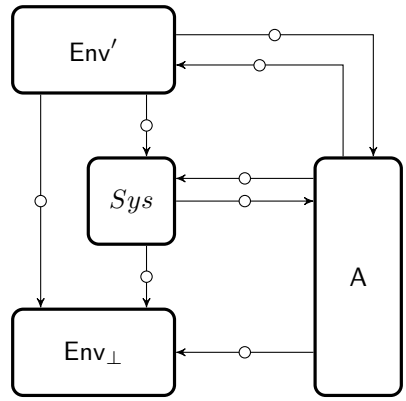


Figure 16: Privacy configuration with two distinct parts of the environment $\mathsf{Env} = \mathsf{Env}' \cup \mathsf{Env}_\perp$.

where $\mathit{ports}(\mathsf{Env}) \cap \mathit{forb}(\mathfrak{M}_2, S) = \emptyset$, there exists a privacy configuration

$$\mathit{conf}_2 = (\mathfrak{M}_2, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{A}_2) \in \mathit{Conf}(\mathit{Sys}_2)$$

with the same $\mathsf{Env} = \mathsf{Env}' \cup \mathsf{Env}_\perp$, such that the environment views restricted to Env' coincide

$$\mathit{view}_{\mathit{conf}_1}(\mathsf{Env}') = \mathit{view}_{\mathit{conf}_2}(\mathsf{Env}') \ .$$

The above definition can be extended to computational and statistical input privacy in the traditional way. Statistical input privacy requires statistical indistinguishability of the views. For the computational case, it is necessary to consider only polynomial-time configurations and the computational indistinguishability of the respective views. Furthermore, different restrictions on the adversary can be applied to consider some special security models. Black-box privacy, where $\mathsf{A}_2 = \mathsf{Sim} \cup \mathsf{A}_1$ and the simulator Sim only depends on the system and A_1, and not on the environment, can also be defined. Theorem 5 will also show a specific sense in which input privacy is composable.

For a simple example, a composition of two independent input-private systems that do not communicate with each other is input-private. To prove this, either system can be considered as running in the context where the other system is included in the output part $\mathsf{Env}_\perp$ of the environment. This is a suitable privacy configuration for the remaining system.

Note that not explicitly considering the outputs is the main thing distinguishing input privacy from the security property. Discarding the view of $\mathsf{Env}_\perp$ means that systems Sys_1 and Sys_2 where $\mathit{Sys}_1 \geq_{\mathit{priv}} \mathit{Sys}_2$ may give different outputs. Hence, the correctness of Sys_1 with respect to Sys_2 should be evaluated separately. Instead of considering correctness directly as part of this definition, it will be addressed as the predictability of the output of either system. However, discarding the output is also part of the strength of the definition. In security proofs, it is the job of the simulator to make it so that the outputs the adversary sees in the simulation

match that of the functionality, but this is not required in the input privacy proof. Rather, to prove input privacy, the simulator must be able to simulate the view of the protocol without knowing the inputs of the honest parties. Hence, if this simulation and the real view are indistinguishable, then the protocol keeps the privacy of the inputs of the honest parties.

An ideal private functionality is such that it does not give the adversary anything except the corrupted parties' inputs. Hence, any protocol that is as input-private as that ideal functionality is such that the (corrupted) outputs of the protocol do not reveal significant information about the honest parties' inputs. For the black-box case, the simulator then has to simulate the whole view of the protocol, including the outputs of corrupted parties, based on the corrupted inputs, hence stressing that the outputs or trace of the protocol does not reveal anything extra about the inputs. Note that this significantly limits the values that an input-private protocol can publish during its execution.

4.1.1. Input Privacy and the General Security Definition

The input privacy definition is a special case of the general form of the security definitions Definition 51 in Section 2.2.3. In secure multiparty computation, the initial systems represent some real protocols. A system consists of a collection of machines and a defined interface. Hence, $Sys = (\Pi, S)$ in the notation Π used for the collection in Section 2.2.3. In these terms, the input privacy can be defined as follows.

Definition 21 (Input privacy)**.** Let Π_1 and Π_2 be collections with an identical interface for $\mathsf{Env}' \cup \mathsf{Env}_\perp$. Let the class $\mathbb{E}_{priv}$ of environments consist of polynomial time environments $\mathsf{Env} = \mathsf{Env}' \cup \mathsf{Env}_\perp$ such that the data can only flow from Env' to $\mathsf{Env}_\perp$ as illustrated in Figure 16. Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be the set of adversaries compatible with Π_1 and Π_2 respectively. Then Π_1 is as input private as as Π_2 (denoted as $\Pi_1 \geq_{priv} \Pi_2$), if there exists a construction $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\mathsf{Env}'\langle \Pi_1, \mathsf{A}_1 \rangle \equiv \mathsf{Env}'\langle \Pi_2, \rho(\mathsf{A}_1) \rangle$$

for all $\mathsf{A}_1 \in \mathbb{A}_1, \mathsf{Env}' \cup \mathsf{Env}_\perp \in \mathbb{E}_{priv}$.

The classes of adversaries can be restricted to achieve different versions of input privacy. For perfect input privacy, the runtime of the adversary is not bounded. For computational input privacy, the adversary has to run in polynomial time. For the following theorems, we assume that the construction ρ defines a simulator that acts as an interface between Π_2 and A_1. The simulator has to be efficient and has to produce the messages for A_1 according to the specification of Π_1. The simulator has access to A as a separate machine in the configuration, but it does not have access to the code of the adversary.

The following of this chapter uses the notation with *Sys* as some of the following definitions benefit from the exact notation of the ports and specified interfaces which is more explicit in this notation.

4.2. Ordered Composition

Composability properties usually consider the timing of protocol execution. For the universal composability case, the timing is arbitrary. The sequential composition model can be used if one wishes to simplify the understanding of the execution timing to only consider that other actions may happen before or after the protocol and not during it. However, in addition to timing, other details of the protocol composition can be considered. In the following, rather than restricting timing, the data dependencies between the composed protocols are restricted to only go in one direction. For example, protocol Π_1 can send its outputs to Π_2 only if it does not expect anything from Π_2 itself.

In the case of arithmetic circuits, the gates of the circuit can always be sorted using topological sort to arrive at an ordering where, for any gate, the operations on the input wires to that gate have a smaller index than the given gate. Hence, based on the topological order, any gate may only get inputs from previous gates and give outputs to the later gates. Hence, a protocol can be built for this circuit from the protocols for the gates such that each separate protocol only has one-way communication. Note that full topological order is not necessary to compose an arithmetic circuit in an ordered manner, but the existence of the topological order shows that it can always be done.

Recall that $\mathit{forb}(\mathfrak{M}, S)$ is the collection of ports belonging to the machines in $\mathfrak{M}$ used to either connect these machines to each other or for the ports in S and $\overline{S}$ to connect to the other parts of the system and the adversary respectively. Each port p connects to its complement p^c. The ports in $\mathit{free}(\mathfrak{M})$ are the ones that other machines outside $\mathfrak{M}$ can connect to.

Definition 22 (Ordered composition)**.** The ordered composition of two structures $\mathit{Sys}_1 = \{(\mathfrak{M}_1, S_1)\}$ and $\mathit{Sys}_2 = \{(\mathfrak{M}_2, S_2)\}$ is defined if the structures are composable, meaning that the port structures are compatible

$$\begin{aligned} \mathit{ports}(\mathfrak{M}_1) \cap \mathit{forb}(\mathfrak{M}_2, S_2) &= \emptyset \ , \\ \mathit{ports}(\mathfrak{M}_2) \cap \mathit{forb}(\mathfrak{M}_1, S_1) &= \emptyset \ , \end{aligned}$$

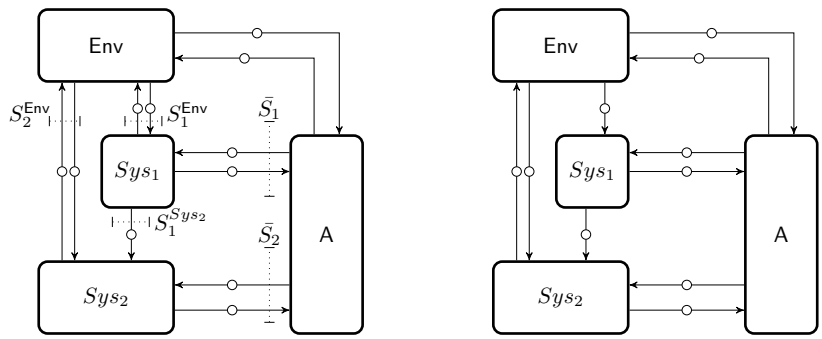
and input-output ports match

$$S_1 \cap \mathit{free}([\mathfrak{M}_2])^c = S_2^c \cap \mathit{free}([\mathfrak{M}_1]) \ .$$

The composition is ordered from Sys_1 to Sys_2 when the data flow is limited to passing from $\mathfrak{M}_1$ to $\mathfrak{M}_2$, especially all ports $S_1 \cap \mathit{free}([\mathfrak{M}_2])^c$ are output ports and all ports $S_2 \cap \mathit{free}([\mathfrak{M}_1])^c$ are input ports to only allow unidirectional message flow from $\mathfrak{M}_1$ to $\mathfrak{M}_2$. The ordered composition with data flow from Sys_1 to Sys_2 is denoted as $\mathit{Sys}_1 \to \mathit{Sys}_2$.

Definition 23 (Fully ordered composition)**.** The ordered composition $\mathit{Sys}_1 \to \mathit{Sys}_2$ is fully ordered if all the outputs of $\mathfrak{M}_1$ go to $\mathfrak{M}_2$, i.e., all output ports in S_1 belong to $\mathit{free}([\mathfrak{M}_2])^c$.

Note that there is no limitation of the timing of the execution or the buffer clocking, just the direction of data flow is limited. Note that the topological order of the arithmetic circuit gives us an ordered but not fully ordered composition. However, if needed, then each gate can be enhanced with added copying gates to pass through the unneeded inputs from previous systems and make it to a fully ordered composition. On the other hand, machines that are not communicating are trivially in ordered composition, but they cannot be in fully ordered composition. An illustration of the two modes of ordered composition is given in Figure 17.



(a) Ordered composition.

(b) Fully ordered composition.

Figure 17: Ordered and fully ordered composition $Sys_1 \to Sys_2$ where $S_1 = S_1^{\mathsf{Env}} \cup S_1^{Sys_2}$ and $S_2 = S_2^{\mathsf{Env}} \cup S_2^{Sys_1}$ meaning that $S_1^{Sys_2} = (S_2^{Sys_1})^c$.

This definition is a complement to the more common composition definitions and can be used in combination with them. Note that while the definition focuses on the data flow, it is clear that a functionality that needs an input from a previous functionality in ordered composition cannot execute before the previous functionality has given it its input. Hence, it does propose some timing constraints. However, when considering reactive functionalities, it is possible for the two functionalities to still run in parallel and have ordered composition as well.

Ordered composition is necessary to further discuss the idea that only intermediate values in the protocol are allowed to be less secure than required by the traditional UC security definitions. Hence, the following focuses on studying ordered compositions where the first protocol is only input-private and the final protocol has universally composable security.

The ordered composition $\Pi_1 \to \Pi_2$, in general, means a composition with no data flow from Π_2 to Π_1. Π_1 gets all inputs from Env and sends all outputs to Env or to Π_2. It also has specific ports for communicating with the adversary. The composition is fully ordered if all outputs of Π_1 are given as inputs to Π_2.

Note that fully ordered composition can be further simplified to cases where all inputs are given to Sys_1 as shown in Figure 18. To be more precise, Sys_1 can always be extended to Sys_1' so that it contains a machine to copy all inputs that should be directly given to Sys_2 and the machine Sys_1. Note that for a secure copy and Sys_1 functionalities, the system Sys_1 is secure. It will also be proven in

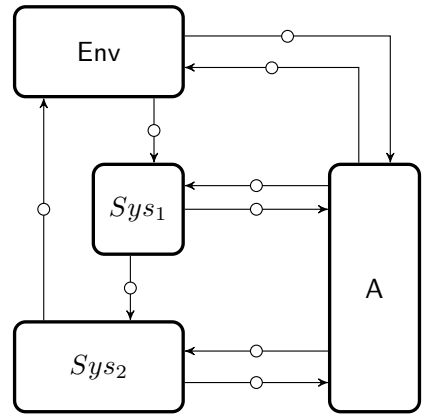


Figure 18: Simple fully ordered composition.

Section 4.6 that if both components are input private, then so is Sys_2. Furthermore, a copying functionality that simply has an output port for each of its input ports and on each input, it copies the input to the respective output port and clocks the output can be a local functionality and is, therefore, input private. Much of the following assumes this simplified case as it makes the composition structurally simpler but does not lessen the generality of the protocol or the results.

Definition 24 (Simple fully ordered composition)**.** The ordered composition of two systems $Sys_1 \to Sys_2$ simple fully ordered composition if all the outputs of $\mathfrak{M}_1$ go to $\mathfrak{M}_2$ and all inputs of $\mathfrak{M}_2$ come from $\mathfrak{M}_1$.

4.3. Corruptible Ideal Functionalities for Secure Protocols

Overall, the ideal functionalities for MPC behave as specified before in Section 3.4. This section defines some details of the ideal functionality slightly differently. Some of this is due to the fact that this section does not use leaky channels, and instead, more ports are used to distinguish the roles of the inputs and outputs. Secondly, this specification is focused on the semi-honest security model where the adversary cannot affect the functionality.

Definition 25 (Semi-honestly corruptible ideal functionality)**.** A structure $\{(\mathfrak{M}, S)\}$ is an ideal semi-honestly corruptible functionality for n parties if it meets the following conditions.

1. $\mathfrak{M}$ contains only one machine $\mathcal{F}$.
2. The ports in S are partitioned to $S^1 \cup \ldots \cup S^n$ where S^i contains the ports connecting to party $\mathcal{P}_i$.
3. $\mathcal{F}$ has all the ports in S and ports $\mathsf{in}_{\mathcal{F}}?$ and $\mathsf{out}_{\mathcal{F}}!$ for communicating with the adversary.
4. It has no additional ports and no buffers from $\mathcal{F}$ to $\mathcal{F}$.

Note This means that all the ports specified before are connected through

buffers to other machines like $\mathcal{P}_i$ and A.

5. It writes at most once to each output port per protocol instance.
6. $\mathcal{F}$ expects at most one input for a protocol instance from each input port.

   Note There can be multiple inputs from each party (or outputs to that party), but a separate buffer is used for each input (or output). This could be replaced by using one leaky buffer to give the adversary the information about which protocol messages are available in the buffers.
7. Each output is produced when all inputs necessary for computing it have been received. There can be multiple outputs computed at different times.

   Note For the functionalities in the canonical form (Definition 14), each output is computed if the input can be reconstructed by $\mathcal{R}$, the output can be computed and distributed by $\mathcal{S}$ and $\mathcal{T}_{\mathcal{S}}$.
8. The commands from $\mathsf{in}_{\mathcal{F}}$? do not affect the input-output behaviour of $\mathcal{F}$ for ports in S.

   Note The adversary cannot change the behaviour of $\mathcal{F}$ for anything other than the messages received by the adversary specified in point 9.
9. On input $(corrupt, i)$ from $\mathsf{in}_{\mathcal{F}}$? the behaviour diverges for private and secure functionalities as follows. In both cases, $\mathcal{F}$ clocks the buffer connected to $\mathsf{out}_{\mathcal{F}}$ whenever it writes there.
   - An input-private functionality will write all inputs received from S^i to $\mathsf{out}_{\mathcal{F}}$ and clock that buffer. In the future, it will forward all new inputs received from S^i in the same manner.
   - A secure functionality forwards all inputs received from S^i as well as outputs sent to S^i in the same manner.
   - In both cases, the input and output are sent together with an indicator about which buffer was used to send it. The format used is $(output, \ell_x, x, i)$ and $(input, \ell_x, x, i)$ for message x for party $\mathcal{P}_i$. The label ℓ_x can contain the information about the buffer as well as any other metadata about this message.

   Note The corruption does not modify the behaviour of $\mathcal{F}$ in any other way. This is reasonable for the passive security model, where the adversary cannot change the protocol execution.

   Note The buffer is indicated to specify the role of the output, as all output buffers are used once per protocol instance.
10. $\mathcal{F}$ does not react to any other commands from $\mathsf{in}_{\mathcal{F}}$ or write anything else to $\mathsf{out}_{\mathcal{F}}$.

Note that the main difference from canonical ideal functionalities is that in Definition 25, each port is written to only once per protocol instance since it will make the structural compositions easier. This allows separating between ports connecting to machines in the composition or to Env. In addition, thanks to fo-

cusing on the semi-honest case, $\mathcal{F}$ does not expect any adversarial commands and always executes the desired functionality. The adversary can simply receive values from $\mathcal{F}$. The restriction on ports could be easily lifted to allow multiple messages from one port that use tags to distinguish the role of the message in the protocol using leaky buffers. Hence, this can be simply seen as a special description of the canonical ideal functionality with semi-honest corruption mode. The original description of the canonical ideal functionalities assumes that there are parties that give their inputs and outputs to the adversary. For the semi-honest case, the version where also the ideal functionality shares them is equivalent since the adversary could not modify these values anyway.

Note that since the Definition 25 diverges for ideal secure and input-private functionalities, one can draw no direct conclusions about whether or not a functionality that is secure is also private. By intuition, not all secure protocols are private. For example, a secure protocol may give a public output which is also seen in protocol messages. By intuition, security is defined as leaking nothing other than the output, while privacy is more restricting and allows no leakage about the inputs. Hence, a public output that is computed from the private inputs is usually violating input privacy. However, a secure functionality that gives protected outputs in a hiding storage domain is usually input-private as then the outputs of a protocol could be simulated using a simulator from the hiding definition (Definition 8) instead of the real value. Hence, any secure protocol with sufficiently protected outputs is also input-private.

Note that an input-private ideal functionality version of Definition 25 does not give outputs to the adversary hence justifying the name *privacy*. The intuition is that, in such a protocol, all that the adversary sees are the inputs of the corrupted party. The adversary could see some outputs in a real protocol implementing this functionality. However, if the real protocol is as input-private as this ideal functionality, then it is guaranteed that the output of the protocol does not reveal information about the honest parties' inputs. Another way to say it is that for input-private protocols, the adversary has the same view distribution of different protocol runs as long as the inputs of the corrupted parties remain the same. For example, such values could be either random values generated in the protocol or protected values that do not reveal any real insight about the protocol, even if they are seen by the adversary.

4.3.1. Real Protocol Structure

Overall, the goal is to not overly restrict which real protocols can be considered. However, this chapter does make some assumptions regarding the communication with the adversary and the clocking of the system *Sys*.

Firstly, each party $\mathcal{P}_i$ has dedicated communication channels for the adversary. These can be used for sending corruption requests and sending any messages that the adversary should be aware of. Without loss of generality, each party in the

system could be corrupted and therefore needs some connection to the adversary. We assume that each party $\mathcal{P}$ has ports $\mathsf{out}_{i,Sys}!$ and $\mathsf{in}_{i,Sys}?$. These are used to respectively send messages to A or get the commands from A. As we are considering a passive security model where the party keeps following the protocol, then it suffices to assume that the only message received from A is the corruption request. After that, each action of the party is reported to $\mathsf{out}_{i,Sys}!$. The adversary is responsible for clocking the buffers $\mathsf{out}_{i,Sys}$ and $\mathsf{in}_{i,Sys}$. Note that, in case of active corruption, the party does not send any messages on its own. Hence, the formalism of MPC in Section 3 is more geared towards each party clocking the buffers to the adversary. However, in this chapter, considering the case, where a party does not clock this message, leaves the structure of possible allowed *Sys* to be more open as then the party could be decomposed to several machines that clock inputs to each other. Note that the case where each party clocks their message to A is equivalent for adversaries that always give control back to the party.

The real protocol contains several parties that communicate with each other. In general, we assume that such party-to-party communication is clocked by the adversary. For each such network communication buffer net_j, the adversary has the control of clocking it. For simplicity, we assume that each party reports all inputs that it receives and any outputs that it computes in a specific format. Moreover, to make the timing of the protocol more explicit, we assume that even the honest parties notify the adversary about which kinds of outputs were computed (for example, to which ports these outputs were written). The messages to the adversary are $(input, \ell_x, x)$ and $(output, \ell_y, y)$ if the party is corrupted and reports the message content x and y. For honest parties, just some metadata stored in ℓ_x is reported as $(input, \ell_x)$ and $(output, \ell_y)$. These messages are written when the outputs are written to the output buffer and are clocked by the adversary. Note that such behaviour of honest parties is there for just simplifying the discussion about the protocol execution. The adversary controls the network buffers and, therefore, knows which inputs it has delivered and which actions can be taken. Hence, this convenience here is an alternative to the leaky buffers used elsewhere in this thesis as using separate messages makes the following discussion about simulators more straightforward.

Note that the clocking of the communication between the real system and the ideal system differs. The real system assumes passive communication with the adversary where the party writes the information intended for the adversary to the buffer but does not clock this. Such behaviour allows for a more general description of the real system, as the party could then clock some other machine in the system. However, it is only reasonable for a passive adversary as the adversary does not modify the actions of the party and only needs knowledge about what is happening. The communication to the adversary is clocked by the adversary itself. If the party does not clock any buffers, then the adversary always gets the control after the party finishes and can immediately clock these channels. This would have the same effect as the party clocking them itself. On the other hand, the

ideal functionality is defined to clock its messages to the adversary. This is for simplicity, as the ideal functionality has no other buffers to clock. With such an ideal functionality, it is clear that the adversary has immediate knowledge about all information that is available to it in the execution. Note that the difference in clocking does not significantly complicate the security proofs. The simulator has to anyway hide the internal structure of the real system as well as it has to translate the interfaces (set of buffers) that either version has for the adversary.

4.3.2. Composition of Ideal Functionalities

In this thesis, an ideal functionality is always a single machine with well-defined ports and interfaces. Hence, a straightforward composition of two or more ideal functionalities is not an ideal functionality by this definition. This section shows how a composition of ideal functionalities can be modified to arrive at the specification of the ideal composed functionality that is also an ideal functionality according to Definition 25.

The ideal functionality composition of $\mathcal{F}_1$ and $\mathcal{F}_2$ that gives a new ideal functionality is denoted as $\mathcal{F}_1|\mathcal{F}_2$. The shorthand idea is that this composition behaves mostly like regular composition, but the values computed by $\mathcal{F}_1$ that are processed by $\mathcal{F}_2$ are not revealed to the adversary, and the adversary has no control over their timing.

Definition 26 (Ordered ideal composition of ideal functionalities)**.** Let $Sys_1 = \{(\{\mathcal{F}_1\}, S_1)\}$ and $Sys_2 = \{(\{\mathcal{F}_2\}, S_2)\}$ be two ideal functionalities for n parties such that the ordered composition $Sys_1 \to Sys_2$ is defined. Let $\mathsf{in}_{\mathcal{F}_j}?$ and $\mathsf{out}_{\mathcal{F}_j}!$ be the ports that the adversary uses to communicate with $\mathcal{F}_j$. Partition the input and output ports of $\mathcal{F}_j$ as $S_j = S_{j,\mathsf{in}} \cup S_{j,\mathsf{out}}$. The ordered ideal composition is the following ideal functionality $Sys = (\{\mathcal{F}\}, S)$ where

- $S = S_{1,\mathsf{in}} \cup (S_{2,\mathsf{in}} \setminus S^c_{1,\mathsf{out}}) \cup S_{2,\mathsf{out}} \cup (S_{1,\mathsf{out}} \setminus S^c_{2,\mathsf{in}})$;
- ports in S are divided among the parties according to S_1 and S_2. For $\mathcal{P}_i$ the ports are $S^i = (S^i_1 \cup S^i_2) \cap S$;
- In addition to ports in S, the machine $\mathcal{F}$ has ports $\mathsf{in}_{\mathcal{F}}?$ and $\mathsf{out}_{\mathcal{F}}!$ for communicating with the adversary;
- Machine $\mathcal{F}$ executes by executing $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively based on the received inputs;
- On input $(corrupt, i)$ from $\mathsf{in}_{\mathcal{F}}?$ the machine will behave as defined in Definition 25 for the party $\mathcal{P}_i$ ports S^i.

Lemma 2. *For all ideal machines $\mathcal{F}_1$ and $\mathcal{F}_2$ the machine $\mathcal{F} = \mathcal{F}_1|\mathcal{F}_2$ can be obtained from $\mathcal{F}_1$ and $\mathcal{F}_2$ and a filtering machine* Filter *as in Figure 19 where* Filter *is uniquely determined by the $\mathcal{F}_1$ and $\mathcal{F}_2$.*

Proof. In Definition 26, the behaviour of $\mathcal{F}_1|\mathcal{F}_2$ is defined based on the ports in S intended for communication with Env. Notably, the communication with the

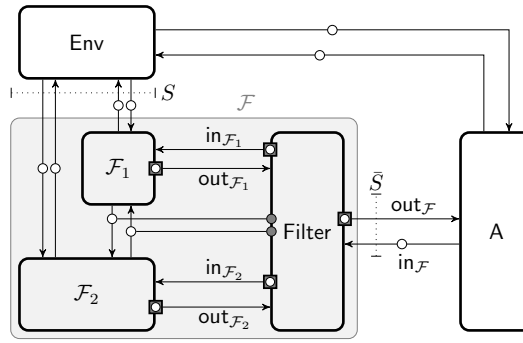


Figure 19: Decomposition of a composed ideal functionality.

adversary on ports $\mathsf{in}_{\mathcal{F}}$ and $\mathsf{out}_{\mathcal{F}}$ is defined based on the connections between the two machines. The filtering machine Filter is specified as follows.

- It has the $\mathsf{in}_{\mathcal{F}}?$ and $\mathsf{out}_{\mathcal{F}}!$ ports free to connect to A.
- It connects to $\mathsf{in}_{\mathcal{F}_1}?$, $\mathsf{out}_{\mathcal{F}_1}!$, $\mathsf{in}_{\mathcal{F}_2}?$, and $\mathsf{out}_{\mathcal{F}_2}!$.
- It has no other ports.
- For all messages from A on $\mathsf{in}_{\mathcal{F}}?$, it forwards (writes and clocks) the message to respective $\mathsf{in}_{\mathcal{F}_j}?$. Note that if it needs to be sent to both, then it can do so by waiting for the response from the first machine and then sending it to the second.

  Note According to Definition 25 the ideal functionality does not do other actions than collecting the internal values that are needed and sending them to A.

- Upon receiving the message from $\mathsf{out}_{\mathcal{F}_j}!$ the Filter has to really filter the messages. First, it unpacks the message to learn all ports and roles (inputs or outputs) of all messages. If there are output messages for ports not in S, then the Filter goes to *collection* mode if not already in this mode. Collection mode is used to put together the message m for A. All ports not in S with output messages are added to P.
  - All messages sent or received by $\mathcal{F}_j$ from port $p \in S$ are collected to message m.
  - Then Filter removes the next port p from P and clocks the corresponding outgoing buffer connected to port p. $\mathcal{F}_i$ receives this message.
  - If P is empty, then the collection mode is finished and the collected message m is sent and clocked to A using $\mathsf{out}_{\mathcal{F}}!$. The Filter can format this message as needed.

    Note Each message contains information about the buffer. The Filter only forwards the messages to and from Env in the interface S and not the messages sent between $\mathcal{F}_1$ and $\mathcal{F}_2$.

    Note Filter always gets control back from $\mathcal{F}_i$ when clocking the buffer from $\mathcal{F}_j$ to $\mathcal{F}_i$ as $\mathcal{F}_i$ reports the inputs that it received and the

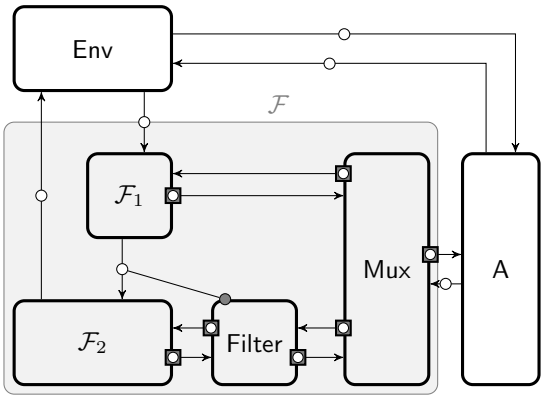


Figure 20: Decomposition of a composed ideal functionality for the ordered composition of the private and secure protocol.

outputs that it computed.

- Filter always clocks the buffer to where it writes a message to.

Note At each activation, it writes at most one message to its outgoing buffers.

Also, note that the interface to Env remains the same as ideal functionalities must have unique ports for all inputs and outputs.

By definition, the ideal functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$ do not clock their messages to Env or to each other. Hence, the Filter has to clock these in order to manage the timing of the resulting composed functionality. The previous description of the Filter is fairly straightforward for everything other than the collection mode. There can be a long exchange of messages between $\mathcal{F}_1$ and $\mathcal{F}_2$. The collection mode clocks each of the messages sent between these parties as this is part of the internal computation of $\mathcal{F}$. All the inputs received from Env as well as any outputs written to Env, are sent after the internal computations are done. This is in line with the ideal functionality specification that fixes that all outputs are computed as soon as the inputs are available. □

Corollary 1. *For all private ideal functionalities $\mathcal{F}_1$ and ideal functionalities $\mathcal{F}_2$ in the fully ordered composition $\mathcal{F}_1 \to \mathcal{F}_2$, the machine $\mathcal{F} = \mathcal{F}_1|\mathcal{F}_2$ is uniquely determined by the initial machines, a filter machine* Filter *affecting only the communication between $\mathcal{F}_2$ and* A *and a communication multiplexer* Mux.

Proof. Without loss of generality, we can assume the simplified fully ordered composition where all inputs are received by $\mathcal{F}_1$. From Lemma 2, it is known that the construction can be obtained with a filter between both ideal functionalities and A. However, note that private ideal functionality only communicates its inputs to A and these are forwarded by Filter. On the other hand, ${}_1\mathcal{F}$ never sends the outputs it gives to $\mathcal{F}_2$ to the adversary. Hence, it suffices to filter the messages from $\mathcal{F}_2$ only.

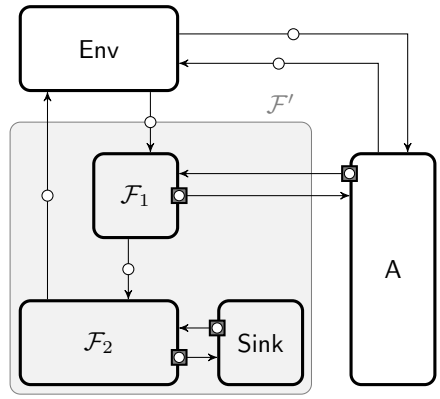


Figure 21: Decomposition of a composed ideal functionality for the ordered composition of two input-private protocols.

However, note that a multiplexer Mux is needed to join the messages from Filter and from $\mathcal{F}_1$ into the $\mathsf{out}_{\mathcal{F}}$ and to distribute the messages from $\mathsf{in}_{\mathcal{F}}$. Mux sends all messages from $\mathcal{F}_1$ and Filter directly to the adversary A. For all messages from A, it behaves like Filter in Lemma 2. The resulting configuration is shown in Figure 20. □

Lemma 3. *For all private ideal functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$ in the simple fully ordered composition $\mathcal{F}_1 \to \mathcal{F}_2$ and added* Sink*, the functionality $\mathcal{F}'$ with the open clocking port as shown in Figure 21 is as input-private as the private ideal functionality $\mathcal{F} = \mathcal{F}_1|\mathcal{F}_2$.*

Proof. The construction of $\mathcal{F}'$ is shown in Figure 21. The sink simply collects everything $\mathcal{F}_2$ intends to send to the adversary and does not do anything else. The control of the execution goes to the master scheduler after Sink finishes.

From Lemma 2, it is known that the composition F can be obtained with a Filter that only gives out the inputs received from the Env and the outputs sent to Env. In the simplified fully ordered composition, all inputs are received by $\mathcal{F}_1$. Hence, $\mathcal{F}_1$ also sends all input notifications to A as required. It does not send information about outputs as it is a private functionality. In the plain composition, $\mathcal{F}_2$ receives all the values from $\mathcal{F}_1$ and sends them to A as its inputs. Hence, Sink machine that simply receives these inputs from $\mathcal{F}_2$ and does nothing can be introduced. This is in line with the definition of the private ideal functionality as the adversary learns all the inputs and nothing else from $\mathcal{F}$ and it has the desired functionality.

Hence, the only difference between the functionalities $\mathcal{F}$ and $\mathcal{F}'$ is the clocking of the buffers from $\mathcal{F}_1$ to $\mathcal{F}_2$. In $\mathcal{F}$, there is no such clocking port for A. In order to show that the construction $\mathcal{F}'$ is as secure as $\mathcal{F}$, the adversary A against $\mathcal{F}'$ needs to be transformed to an equivalent adversary against $\mathcal{F}$. Note that the only observable effect of A clocking the buffers from $\mathcal{F}_1$ to $\mathcal{F}_2$ is the timing when $\mathcal{F}_2$ produces outputs. The simulator needs to simulate the buffers out from $\mathcal{F}$ so that

they would only get outputs of $\mathcal{F}_2$ once the inputs have been clocked to $\mathcal{F}_2$. As each input buffer to $\mathcal{F}_1$ has a distinct role, it is straightforward to keep track of when $\mathcal{F}_1$ would give outputs to $\mathcal{F}_2$. The simulator also simulates these buffers and allows the adversary against $\mathcal{F}'$ to clock them. The real outputs of $\mathcal{F}$ are clocked when the adversary against $\mathcal{F}'$ has clocked all inputs to produce it to $\mathcal{F}_2$ and then clocks the output buffer. □

As a consequence of Lemma 3 in order to show input privacy of ordered composition of $Sys_1 \rightarrow Sys_2$ with respect to $\mathcal{F} = \mathcal{F}_1|\mathcal{F}_2$, it suffices to show that the composition is as input-private as the construction $\mathcal{F}'$ in Figure 3. By transitivity, it is also then as input-private as $\mathcal{F}$.

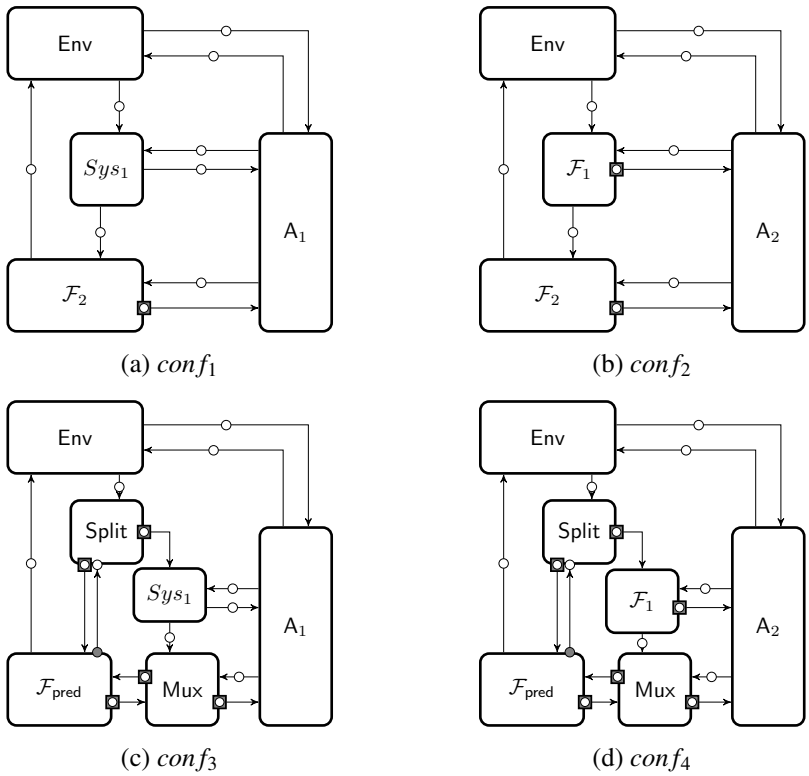
4.4. Output Predictability

As said before, input privacy is a very specific security notion that does not guarantee the correctness and where the outputs of the protocol may reveal unwanted information about the inputs. For example, a protocol where each party adds 1 to their share is input-private if each party computes it locally. However, if any party sees the output share, then it also knows what the input share for that party was.

Hence, using an input-private protocol in a composition can lead to unwanted leakages unless the structure of the composition and the other system are carefully chosen. The following focuses on an ordered composition $Sys_1 \rightarrow Sys_2$ where Sys_1 is an input-private protocol. Such composition can be insecure if Sys_2 somehow reveals its protected inputs to some party. Clearly, in the general case, such protocols and also the respective ideal functionalities $\mathcal{F}_2$ that reveal all inputs to the same party exist. Security of the ordered composition of an input-private and a secure system can only be achieved when we limit the ideal functionalities that define the properties of the secure system. Intuitively, we have to ensure that the ideal functionalities do not use the inputs from Sys_1 too explicitly. Such limitations of the ideal functionalities for MPC are described already in Section 3.4. The canonical ideal functionalities were restricted to compute fresh output shares of their output. The following property, called output predictability, gives a more general condition that has to be satisfied by the composition in order to ensure that it does not leak too much about the inputs. This enables to use the results regarding input privacy with more general secure functionalities than those described in Section 3.4.

Figure 22 depicts the configurations that define output predictability (Definition 27) for simplified fully ordered composition. Note that without loss of generality, Sys_2 gets all its inputs from Sys_1. It is always possible to enhance Sys_1 so that it simply forwards the extra inputs needed by Sys_2. Also, note that each buffer drawn in the configurations may be multiple buffers in the given direction. Hence, the limitation is that Sys_1 is in fully ordered composition with Sys_2 as $Sys_1 \rightarrow Sys_2$. Both systems may have several input and output ports to com-

municate. Note that system Sys_2 is as secure as some functionality $\mathcal{F}_2$. Hence definitions focus on $\mathcal{F}_2$. The configurations with Sys_1 and $\mathcal{F}_1$ are drawn side by side to highlight the similarities. Both versions are needed to define joint output predictability in Definition 28.



(a) $conf_1$ (b) $conf_2$

(c) $conf_3$ (d) $conf_4$

Figure 22: Configurations for defining joint output predictability for a simplified fully ordered composition $\mathcal{F}_1 \to \mathcal{F}_2$ or $Sys_1 \to Sys_2$.

In Figure 22, the configuration $conf_1$ shows the simplified fully ordered composition setup for $Sys_1 \to \mathcal{F}_2$ with adversary A_1. Configuration $conf_2$ is almost the same but Sys_1 is replaced with the respective ideal functionality $\mathcal{F}_1$, and hence there is a new adversary A_2 that interacts with $\mathcal{F}_1$. Configurations $conf_3$ and $conf_4$ create a very different context where Sys_1 or $\mathcal{F}_1$ is executed. In these configurations, the outputs are generated by an output predictor $\mathcal{F}_{\mathsf{pred}}$ instead of $\mathcal{F}_2$. In addition, $\mathcal{F}_{\mathsf{pred}}$ only sees the inputs of Sys_1 (or $\mathcal{F}_1$) and does not see the outputs of Sys_1 (or $\mathcal{F}_1$) that are inputs to $\mathcal{F}_2$ in $conf_1$ and $conf_2$. Split is a machine that gives all its inputs to both machines getting inputs from it. The multiplexer Mux, on the other hand, joins its inputs to unified outputs for A. The machines Split and Mux are included to ensure correct timing and views for the respective adversary so that it is not possible to distinguish $conf_3$ from $conf_1$ (or $conf_2$ from $conf_4$). These machines operate as defined in the following.

First, note that each input buffer that Sys_1 had is now input to Split, and re-

spectively, it has analogous buffers to pass the values to $\mathcal{F}_{\mathsf{pred}}$ and Sys_1 (or $\mathcal{F}_1$). The machine Split copies each of its input from Env to the respective buffers to both $\mathcal{F}_{\mathsf{pred}}$ and Sys_1 (or $\mathcal{F}_1$). It clocks the buffer to $\mathcal{F}_{\mathsf{pred}}$. The output predictor returns control to Split using the respective clocking buffer. On the signal from $\mathcal{F}_{\mathsf{pred}}$, Split clocks the relevant output buffer to Sys_1 (or $\mathcal{F}_1$). Hence, overall Split simultaneously passes its input to both of the receiving machines with the help of getting control back from $\mathcal{F}_{\mathsf{pred}}$. All these steps happen outside of adversarial control. The adversary has control either before or during clocking the inputs to Split and will next regain control when $Sys_1 \to \mathcal{F}_2$ (or $\mathcal{F}_1 \to \mathcal{F}_2$) yields it. Note that the configurations $conf_4$ and $conf_3$ have to make sure to yield the control at an equivalent time.

The machine Mux has two main goals: to pass the outputs of $\mathcal{F}_{\mathsf{pred}}$ to A with the right timing according to the execution of $Sys_1 \to \mathcal{F}_2$ (or $\mathcal{F}_1 \to \mathcal{F}_2$) and ensuring that $\mathcal{F}_{\mathsf{pred}}$ does not learn the outputs of Sys_1 (or $\mathcal{F}_1$). The multiplexer Mux keeps a list of corrupted parties C and a list E of messages m received from $\mathcal{F}_{\mathsf{pred}}$ or Sys_1 ($\mathcal{F}_1$) that it needs to send to A. Concretely, Mux works as follows.

- On input $(output, \ell_x, x)$ on behalf of $\mathcal{P}_i$ from Sys_1 or $\mathcal{F}_1$ it sends $(output, \ell_x, i)$ to $\mathcal{F}_{\mathsf{pred}}$ and clocks the buffer. Immediately after that, $\mathcal{F}_{\mathsf{pred}}$ gives control back to Mux. Mux sends the message $m = (input, \ell_x, x, i)$ to A if $\mathcal{P}_i$ is corrupted ($i \in C$) or otherwise stores m in E.
- On input $(corrupt, i)$ from A Mux adds i to C. It sends the corruption request to $\mathcal{F}_{\mathsf{pred}}$ that answers with the messages $(output, \ell_x, x, i)$. It writes all entries $(input, \ell_x, x, i) \in E$ and $(output, \ell_x, x, i)$ to the buffer to A. It clocks the buffer to A.
- On input $(output, \ell_x, x, i)$ from $\mathcal{F}_{\mathsf{pred}}$ it writes these values to the buffer for A and clocks the buffer.
- On input $(input, \ell_x, x, i)$ from $\mathcal{F}_{\mathsf{pred}}$ the machine Mux does nothing.

Note $\mathcal{F}_{\mathsf{pred}}$ needs to be defined based on the concrete functionalities. As an ideal functionality, it is most straightforward to assume that it produces both input and output notifications, but the input notification can also be discarded as it is not necessary for the definition of output predictability.

Note that the definition does not specify exactly how $\mathcal{F}_{\mathsf{pred}}$ works, but it has to support the clocking of the whole system. Rather, $\mathcal{F}_{\mathsf{pred}}$ is the machine that needs to be defined to prove output predictability. Furthermore, the output predictor can use the input signals from Split and Mux and its knowledge about $\mathcal{F}_2$ to provide outputs to Env at the same time that Env would receive them from $\mathcal{F}_2$ in $conf_1$ or $conf_2$. In general, $\mathcal{F}_{\mathsf{pred}}$ depends on Sys_1 as well as $\mathcal{F}_2$.

Definition 27 (Predictable outcome)**.** Ordered composition $Sys_1 \to \mathcal{F}_2$ has a predictable outcome if there exists a predictor machine $\mathcal{F}_{\mathsf{pred}}$ such that for the configurations in Figure 22 $view_{conf_1}(\mathsf{Env}) = view_{conf_3}(\mathsf{Env})$.

Definition 28 (Jointly predictable outcome)**.** The ordered compositions $Sys_1 \rightarrow \mathcal{F}_2$ and $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ have a jointly predictable outcome if there exits a common output predictor $\mathcal{F}_{\mathsf{pred}}$ such that for the configurations in Figure 22

$$view_{conf_1}(\mathsf{Env}) = view_{conf_3}(\mathsf{Env}) \text{ and}$$
$$view_{conf_2}(\mathsf{Env}) = view_{conf_4}(\mathsf{Env}) \ .$$

While the definitions of output predictability are uncommon, the property holds for many cases without the need to prove it separately. As said before, it is mostly designed to ensure that Sys_1 is correct with respect to $\mathcal{F}_1$ or the composition is such that the correctness does not affect the outputs of $\mathcal{F}_2$. Notably, the outputs of $\mathcal{F}_2$ for a canonical ideal functionality are independent of the randomness provided by the storage domain for the inputs of $\mathcal{F}_2$. Hence, it only cares about the value that is returned by Sys_1. Also, joint predictability is very simple if $\mathcal{F}_2$ ignores all inputs that it gets from Sys_1 or at least ignores the ones that are used or computed by Sys_1. Furthermore, for correct implementations that return the same outputs value (the value protected inside the storage domain) as the respective ideal functionalities, the two predictability properties are, in fact, equivalent. Note that the protection mechanism may be such that the ideal functionality, by definition, returns a fresh output, whereas the distribution given by the protocol may be different but can still be reconstructed to the same value. If the functionality is randomized then this can be generalized to the distribution of the shared values.

Lemma 4. *If Sys_1 is a correct implementation of the functionality $\mathcal{F}_1$ (meaning it returns the same protected output value), then $Sys_1 \rightarrow \mathcal{F}_2$ has a predictable outcome if and only if $Sys_1 \rightarrow \mathcal{F}_2$ and $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ have a jointly predictable outcome.*

Proof. Let $\mathcal{F}_{\mathsf{pred}}$ be the output predictor for $Sys_1 \rightarrow \mathcal{F}_2$. Assume, by contradiction, that it is not a suitable output predictor for $\mathcal{F}_1 \rightarrow \mathcal{F}_2$. Hence, if this machine is put into the respective configurations, then, by definition, $view_{conf_2}(\mathsf{Env}) \neq view_{conf_4}(\mathsf{Env})$ for at least some A_2 and Env. An adversary A_1 can then be constructed against the same environment where A_1 internally runs A_2 and passes all the messages from Sys_1 that $\mathcal{F}_1$ would generate to A_2. Since Sys_1 is a correct implementation of $\mathcal{F}_1$, the correct messages are used as inputs to A_2. In addition, A_1 can also input the respective messages from either Mux or $\mathcal{F}_2$ to A_2. It also sends everything A_2 sends to the Env and nothing else. The resulting configurations $conf_1$ and $conf_3$ are equivalent to the previous configurations $conf_2$ and $conf_4$, respectively. Consequently, $view_{conf_1}(\mathsf{Env}) \neq view_{conf_3}(\mathsf{Env})$ which is a contradiction with the assumption that $\mathcal{F}_{\mathsf{pred}}$ is the output predictor for the ordered composition $Sys_1 \rightarrow \mathcal{F}_2$. □

Often, a protocol is computing a deterministic functionality where the output is uniquely determined by the inputs, and the only randomisation is in the protection applied to the protocol outputs. If the function computed by Sys_1 and $\mathcal{F}_1$ is

deterministic, then the correctness of Sys_1 with respect to $\mathcal{F}_2$ is sufficient for joint output predictability. In this case, $\mathcal{F}_{\mathsf{pred}}$ gets the same inputs and can compute the same output value. If $\mathcal{F}_2$ does not use the randomness applied by Sys_1 to protect the output and only uses the output value, then it is easy to predict the outputs of $\mathcal{F}_2$. In the following, the joint output predictability is a strong property that is required to prove the security or input privacy of the ordered composition of the protocols. Note that output predictability covers correctness, but it is more general.

Output predictability may restrict some protocols that are randomised even if they are correct. For example, consider a protocol where party $\mathcal{P}_1$ generates a random bit and sends it to $\mathcal{P}_2$. $\mathcal{P}_1$ outputs this bit and $\mathcal{P}_2$ does not output anything. This protocol is a correct protocol for generating a random bit for $\mathcal{P}_1$. It is also trivially input-private since there are no inputs. However, it is not a secure protocol for randomness generation as the bit is leaked to $\mathcal{P}_2$. Hence, if such input-private protocol is composed with a protocol secure protocol $\mathcal{F}_2$ that randomises the output of $\mathcal{P}_1$, then it is important to consider output predictability. This protocol is not predictable since the adversary learns the correct output bit if it corrupts $\mathcal{P}_2$, but the predictor does not learn this bit. Hence, the values that the environment gets from $\mathcal{F}_{\mathsf{pred}}$ may not represent the same bit, as $\mathcal{F}_{\mathsf{pred}}$ does not know which bit was generated by Sys_1 as $\mathcal{F}_{\mathsf{pred}}$ only learns the inputs. It would learn the output of corrupted $\mathcal{P}_2$, but this party has no output. To generalise, if the functionality is randomised, then input privacy does not guarantee output predictability if the randomness is leaked to the adversary. For random protocols, also the randomised outputs must remain private or be public for everyone. If the outputs remain private, then it is a further task for the $\mathcal{F}_{\mathsf{pred}}$ to be able to simulate the same random distribution as produced by the outputs of Sys_1 if $\mathcal{F}_2$ uses the values it gets. Hence, while output predictability is simple for most cases, it is an important property to consider for non-deterministic protocols.

4.5. Black-Box Simulators

The main theorems for privacy composition work with black-box security and privacy definitions. The core idea is that such a security proof defines a simulator that acts as a mediator between the protocol and the adversary without interfering with the internal state of either system. This section specifies more details regarding simulators in the black-box proofs of privacy to later use these properties of the simulators in the composition theorems.

The goal is to first describe the idea of the simulator and then to define an extended simulator that can be used as a building block when proving security of a composition. As the ideal functionalities are defined as a single system in Section 4.3 then a proof strategy in the following also extends a simulator of a single system to a part of the simulator of a composed protocol. For that it is important to define inputs and outputs of simulators to be able to connect them.

4.5.1. Privacy Simulator

Let $Sys = (\mathfrak{M}, S)$ be the real structure and $Id = (\mathcal{F}, S)$ be the corresponding ideal privacy structure Id. The respective simulator $\mathsf{Sim}^{Id,Sys}$ has ports $\mathsf{in}_{\mathcal{F}}!$ and $\mathsf{out}_{\mathcal{F}}?$ for connecting to Id and a set of ports $\overline{S}$ from Sys to connect to the adversary running against Sys. This $\mathsf{Sim}^{Id,Sys}$ has the necessary ports of a black-box privacy simulator as it is between a real-world adversary and an ideal private functionality $\mathcal{F}$. The simulator is illustrated in Figure 23. The $\mathsf{net}_{i,Sys}$ denote all the connections to the buffers inside Sys that the adversary clocks. The buffers $\mathsf{in}_{i,Sys}$ and $\mathsf{out}_{i,Sys}$ denote the buffers that the adversary has to machine M_i in Sys.

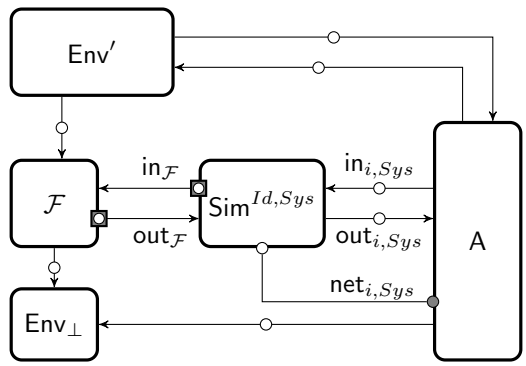


Figure 23: Simulator $\mathsf{Sim}^{Id,Sys}$ for the functionality $\mathcal{F}$ and the adversary A in the privacy configuration.

The buffer $\mathsf{in}_{\mathcal{F}}$ is clocked by the simulator and $\mathsf{out}_{\mathcal{F}}$ is clocked by $\mathcal{F}$ as usual. The set of adversary ports in the real system contains $\mathsf{out}_{i,Sys}!$ for each machine M_i in the system Sys. Generally, these buffers $\mathsf{out}_{i,Sys}$ are clocked by the adversary and are used by the system to give the adversary the information that it is supposed to see. Moreover, the adversary completely controls the execution timing of Sys, including the communication between M_i. Hence, one input clock signal to Sys affects only one machine M_i and, therefore, only $\mathsf{out}_{i,Sys}$ among the buffers to the adversary can be changed at a time. Therefore, the following assumes that, during each invocation of the simulator, $\mathsf{Sim}^{Id,Sys}$ writes to at most one $\mathsf{out}_{i,Sys}$ buffer at a time.

Note that because of the definition of an input-private ideal functionality, the simulator does not have the outputs of the computations. It only leans the inputs of the corrupted party from the ideal functionality. However, $\mathsf{Sim}^{Id,Sys}$ has to simulate the outputs as they are generated by Sys. The outputs include any public outputs of the protocol as well as any outputs made only to corrupted parties and the corrupted view of the secret outputs. Input privacy is only achievable if the outputs are either in some hiding storage domain or are computed so that the adversary knows the value. For example, the adversary knows the output value if the output is either a local computation from the inputs of the corrupted party or if the hiding property has been broken. If the hiding property is broken for

the inputs of the ideal functionality, then the adversary can learn the true protocol inputs from the inputs it gets from $\mathcal{F}$, and both the adversary and the simulator can compute the true output. The simulator always has to simulate the protocol outputs. The simulated outputs are correct if either the adversary view has the right distribution if the values are still private in some hiding storage domain, or if the adversary view contains the same value (or distribution) as computed by $\mathcal{F}$ if the output is not private any more. If the outputs are not correct, then it is easy for the adversary to distinguish the real interaction from the simulation. Hence, the privacy simulator has to also be able to simulate correct outputs.

A privacy definition considers configurations $conf_1 = (\mathfrak{M}, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, A)$ and $conf_2 = (\mathcal{F}, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{Sim}^{Id,Sys} \cup A)$. A simulator $\mathsf{Sim}^{Id,Sys}$ provides perfect privacy if $view_{conf_1}(\mathsf{Env}') = view_{conf_2}(\mathsf{Env}')$. The following will only consider simulators that provide perfect privacy as privacy simulators. However, in principle, the definition can be extended to simulators providing statistical or computational privacy using traditional variations of indistinguishability of the views.

4.5.2. Extended Privacy Simulator

The security of a composition can be proven in different ways, but the most straightforward is to base it on the security or input privacy of the components. A black-box security or input privacy proof defines a simulator like in the previous section. However, the privacy simulator in the previous section is defined as a stand-alone system, whereas the composition proofs need to combine several simulators into the simulator of the composed system. Hence, an extension of a privacy simulator is needed. Especially this extension must be able to produce the simulated outputs of the first system so that the simulator of the second component can use them as its input. Concretely, an extended privacy simulator is a simulator that, in addition to the main work of the simulator, also gives out the outputs computed for the corrupted parties. The privacy simulator has to exist for any input-private protocol. This section describes how a privacy simulator can be turned into a suitable extended simulator knowing the fact that the privacy simulator has to be able to simulate correct outputs.

When composing real systems Sys_1 and Sys_2, there can be several buffers between the two systems. These systems correspond to ideal functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$ such that the composed real system is as secure as (or as private as) the composed ideal systems. However, by ideal functionality composition definition (Definition 26), instead of the straightforward composition of ideal functionalities, the composition of $\mathcal{F}_1$ and $\mathcal{F}_2$ is a single machine $\mathcal{F}$. Hence, there are no equivalences to the buffers between the two real systems in the ideal composed system. Instead, these buffers have to be part of the simulation of the composed system when transforming a real adversary into the ideal adversary. These simulated buffers will be the ones used to carry the outputs of the simulation of Sys_1 to the simulator for Sys_2. In the following, these simulated buffers are denoted as $\mathtt{output}_i$. For a static

adversary, this is all that is needed. For an adaptive adversary, these need to be enhanced by fast output buffers $\mathsf{foutput}_i$, one for each party $\mathcal{P}_i$. The fast outputs are needed because the second component may need outputs from Sys_1 that have been computed and clocked before the corruption call. For example, if $\mathcal{P}_i$ becomes corrupted in the real world, then the first system reports all its inputs and outputs and the second system also reports all its inputs. As some of the inputs of the second system are the outputs of the first that, in the case of an ideal system, are generated by the simulator, then the composed simulators need to share this information. The buffers output_i are used to simulate messages that are sent in the real system, and the buffers $\mathsf{foutput}_j$ are used purely to enable the simulators to exchange information about previously computed outputs if needed. The exact details of their use are given in the following definition. Figure 24 gives the overall structure of the simulator.

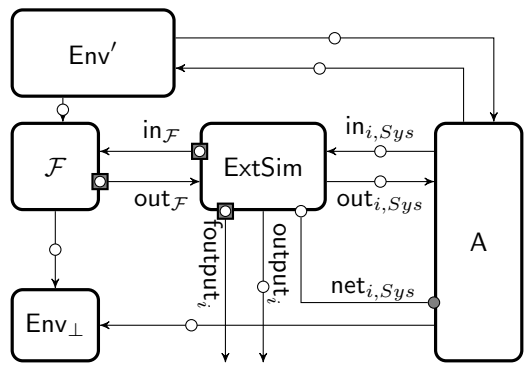


Figure 24: Extended privacy simulator $\mathsf{ExtSim}^{Id,Sys}$ for the functionality $\mathcal{F}$ and the adversary A in the privacy configuration so that ports $\mathsf{foutput}_i?$ and $\mathsf{output}_i?$ are open.

Definition 29 (Extended privacy simulator)**.** An extended privacy simulator (denoted $\mathsf{ExtSim}^{Id,Sys}$) for n-party ideal structure $Id = (\mathcal{F}, S)$ and a real structure $Sys = (\mathfrak{M}, S)$ is a machine with the following properties.

- It has the ports AP^{Sys} that the adversary expects in Sys as well as the ideal adversary ports $\mathsf{in}_{\mathcal{F}}!$ and $\mathsf{out}_{\mathcal{F}}?$ and the sets of ports $\{\mathsf{output}_1!, \ldots, \mathsf{output}_k!$ and $\mathsf{foutput}_1!, \ldots, \mathsf{foutput}_n!\}$ where output_i is for sending regular outputs, one port for every one of the k connections between Sys_1 and Sys_2 and $\mathsf{foutput}_i$ is for outputs of $\mathcal{P}_i$ that need to be fast-forwarded due to late corruption for the case of an adaptive adversary.
- In case of a corruption request to corrupt party $\mathcal{P}_i$ coming from $\mathsf{in}_{i,Sys}?$ comes after some $(output, \ell_x)$ has been sent on $\mathsf{out}_{i,Sys}$ then $\mathsf{ExtSim}^{Id,Sys}$ forwards the corruption request to $\mathcal{F}$ and receives the input of i-th party from $\mathcal{F}$. Upon learning the input, $\mathsf{ExtSim}^{Id,Sys}$ computes all the previously ready outputs x and immediately makes an output $(output, \ell_x, x)$ for each message where the notification ℓ_x was sent. These outputs are written on

$\mathsf{foutput}_i$ and this buffer is clocked. This is the only time when anything is written to $\mathsf{foutput}_i$.

Note Note that at most one party gets corrupted per a corruption request, and therefore, there are new messages in at most one $\mathsf{foutput}_i$ so that it can be clocked by the extended simulator.

- It is a privacy simulator when $\mathsf{foutput}_i$ and output_i buffers are ignored. More concretely, consider configurations $conf_1 = (\mathfrak{M}, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{A} \cup \mathsf{Sink}')$ and $conf_2 = (\mathcal{F}, S, \mathsf{Env}' \cup \mathsf{Env}_\perp, \mathsf{ExtSim}^{Id,Sys} \cup \mathsf{A} \cup \mathsf{Sink})$ with the coinciding environment views $view_{conf_1}(\mathsf{Env}') = view_{conf_2}(\mathsf{Env}')$ where:
  - Sink is a machine with ports $\mathsf{output}_1?, \mathsf{foutput}_1?$ that does not act on any input.
  - Sink' has both $\mathsf{output}_i!$ and $\mathsf{output}_i?$ ports and does nothing. It simply exists to enable the output_i buffers.
- It correctly computes outputs $\mathcal{O}_{conf_1} = \mathcal{O}'_{conf_2}$ for the same $conf_1$, $conf_2$, Env and A as defined for the privacy property. The outputs of the protocol are defined as follows.
  - Let τ be the trace of one concrete protocol execution recording which messages were sent by parties on which buffers and in which order. Then define $\mathcal{O}(\tau)$ to be the list of protocol outputs observed by the adversary A in the real system. If there was a corruption request for $\mathcal{P}_i$, then the output contains all messages written to out_i. If the party was not corrupted, then there are no messages corresponding to that party in $\mathcal{O}(\tau)$.
  - Let $\mathcal{O}'(\tau)$ be the list of outputs generated by $\mathsf{ExtSim}^{Id,Sys}$ and written to output_i or $\mathsf{foutput}_\mathsf{i}$ for corrupted $\mathcal{P}_i$.
  - Let $\mathcal{O}_{conf}$ and $\mathcal{O}'_{conf}$ respectively be the distributions of $\mathcal{O}(\tau)$ and $\mathcal{O}'(\tau)$ over all possible runs of the configuration.

The machine Sink can be removed from the extended simulator during composition, and instead, these ports will be connected to a multiplexer that gives inputs to the next simulator. The multiplexer can then merge inputs from the previous simulators and the ideal functionalities as necessary.

Similarly to Sink, Sink' is also a dummy machine used to introduce the output_i buffers to the real system to allow the adversary to clock them. Note that the initial real-world adversary does not have access to this buffer. However, the adversaries created in the composition may need to clock these buffers. Later these are used in constructions where output is clocked synchronously with the outputs of $\mathcal{F}$.

Lemma 5. *If there exists a perfect privacy simulator* $\mathsf{Sim}^{Id,Sys}$ *for an ideal structure Id and real structure Sys, then there also exists an extended simulator* $\mathsf{ExtSim}^{Id,Sys}$ *for the same structures.*

Proof. This proof constructs an extended simulator from the simulator and some additional simple machines, as shown in Figure 25. The buffers $\mathsf{net}_{i,Sys}$ and $\mathsf{net}_{i,out}$

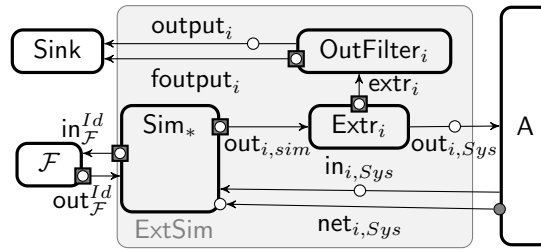


Figure 25: The construction of ExtSim from privacy simulator Sim connected to the ideal functionality $\mathcal{F}$, the adversary A and Sink.

denote the usual buffers that the real adversary A can clock inside the real system or as outputs of the system. The respective ports belong to the set AP^{Sys} of ports that the extended simulator has.

First, let $\mathsf{Sim}_*^{Id,Sys}$ be a version of Sim with ports $\mathsf{out}_{i,Sys}!$ renamed to $\mathsf{out}_{i,sim}!$ and added functionality to make $\mathsf{out}_{i,sim}$ self-clocked. The $\mathsf{out}_{i,sim}$ is clocked whenever a message is written to the buffer. Note that since at most one party is making the transition in the real system, then also the simulator needs to write at most one $\mathsf{out}_{i,sim}$ in response to one adversary action.

In order to to have the right structure with the right output buffers, the extractor machines Extr_i with ports $\mathsf{out}_{i,sim}?$, $\mathsf{extr}_i!$ and $\mathsf{out}_{i,Sys}!$ are needed. Hence, the machine mediates between the Sim and A and has the additional port to forward the outputs as necessary. Whenever Extr_i receives something from $\mathsf{out}_{i,sim}?$ it copies the message to $\mathsf{out}_{i,Sys}!$. In addition, if the message is labelled as output in format $(output, \ldots)$, then this message is also copied to $\mathsf{extr}_i!$ and then Extr_i also clocks extr_i.

$\mathsf{OutFilter}_i$ is a machine that filters outputs that the simulator gives in time and separates them from the messages that the simulator has already computed but needs to send after a new corruption request. Messages from buffer extr_i are received by $\mathsf{OutFilter}_i$. In addition to port $\mathsf{extr}_i?$ this machine also has ports $\mathsf{output}_i!$ and $\mathsf{foutput}_i!$ that are the output ports of the extended simulator. $\mathsf{OutFilter}_i$ may receive two types of output messages. If the party i is honest, then it just receives $(output, \ell_x)$ where ℓ_x is the label of the output. However, for corrupted parties, it receives $(output, \ell_x, x)$ where x is the value of the actual output. The goal of the output filter is to distinguish if something is a new output or an output that has to be released when a party is corrupted after the output has been computed. Toward that goal, the machine keeps a list $\mathcal{L}_{i,out}$ of all the labels it has seen and upon all inputs, it adds ℓ_x to $\mathcal{L}_{i,out}$. Messages $(output, \ell_x)$ are always written to $\mathsf{output}_i!$. Messages $(output, \ell_x, x)$ from corrupted parties are processed depending whether ℓ_x is in $\mathcal{L}_{i,out}$.

- If $\ell_x \in \mathcal{L}_{i,out}$ then $(output, \ell_x, x)$ is written to $\mathsf{foutput}_i!$ and the buffer is clocked.

  Note Note that the real protocol sends notifications to the adversary about all the outputs. In this case, there has been a message $(output, \ell_x)$

before to denote that this output has been computed. The new message appears if the party is newly corrupted and needs to send some previously computed outputs. It is needed to deliver the content x.

- If $\ell_x \notin \mathcal{L}_{i,out}$ then $(output, \ell_x, x)$ is written to $\mathsf{output}_i!$.

The extended simulator ExtSim is a composition of the described machines as illustrated in Figure 25. It satisfies the structural properties of an extended simulator on the ports and clocking buffers. The extended simulator provides privacy as it forwards all the same values as the perfect privacy simulator and the added values on the output_i and $\mathsf{foutput}_i$ buffers do not affect the view of the Env' or A. The outputs to output_i and $\mathsf{foutput}_i$ are computed correctly as the Sim simulates them correctly.

The behaviour and distinction between output_i and $\mathsf{foutput}_i$ is also kept as required by the extended simulator definition. All outputs that have not yet appeared are written to output_i and outputs that have appeared as honest parties labels are written to $\mathsf{foutput}_i$ when they are sent again by Extr_i. Note that the immediate return is ensured by the properties of the functionality and the simulator Sim. When a corruption request is clocked to Sim, it is clocked to $\mathcal{F}$, and the input values of the corrupted party are returned and clocked by $\mathcal{F}$. The simulator also has to then create the view of the newly corrupted party and send it out so that it is received by Extr that can then extract the output and write it to $\mathsf{extr}_i!$. Since the output was already computed, then $\mathsf{OutFilter}_i$ has received $(output, \ell_x)$ and therefore, the newly received value is immediately clocked out of $\mathsf{foutput}_i$. In the current configuration, the Sink then absorbs the message and the control is handed back to the master scheduler. Hence, from the scheduler's viewpoint, the return of the suitable values is immediate and only the required values are ever written to $\mathsf{foutput}_i$. □

4.6. Privacy of the Composition of Private Protocols

This section studies the ordered composition of input-private protocols and establishes that the notion of input privacy is composable for the black-box version of the definition. It is sufficient to consider only fully ordered or simple fully ordered composition when discussing the ordered composition of input-private systems.

Lemma 6. *The ordered composition of input-private protocols is input-private if simplified fully ordered composition of input-private protocols is input-private.*

Proof. Firstly, note that the composition of two input-private systems that do not communicate with each other is always input-private. In terms of the input privacy definition, one system can be merged into the environment, and then the other is in the simple input privacy configuration.

Secondly, note that a simple functionality of a communication channel that does no modifications to the inputs is always input-private. This functionality

simply receives the inputs and forwards them to the recipients. It has no computations and the adversary can only see the corrupted inputs, which ensures the hiding property. This can be seen as a local identity or copying functionality, where the adversary can only see the corrupted party's view and no new information is revealed in this protocol. Hence, one can first find a topological order for all systems in an ordered composition. Then, all systems can be extended to copy all outputs of the previous functionalities as their outputs. The composition of extended functionalities is a simplified fully ordered composition. □

Theorem 5 (Theorem 4 formally)**.** *Let* $Sys_1 \geq_{priv}^{model} \mathcal{F}_1$ *and* $Sys_2 \geq_{priv}^{model} \mathcal{F}_2$*, both with black-box privacy. The model may be perfect, statistical or computational as needed. Then, for ordered composition* $Sys_1 \to Sys_2$ *with a coherent adversary,* $Sys_1 \to Sys_2 \geq_{priv}^{model,\mathbb{A}_c} \mathcal{F}$ *with black-box simulation, where* $\mathcal{F}$ *is the ideal composition of* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *as in Definition 26.*

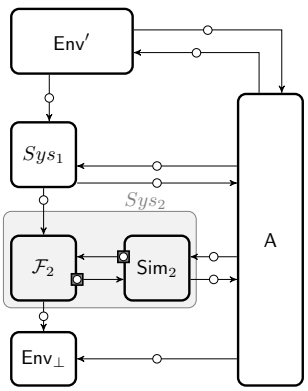
Proof. Firstly, based on the observation in Lemma 6 regarding the privacy of non-communicating systems, the statement can be proved only for the simple fully ordered case. For the general case, Sys_1 can always be extended with the private channels to forward the inputs of Sys_2 and Sys_2 can forward the inputs from Sys_1 as outputs.

This proof is illustrated by Figure 26, where a simulator is derived for the real fully ordered composition for the ideal composed system $\mathcal{F}$. Figure 26a pictures the original composition of the two real systems $Sys_1 \to Sys_2$ ($conf_0$) with Sys_2 in grey as well as the version where Sys_2 has been substituted with the equivalent combination of the $\mathcal{F}_2$ and the simulator Sim_2 from the black-box privacy definition ($conf_1$). This step is allowed, as Sys_1 is defined as part of the Env' for the Sys_2 privacy configuration. Since the simulator has to exist for all Env and A, then such Sim_2 exists since the protocol has black-box privacy. Note that, in $conf_1$, Sys_1 is not in a privacy configuration as its outputs are leaked to the adversary as inputs of Sys_2, and hence analogous substitution cannot be done with Sys_1.

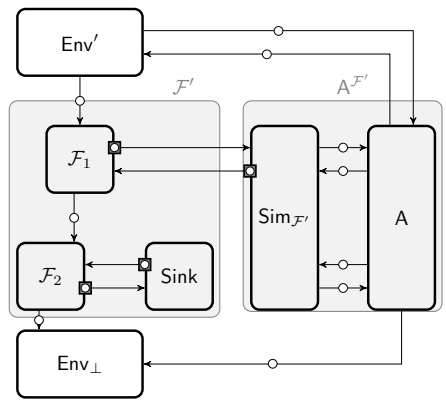
In order to prove input privacy, the real system must be indistinguishable from the respective ideal functionality $\mathcal{F}$. However, applying Lemma 3 specifies another functionality likes $\mathcal{F}$ denoted as $\mathcal{F}'$ but the adversary clocks the buffers from $\mathcal{F}_1$ to $\mathcal{F}_2$. It suffices to prove input privacy with respect to that functionality to derive that the real system is as input private as $\mathcal{F}$. Figure 26b demonstrates the functionality $\mathcal{F}'$ in the privacy configuration with the adversary $\mathsf{A}^{\mathcal{F}'}$ in gray as $conf_0'$. The goal of the input privacy proof is to show that there exists $\mathsf{Sim}_{\mathcal{F}'}$ such that $\mathsf{A}^{\mathcal{F}'} = \mathsf{Sim}_{\mathcal{F}'} \cup \mathsf{A}$ for any real adversary A. In order to simplify the following construction of the simulator, define $conf_1'$ based on the construction for the composed ideal functionality with an extra clocking port from Lemma 3. Note that, in fact, $\mathcal{F}_1$ is now in a privacy configuration as $\mathcal{F}_2$ does not communicate with anyone except Sink and $\mathsf{Env}_\perp$ so these can be merged to $\mathsf{Env}_{*\perp}$.

Finally, Figure 26c proposes the simulator construction for $\mathsf{Sim}_{\mathcal{F}}$ using ExtSim (Definition 29) constructed from the privacy simulator Sim_1 for the first system. It

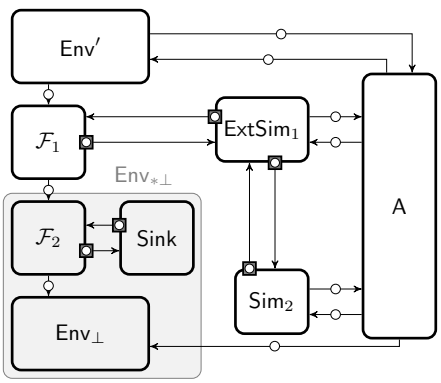
remains to argue that the view of A in $conf_2'$ and $conf_1$ is the same. In both cases, one part of the interaction is with Sim_2, however, Sim_2 gets its inputs from different sources. Note that ExtSim contains Sim_1 and forwards the outputs generated in the simulation in Sim_1 as inputs to Sim_2 and Sim_2 does not expect any other inputs. By definition, Sim_1 produces a view that is indistinguishable from Sys_1 running. Hence, the adversary cannot distinguish this part from Sys_1. However, this also means that the outputs extracted from the outputs of Sim_1 are equivalent to those generated by Sys_1 and, therefore, Sim_2 gets equivalent outputs in the two worlds. □



(a) Two ordered systems in privacy configuration, $conf_0$ (with Sys_2 in gray) and $conf_1$.



(b) Composed ideal functionality in the privacy configuration, $conf_0'$ (with $\mathcal{F}$ and $\mathcal{A}^{\mathcal{F}}$ in gray) and $conf_1'$.



(c) Simulator construction for the deconstructed ideal functionality, $conf_2'$.

Figure 26: Configurations to build the simulator for the fully ordered composition of two real input-private systems.

Corollary 2 (Composition of input privacy for general adversaries)**.** *Let* $Sys_1 \geq_{priv}^{model} \mathcal{F}_1$ *and* $Sys_2 \geq_{priv}^{model} \mathcal{F}_2$ *both with black-box security. The model may be perfect, statistical or computational as needed. Let the ordered compositions* $Sys_1 \to \mathcal{F}_2$

and $\mathcal{F}_1 \to \mathcal{F}_2$ have a jointly predictable outcome. Then, for fully ordered composition $Sys_1 \to Sys_2$, $Sys_1 \to Sys_2 \geq_{priv}^{model} \mathcal{F}$ with black-box simulation where $\mathcal{F}$ is the ideal composition of $\mathcal{F}_1$ and $\mathcal{F}_2$ as in Definition 26.

Proof. A direct result from Theorem 5 and Lemma 1. □

Structural induction can be used to extend Corollary 2 to the ordered composition of two or more systems. Joining this result with that of Corollary 3 allows us to conclude that any number of input-private systems can be composed with a secure system to obtain a secure protocol.

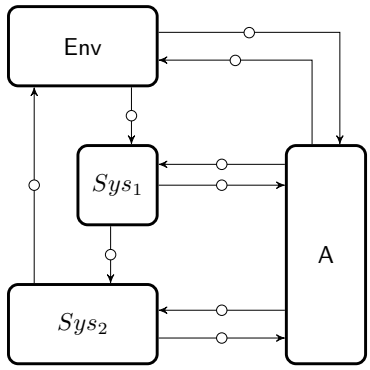
Often, the simplest secure functionality is that of rerandomising the secure outputs of the private system. For example, refreshing the encryption or sharing randomness. However, most of the computation protocols can be composed of input-private systems and the secure functionality can be pushed to the end of the computation. In the context of arithmetic circuits, where there often are many intermediate values and a small number of outputs, this is especially valuable since most of the components can be input-private protocols that can be more efficient than secure protocols. The more expensive secure computations are only needed for the potentially small number of outputs. A concrete example of this is discussed in Section 4.8.1.

4.7. Security of Ordered Composition of Private and Secure Protocols

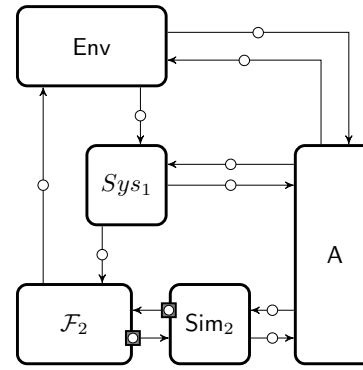
This section formalises the main security theorem for composing private and secure protocols. Since the proof is quite detailed, it is separated into several independent lemmas. This section uses the class of coherent adversaries $\mathbb{A}_c$ (Definition 19) that always corrupt all machines representing the same party simultaneously. The class of coherent adversaries is equivalent to the generic adversaries as shown in Lemma 1. In principle, the coherent adversary corrupts all machines representing one party together, but in practice, independent corruption requests still have to reach each machine. Without loss of generality, we assume that for ordered composition $Sys_1 \to Sys_2$, the party in Sys_1 is corrupted first, as this simplifies the following proofs.

Theorem 6 (Theorem 3 formally)**.** *Let $Sys_1 \geq_{priv}^{model} \mathcal{F}_1$ and $Sys_2 \geq_{sec}^{model} \mathcal{F}_2$, both with black-box security against coherent adversaries. The model may be perfect, statistical or computational as needed. Let the ordered compositions $Sys_1 \to \mathcal{F}_2$ and $\mathcal{F}_1 \to \mathcal{F}_2$ have a jointly predictable outcome. Then, for a fully ordered composition $Sys_1 \to Sys_2$ with a coherent adversary, $Sys_1 \to Sys_2 \geq_{sec}^{model,\mathbb{A}_c} \mathcal{F}$ with black-box simulation where $\mathcal{F}$ is the ideal composition of $\mathcal{F}_1$ and $\mathcal{F}_2$ (as in Definition 26).*

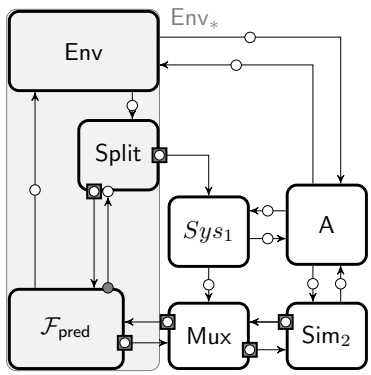
Proof. Similarly to Theorem 5, it is suitable to only consider simple fully ordered composition. In the latter, all inputs of Sys_2 must come from Sys_1 and Sys_1 is an



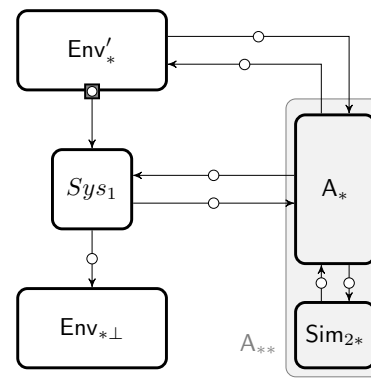
(a) Fully ordered composition of two systems, $conf_0$.



(b) Ordered composition of a real and ideal system, $conf_1$.



(c) Using output predictability and regrouping the machines, $conf_2$ and $conf_3$.



(d) Splitting Mux and redefining the adversary, $conf_4$ and $conf_5$.

Figure 27: Configurations to simplify the ordered composition of two real systems.

input-private system. Hence, analogous to the logic in Lemma 6, it is possible to extend any input-private system with a copying functionality that copies all inputs that are given from Env to Sys_2. This simplifies the structure of the following proof, and therefore, it is beneficial to consider simple fully ordered composition.

This proof firstly transforms the real ordered composition as in Figure 27. Secondly, the proof shows a candidate simulator construction in Figure 28 and transforms it using the properties of its components. The final goal is to show that the real system in Figure 27a and the ideal system in Figure 28a are equivalent to their respective transformations and that it is easy to see that Figure 27d and Figure 28e are equivalent.

Transformation of the real system. Figure 27a illustrates the ordered composition of the two original systems. We'll define this as $conf_0$. The black-box security of Sys_2 allows to use the RSIM composition result in Theorem 2 to replace Sys_2 with the ideal functionality $\mathcal{F}_2$ and a simulator Sim_2 to be $conf_1$

such that $view_{conf_0}(\mathsf{Env}) \approx view_{conf_1}(\mathsf{Env})$. In the perfect case, $view_{conf_0}(\mathsf{Env}) = view_{conf_1}(\mathsf{Env})$. The resulting configuration $conf_1$ is given in Figure 27b. Note that A is always the master scheduler, and, in the simulated case, Sim_2 becomes the scheduler. Therefore, without loss of generality, we can consider that Sim_2 clocks the buffer to $\mathcal{F}_2$.

As a second step, the output predictability (Definition 28 and Figure 22) of $Sys_1 \to \mathcal{F}_2$ is applied to separate Sys_1 and Sys_2. For that, consider Sim_2 in $conf_1$ as part of the adversary in the predictability definition and obtain $conf_2$ as in Figure 27c. Predictability tells that $view_{conf_1}(\mathsf{Env}) = view_{conf_2}(\mathsf{Env})$. In addition, joint output predictability defines that the same predictor $\mathcal{F}_{\mathsf{pred}}$ can be used to predict the outcome of the composition of the ideal functionalities $\mathcal{F}_1 \to \mathcal{F}_2$.

Define a new machine Env_* as $\mathsf{Env}_* = \mathsf{Env} \cup \mathcal{F}_{\mathsf{pred}} \cup \mathsf{Split}$ and define the resulting simplified configuration as $conf_3$. This serves as the new environment that gives inputs to Sys_1 and Mux. Such transformation is allowed as the environment and its communication with the adversary A is not limited. Configuration $conf_3$ is depicted of Figure 27c with Env_* in gray. Note that, in the following, whenever the view is indistinguishable for Env_*, it is also indistinguishable for Env, which is a sub-machine of the new environment and does not get any extra information from elsewhere.

Details of Mux are defined in Definition 28, its main role is to pass the outputs of $\mathcal{F}_{\mathsf{pred}}$ to A and to let it know when Sys_1 has computed an output. As the next step, the Mux is split into two as it serves two purposes in the current configuration. This split gives a new configuration $conf_4$ in Figure 27d with $view_{conf_3}(\mathsf{Env}_*) = view_{conf_4}(\mathsf{Env}_*)$. The machine Mux is broken into two parts, the part that ignores outputs of Sys_1 becomes $\mathsf{Env}_{*\perp}$. In $conf_4$, all outputs of Sys_1 are sent to $\mathsf{Env}_{*\perp}$. The remaining part only communicates with A (through Sim_2) and $\mathcal{F}_{\mathsf{pred}}$. The new adversary A_* is like a joint version of A and Mux that only sees the corrupted outputs. The simulator Sim_{2*} is like Sim_2 just connected to the part of the Mux that is now internal to A_*. The full details for this split and A_* in case of an adaptive adversary will be shown in Lemma 7.

Similarly to redefining the environment, A_* is redefined to merge Sim_2^* to obtain configuration $conf_5$ as in Figure 27d. As the view of A inside A_* or A_{**} remains the same and the rest of the configuration affecting Env_* does not change, it is straightforward that $view_{conf_4}(\mathsf{Env}_*) = view_{conf_5}(\mathsf{Env}_*)$.]

Transformation of the ideal system and proposed simulator. With the previous transformations, the composition $Sys_1 \to Sys_2$ is in a simplified form and it is time to look at $\mathcal{F}$. The main claim of this theorem is that the combination of ExtSim_1 for $\mathcal{F}_1$, a multiplexer Mux_{Sim} and a simulator Sim_2 for $\mathcal{F}_2$ form a suitable simulator to turn any real-world adversary A to an equivalent adversary against $\mathcal{F}$ as shown in Figure 28b. Note that the simulator construction contains a demultiplexer that splits the buffers joined by Mux in $conf_1'$.

The ideal world transformation is shown in Figure 28. Let the original configuration with $\mathcal{F}$ be $conf_0'$ as in Figure 28a. The adversary can send corruption

requests, and the ideal functionality $\mathcal{F}$ sends messages as specified in Definition 25. Applying Corollary 1 gives a possibility to decompose $\mathcal{F}$ as $\mathcal{F}_1$, $\mathcal{F}_2$ and Filter. Let this be $conf_1'$ in Figure 28a. In the ordered ideal composition of ideal functionalities, the Filter is defined in Lemma 2 and the Mux is added in Corollary 1. The Filter only gives out the outputs of $\mathcal{F}_2$ to A and stops the inputs. The Filter also clocks the channel from $\mathcal{F}_1$ to $\mathcal{F}_2$ when it learns that $\mathcal{F}_1$ has written an output. The multiplexer forwards the inputs for $\mathcal{F}_1$ and the outputs of $\mathcal{F}_2$ to the adversary. It also lets Filter know if $\mathcal{F}_1$ has produced an output. Mux also forwards the corruption requests sent by A to the functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$. Note that in such case, the functionality sends a response to the corruption request and Mux always gets the control back.

Next, the ideal adversary $\mathsf{A}^{\mathcal{F}}$ is replaced with the combination of the proposed simulator $\mathsf{Sim}_{\mathcal{F}}$ and a real adversary A. The proposed construction for $\mathsf{Sim}_{\mathcal{F}}$ appears in $conf_2'$ in Figure 28b. In essence, there is a demultiplexer in the simulator construction that takes the messages from Mux and separates these into messages from $\mathcal{F}_1$ and Filter. In $conf_2'$, the multiplexer Mux and demultiplexer are cancelled out and replaced by simply keeping the respective buffers. ExtSim is built as in Lemma 5 from Sim_1. Note that, from the preconditions of the theorem, it is known that there exists a black-box perfect privacy simulator Sim_1 for $\mathcal{F}_1$ and a black-box security simulator Sim_2 for $\mathcal{F}_2$. The multiplexer Mux_{Sim} is akin to the multiplexer used in the predictable outcome definition (Definition 28). Here Mux_{Sim} works as follows:

- On input $(output, \ell_x, x)$ from ExtSim (on either output_i or $\mathsf{foutput}_i$) it sends it to Filter and clocks the buffer. The Filter clocks the respective buffer between $\mathcal{F}_1$ and $\mathcal{F}_2$ if the message was from output, it gets control back from $\mathcal{F}_2$ and then gives the control back to Mux_{Sim}. Finally, Mux_{Sim} sends the message $(input, \ell_x, x)$ to the buffer to Sim_2 if the party $\mathcal{P}_i$ is corrupted.
- On input $(corrupt, i)$ from Sim_2 who forwards the message from A, Mux_{Sim} adds i to the list of corrupted parties C. It then writes the corruption request to Filter and clocks the buffer, and gets control back with the response message $(output, \ell_x, x, i)$ if any outputs have been computed. It then sends all entries $(input, \ell_x, x, i)$ and $(output, \ell_x, x, i)$ to Sim_2 as one message in response to the corruption request. Note that the adversary is coherent and has corrupted Sys_1 before Sys_2. Hence ExtSim has already processed the corruption request and the inputs are available.
- On input $(output, \ell_x, x, i)$ from Filter it sends these to Sim_2 unaltered.

  Note Input $(input, \ell_x, x, i)$ is never received from Filter as the purpose of the filter in the construction of $\mathcal{F}$ is to only send out the outputs of $\mathcal{F}_2$ and to filter out and never send the inputs.
- Any unspecified message is ignored.

Hence, compared to the multiplexer in the predictability definition, Filter has the same role as $\mathcal{F}_{\mathsf{pred}}$, ExtSim is in the role of Sys_1 (of $\mathcal{F}_1$) and Sim_2 is in the role of

the adversary.

Most of the buffers in the simulator construction are sender clocked. However, the output_i buffers from ExtSim to Mux_{Sim} is a new buffer clocked by A. The adversary A is the real-world adversary that treats these buffers as it would the buffers from Sys_1 to Sys_2. If the adversary clocks this buffer, then the Filter receives an input and can clock the buffer from $\mathcal{F}_1$ to $\mathcal{F}_2$. This clocking capability of A was not obvious in $conf'_1$, but it is a buffer that A expects to use to control the timing of the execution.

Next, consider A, ExtSim, Mux_{Sim}, Filter and Sim_2 as an adversary in the output predictability definition to obtain a configuration $conf'_3$ in Figure 28c that is analogous to previous $conf_2$. This configuration introduces $\mathcal{F}_{\mathsf{pred}}$, Mux and Split machines from Definition 28. The timing of the execution is also specified in the output predictability definition Definition 28.

Next, configuration $conf'_4$ is derived by a similar simplification as before where $\mathcal{F}_{\mathsf{pred}}$ and Split are pushed to Env to obtain Env_*. This collection is shown in Figure 28c together with $conf'_3$.

In order to derive the next configuration $conf'_5$, the different flows that may occur in the given configuration need to be analysed. The details of this substitution are discussed in Lemma 8. The end result is a configuration where the Mux, Mux_{Sim} and Filter are reconfigured together with separating the environment $\mathsf{Env}_* = \mathsf{Env}'_* \cup \mathsf{Env}_{*\perp}$ as in Figure 28d. This configuration is similar to that of $conf_3$ on Figure 27c where Mux from $conf_3$ is represented as Mux_* and $\mathsf{Env}_{*\perp}$. Mux_* is working mostly as Mux_{Sim} with the addition that it notifies $\mathcal{F}_{\mathsf{pred}}$ in Env'_* when outputs are received from ExtSim.

Next, note that the extended simulator has been built from Sim_1 according to Lemma 5. Hence, it can be decomposed to its parts Sim_1, OutFilter and Extr as configuration $conf'_6$. Further, consider a new adversary A_{**} created by merging A with Extr, OutFilter, Sim_2 and Mux_* to arrive to the final configuration $conf'_7$ depicted in Figure 28e together with $conf'_6$.

All these changes have preserved the view of the environment, hence for the initial and final configuration, $view_{conf'_0}(\mathsf{Env}) \approx view_{conf'_7}(\mathsf{Env})$ as Env is a submachine of Env'_*.

Now, putting together the simplifications of the real and ideal composition leaves us with $conf_5$ and $conf'_7$ where the Env_* is, in fact, the same, if both transformations start from the same Env. In both cases, Env_* contains Env, Split, and $\mathcal{F}_{\mathsf{pred}}$ where the added machines are the same based on the joint output predictability (Definition 28). Similarly, the adversaries A_{**} are the same in the two configurations. Note that $conf'_7$ simply also draws the clocking connection, but the output of $\mathcal{F}_1$ is also clocked by A_{**} in $conf_5$. By definition, the machine Sim_2 is the same in $conf_5$ and $conf_2$, and the multiplexers Mux and Mux_* have the same function. In addition, Lemma 7 shows that the step to $conf_4$ can be done with introducing Extr and OutFilter hence they were already in A_* in $conf_4$.

Hence, this proof has shown that the question of whether $Sys_1 \rightarrow Sys_2$ is as

secure as $\mathcal{F}$ reduces to the question of whether $conf_5$ is indistinguishable from $conf_7'$. Note that the final configurations depict the privacy definition configuration where the outputs are not returned. By definition, Sys_1 is as private as $\mathcal{F}_1$ in a black-box manner and Sim_1 is the simulator that proves the input privacy. Hence, by definition, for every adversary A_{**}, one can use the simulator Sim_1 to turn it into an equivalent adversary against the input privacy of the ideal functionality $\mathcal{F}_1$. Hence, $view_{conf_5}(\mathsf{Env}) \approx view_{conf_7'}(\mathsf{Env})$ and the claim of the theorem follows. □

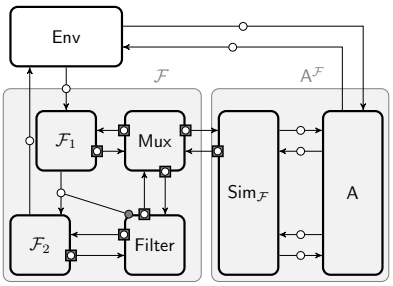
Lemma 7. *For a coherent adversary* A, $view_{conf_3}(\mathsf{Env}_*) = view_{conf_4}(\mathsf{Env}_*)$.

Proof. This lemma specifies a transformation step not fully detailed in Theorem 6 and refers to configurations in Figure 27. The respective theorem needs the equivalence of the views of the environment. However, this transformation is really focused on the adversary. If the view of the adversary A remains the same and the construction for A_{**} is a valid adversary, then the resulting configuration $conf_4$ is equivalent to the $conf_3$. The configurations in question are shown in Figure 29 in detail and belong to Figure 27 in the bigger picture. The proof starts from $conf_3$ with a coherent adversary A. A coherent adversary means that the party is either corrupted in both or neither of the systems. Hence, the adversary A can see all inputs of Sim_2 in $conf_3$ as the corrupted parties' outputs from Sys_1. Note that Sim_2 also learns only the shares of the corrupted parties.

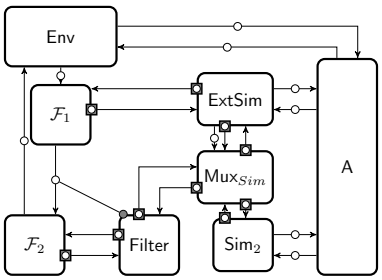
The machine Mux is defined in the predictable output definition. In the current configuration, Mux can be split based on the corruption. The part that works with the values of the corrupted parties can be merged to A to form A_{**}. The part ignoring the other outputs forms $\mathsf{Env}_{*\perp}$.

The construction to split Mux could be achieved by simple rewiring for static corruption. For adaptive corruption, the values seen by Sim_2 must change if the set of corrupted parties changes. However, this can still be managed by the new adversary A_{**}, who is in charge of the corruption. The idea of the following construction is to get the corrupted outputs of Sys_1 from the buffer where they are sent to A and rewire the rest so that the simulator Sim_2 uses these instead of the values from Mux.

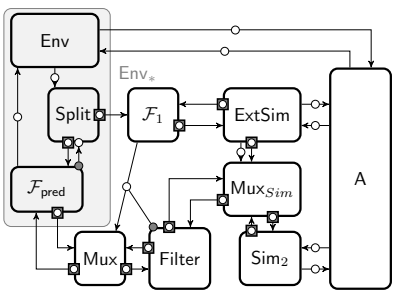
In $conf_4$, all outputs of Sys_1 are sent to $\mathsf{Env}_{*\perp}$. A_{**} is built from an extractor Extr and $\mathsf{OutFilter}$ as in Figure 29b. Here, Extr_i is on the ports for the i-th party and used to split the $(output, \ell_x, x)$ messages similarly to the construction of the extended simulator. Note that there are separate buffers between each party $\mathcal{P}_i$ and the adversary, so the machines Extr_i are each using a set of buffers not affected by the extractors for other parties. Also, $\mathsf{OutFilter}_i$ behaves similarly and has the two ports output and $\mathsf{foutput}$ for either transporting the regular or fast outputs. The fast outputs are those that the system has computed before corruption and need to be distributed correctly after a corruption request. Mux_* behaves like Mux but is connected to the new output and $\mathsf{foutput}$ ports instead of the outputs of Sys_1. The communication with Env_* remains as it was for Mux.



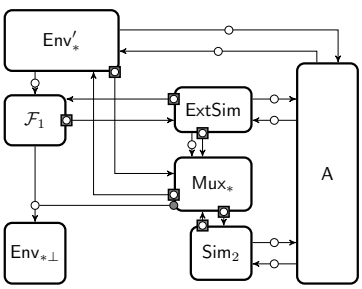
(a) Configuration with the decomposed ideal functionality $\mathcal{F}$ and an ideal adversary, $conf'_0$ and $conf'_1$.



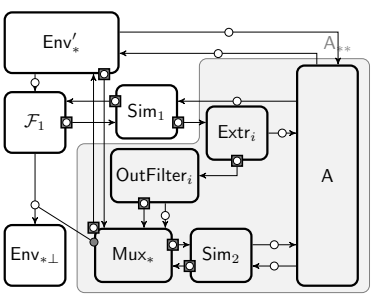
(b) Configuration with the proposed simulator construction with the multiplexing cancelled out, $conf'_2$.



(c) Configuration from the output predictability definition, $conf'_3$ and $conf'_4$.

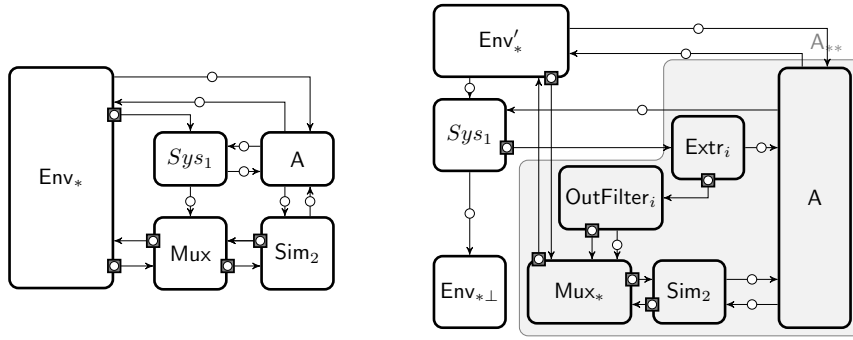


(d) Forming Mux_* and $\mathsf{Env}_{*\perp}$, $conf'_5$.



(e) Decomposing ExtSim and joining machines to A, $conf'_6$ and $conf'_7$.

Figure 28: Configurations to simplify the ordered composition of two ideal systems.



(a) $conf_3$ from Figure 27c. (b) Building A_{**} from $conf_3$.

Figure 29: Constructing $conf_4$ from $conf_3$.

The overall clocking also needs to be adjusted. In Figure 29b, this is marked by the self-clocked port added to Sys_1. Note that this notation is a simplification of the clocking behaviour rather than the true behaviour of Sys_1. If each party in Sys_1 is represented by one machine and never clocks any other machines, then each machine in Sys_1 can simply be modified to add this clocking. In the more general case, an effect similar to Sys_1 clocking any of its messages to A can be obtained by modifying the master scheduler (A_{**}). Either the master scheduler always clocks all buffers from Sys_1 after giving control to Sys_1, or A_{**} clocks the respective buffer when A clocks the buffer from Sys_1 to A to receive these inputs or A clocks the outputs of Sys_1 to $\mathsf{Env}_{*\perp}$. Note that either of these ensures that the right values reach Sim_2 at the same time as in $conf_3$. In addition, clocking of output_i is done at the same time as clocking the respective output of Sys_1 to $\mathsf{Env}_{*\perp}$ to maintain the timing needed by $\mathcal{F}_{\mathsf{pred}}$. Note that, in this case, $\mathsf{Env}_{*\perp}$ may do something, but it does not start any other machines. Hence, the new scheduler can first clock the message to $\mathsf{Env}_{*\perp}$ and then clock the same message in output.

The machines $\mathsf{OutFilter}_i$, Extr_i and Mux_* are merged to A to obtain A_{**}. The machine called Sim_{2*} in $conf_4$ in Figure 27d is the machine Sim_2 simply rewired so that both its connections to A and Mux_* now come from A_*. Note that A_* now communicates with Env as well as $\mathcal{F}_{\mathsf{pred}}$ inside Env'_* although these connections are not all explicitly drawn in $conf_4$ in Figure 27d. Note that the constructions have posed no restrictions to the communication of A and Env and drawing the extra pair of sender-clocked buffers would only complicate the figure. The machine Env'_* is the same as Env_* simply with the updated notation to make the privacy configuration explicit. □

Lemma 8. $view_{conf'_4}(\mathsf{Env}_*) = view_{conf'_5}(\mathsf{Env}_*)$.

Proof. The Filter in $conf'_4$ is used to filter out the inputs sent originally by $\mathcal{F}_2$. However, Mux_{Sim} is defined as ignoring these messages even if it would receive them. Hence Filter can be safely removed from $conf'_4$ without affecting any messages sent out to A or Env_*. However, some machine needs to take over the

clocking role and clock the output buffers from $\mathcal{F}_1$.

The new machine Mux in $conf_4'$ is the multiplexer from the output predictability definition (Definition 27). It is used to deliver the output timing of $\mathcal{F}_1$ to $\mathcal{F}_{\mathsf{pred}}$ and forward messages between $\mathcal{F}_{\mathsf{pred}}$ and A. Especially it forwards the composition outputs learned from $\mathcal{F}_{\mathsf{pred}}$ to A. As discussed before, the inputs of $\mathcal{F}_{\mathsf{pred}}$ are filtered out by Filter and would also be ignored by Mux_{Sim}. Hence Mux can be split to form a part that ignores the value from $\mathcal{F}_1$ and call this $\mathsf{Env}_{*\perp}$.

The second part of Mux is the one forwarding messages. Finally, merge the forwarding part of Mux and Mux_{Sim} to form Mux_* and rewire the timing notification to $\mathcal{F}_{\mathsf{pred}}$. In addition, Mux_* gets the clocking of the output of $\mathcal{F}_1$. Note that, by definition, when $\mathcal{F}_1$ delivers its output to $\mathcal{F}_2$ (or Mux in $conf_4'$), then this timing is duplicated with the timing of clocking output_i buffer of the ExtSim. Here, output_i is clocked by the adversary A and the control is given to Mux_*. Next, Mux_* can send the same signal to $\mathcal{F}_{\mathsf{pred}}$ and by definition it gets the control back from $\mathcal{F}_{\mathsf{pred}}$. Then it can write everything necessary to Sim_2 and get the control back. Finally, Mux_* clocks the output of $\mathcal{F}_1$ to $\mathsf{Env}_{*\perp}$.

Note that as an alternative, the outputs of $\mathcal{F}_1$ can simply remain in the buffer to $\mathsf{Env}_{*\perp}$ as, by definition, $\mathsf{Env}_{*\perp}$ does not interact with other machines and hence this would not change the adversary or Env'_* view of the execution. □

Corollary 3 (Security of private and secure composition for general adversaries)**.** *Let $Sys_1 \geq_{priv}^{model} \mathcal{F}_1$ and $Sys_2 \geq_{sec}^{model} \mathcal{F}_2$ both with black-box security. The model may be perfect, statistical or computational as needed. Let the ordered compositions $Sys_1 \to \mathcal{F}_2$ and $\mathcal{F}_1 \to \mathcal{F}_2$ have a jointly predictable outcome. Then, for fully ordered composition $Sys_1 \to Sys_2$, $Sys_1 \to Sys_2 \geq_{sec}^{model} \mathcal{F}$ with black-box simulation where $\mathcal{F}$ is the ideal composition of $\mathcal{F}_1$ and $\mathcal{F}_2$ (as in Definition 26).*

Proof. This is a direct result from Theorem 6 and Lemma 1, respectively, showing the result for simultaneous corruption and the fact that simultaneous corruption is generic. □

The main restriction of the composed secure system (Corollary 3) is that all outputs of the private system have to be used (hence, at least rerandomised) by the secure functionality.

4.8. Application of the Composition Theorems

This section focuses on examples where the composition theorems are useful and give rise to new efficient protocols. The intuition that input privacy combined with secure protocol gives a secure protocol was known prior to the given formalisation of the approach. This had been explicitly used by protocols in Sharemind as discussed in Section 4.8.1. A similar pattern has also been used by other multiplication protocols. Section 2.1.4 introduced Maurer's and Gennaro-Rabin-Rabin multiplication protocols that first use the multiplicativity of the secret sharing scheme

to compute the multiplication result and then share this result to get valid shares. The first part of the computations is local and, therefore, also input-private. The sharing operation is secure and the full protocol is, therefore, an ordered composition of input-private and secure protocol that is also secure. Hence, these well-known protocols can be seen as following the same structure as now generalised by the definition of input privacy and the composition theorem.

In addition, the proof framework was very explicitly used to prove the security of a hybrid protocol combining garbled circuits and additive secret sharing. This protocol, as well as the proof of security, are given in Section 4.8.2. In addition to the work of the author, the proof framework has been used in different manners by other authors, as discussed in Section 4.8.3.

4.8.1. Multiplication in Sharemind

Sharemind [28, 32, 33] is a framework for secure multiparty computation that can support different protection domains. However, its most developed protection domain is that of three parties using additive secret sharing and tolerating one passively corrupted party. This framework also used input privacy and secure protocol composition idea before it was thoroughly formalised. For example, the informal version of the composition of input-private and secure protocol appears as Theorem 5 in [28] and was already considered in [32] as perfect simulatability in combination with resharing the output. The simplest example for using the input-private and secure composition result is the multiplication protocol where the secure protocol is `Reshare` in Algorithm 11.

The core of the protection domain is the storage domain with additive secret sharing over rings $\mathbb{Z}_{2^k}$. A shared value $[\![x]\!] = ([\![x]\!]_1, [\![x]\!]_2, [\![x]\!]_3)$ is such that $x = [\![x]\!]_1 + [\![x]\!]_2 + [\![x]\!]_3 \mod 2^k$ and each party $\mathcal{P}_i$ knows $[\![x]\!]_i$. Shares are generated so that each share individually is uniformly random in $\mathbb{Z}_{2^k}$. Additive secret sharing is linear, meaning that addition and subtraction can be computed locally as

$$[\![x+y]\!] = [\![x]\!] + [\![y]\!] = ([\![x]\!]_1 + [\![y]\!]_1, [\![x]\!]_2 + [\![y]\!]_2, [\![x]\!]_3 + [\![y]\!]_3) \ ,$$

where $\mathcal{P}_i$ computes $[\![x]\!]_i + [\![y]\!]_i$ on its own. As all local protocols are, by definition, input-private, then so are addition and subtraction. The resharing protocol is given in Algorithm 11 and multiplication in Algorithm 12 based on their description in [28, 33]. The protocols introduced here are also used in Section 4.8.2.

The ideal functionality $\mathcal{F}_{\texttt{Reshare}}$ for `Reshare` is defined basically as the input and output definition in Algorithm 11. The ideal functionality reconstructs the value of x and then shares it again to obtain fresh shares for the value x that is denoted as $[\![y]\!]$. Note that, by definition, there is no dependency between $[\![x]\!]_i$ and $[\![y]\!]_i$ other than that $[\![x]\!]$ and $[\![y]\!]$ can be reconstructed to the same value. Perfect security of resharing for one passively corrupted party is easy to see. Nothing in the protocol execution reveals information about x, and, for a uniformly randomly chosen r_i, the value $a + r_i \mod 2^k$ is uniformly random for all a. On the other

Algorithm 11: Resharing protocol `Reshare`($[\![x]\!]$) in Sharemind.

Input: $[\![x]\!]$, each party $\mathcal{P}_i$ inputs $[\![x]\!]_i$
Output: $[\![y]\!]$ such that $y = x$ and $[\![y]\!]_i$ is uniformly random
$\mathcal{P}_i$ creates $r_{i+1} \xleftarrow{\$} \mathbb{Z}_{2^k}$
$\mathcal{P}_i$ sends r_{i+1} to $\mathcal{P}_{i+1}$
$\mathcal{P}_i$ computes $[\![y]\!]_i = [\![x_i]\!] + r_{i+1} - r_i$
return $[\![y]\!]$

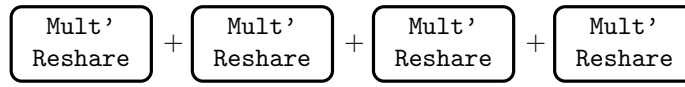
Algorithm 12: Multiplication protocol `Mult` or $[\![x]\!] \cdot [\![y]\!]$ in Sharemind.

Input: $[\![x]\!], [\![y]\!]$
Output: $[\![w]\!]$ where $w = xy$
$[\![u]\!] =$ `Reshare`$([\![x]\!])$
$[\![v]\!] =$ `Reshare`$([\![y]\!])$
$\mathcal{P}_i$ sends $[\![u]\!]_i$ and $[\![v]\!]_i$ to $\mathcal{P}_{i+1}$
$\mathcal{P}_i$ computes $[\![r]\!]_i = [\![u]\!]_i[\![v]\!]_i + [\![u]\!]_i[\![v]\!]_{i-1} + [\![u]\!]_{i-1}[\![v]\!]_i$
$[\![w]\!] =$ `Reshare`$([\![r]\!])$
return $[\![w]\!]$

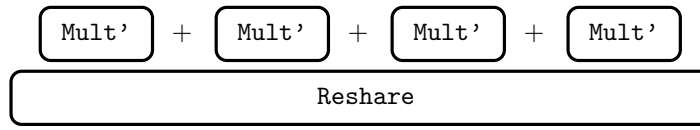
hand, it is easy to simulate a value r_i to get a desired output $[\![y]\!]_i$ from a known $[\![x]\!]_i$. Proper security analysis can be found in [28]. The protocol is also input-private as the output is a secret shared value with uniform shares.

The ideal functionality for multiplication reconstructs x and y, computes $w = xy$ in public and then secret shares w. The multiplication protocol in Algorithm 12 is a composition of `Reshare`, local computations and distributing shares. Its security is proven in [33]. Intuitively, the part of `Mult` before the final `Reshare` (denoted as `Mult'`) is input-private because the values $[\![u]\!]_i$ and $[\![v]\!]_i$ that are sent are uniformly random and each party $\mathcal{P}_i$ only sees $[\![u]\!]_i$ and $[\![v]\!]_i$ and $[\![u]\!]_{i-1}$ and $[\![v]\!]_{i-1}$ in the protocol which does not reveal x or y because $[\![u]\!]_{i+1}$ and $[\![v]\!]_{i+1}$ are also uniformly random and hide the values. Hence, the input privacy simulator can simulate the protocol by sending uniformly random values. However, without the final `Reshare`, the multiplication protocol `Mult'` is not secure.

A significant distinction between the input-private and secure protocols in terms of how the properties are proven is that, for security, the proof has to ensure that the simulated output is the same as given out by the ideal functionality. Without the reshare, the protocol would output $[\![r]\!]$ that is computed from $[\![u]\!]_i$, $[\![u]\!]_{i-1}$, $[\![v]\!]_i$ and $[\![v]\!]_{i-1}$. Hence, in the simulation, the values of $[\![u]\!]_i$, $[\![u]\!]_{i-1}$, $[\![v]\!]_i$ and $[\![v]\!]_{i-1}$ would need to be computed from the desired output $[\![r]\!]_i$ and would still need to have a uniform distribution in $\mathbb{Z}_{2^k}$ as they are outputs of `Reshare`. However, there are correlations between these values. For example if $[\![u]\!]_i = 0$ and $[\![v]\!]_i = 0$, then it must be that $[\![r]\!]_i = 0$. In this case, it means that a valid-looking protocol execution might be impossible to simulate. The common simu-



(a) Protocol with full multiplication on each step.



(b) Protocol with private multiplication before summation and final resharing.

Figure 30: Two variations of computing an inner product of a vector of length 4.

lation strategy to choose three of these values as uniformly random and compute the fourth from these choices and $[\![r]\!]_i$ does not work since not all elements in a ring $\mathbb{Z}_{2^k}$ have multiplicative inverses, and therefore, it is not possible to always have a solution for x in $ax = b$, for a given b and a, especially if a does not have a multiplicative inverse. A more complicated simulation strategy might try to generate several of these values so that they depend on each other, but, in this case, the final distribution would not be the same as the one where each value is chosen uniformly at random and independently of each other. Note that also the full multiplication `Mult` is input-private. The simulator used to prove the security of `Mult` can be transformed to prove input privacy simply by generating uniformly random output share $[\![w]\!]_i$ for the corrupted party $\mathcal{P}_i$.

Note that, in the case of multiplication, it is useful to be able to consider the secure and input-private versions of the protocol separately. For example, computing the inner product of two secret-shared vectors of length ℓ with the secure protocol, then the final resharing step is needed for each of the ℓ multiplications. However, knowing the composition results for input privacy and security, it is instead possible to propose a protocol that first computes the input-private inner product and then, in the end, does one resharing of the result to get the secure inner product protocol. Hence, this is one instance where considering input privacy helps to achieve more efficient protocols. The structure of the two protocols is illustrated in Figure 30.

4.8.2. Combining Garbled Circuits and Additive Secret Sharing

This subsection describes the protocol for combining garbled circuits with additive secret sharing. This section generalises the approach originally proposed in [125] for the concrete use case of floating-point operations. In the current description, the focus is on the construction of the protocol and the security proof using the composition of input-private and secure components. This protocol is tailored for the three-party additive secret sharing-based protocol in Sharemind that tolerates one passively corrupted party.

Overall, the protocol is intended to integrate a garbled circuit evaluation into

otherwise additive secret-sharing-based execution. Hence the inputs and outputs are expected to be secret-shared. The overall idea is to start with the shares, then execute some computation with input privacy using garbled circuits, and convert the result back to secret sharing with a secure resharing protocol. This approach enables to add new functionality to Sharemind efficiently as the garbled circuits method can be used for any computation representable as a circuit and there are various tools available for generating and optimising the underlying circuits [85, 92]. Originally, this approach was used to enhance Sharemind with the floating-point operations that follow the full IEEE 754 floating-point standard [86] rather than implementing some approximation of floating-point operations directly on secret shares, for example, like in [87]. Hence, this is a version of combining secure computation methods as is discussed in Section 2.1.7.

Protocol Description. The garbled circuits protocol is, by definition, asymmetric, as the garbler and evaluator have distinct roles. Hence, the hybrid protocol resulting from combining the protocol with additive secret sharing for three parties is also asymmetric, and the protocol needs to be described separately for all parties. The input of the protocol is secret-shared among all three parties. During the protocol execution, they all take part in the oblivious transfer to transform these shared inputs to the input encodings of the garbled circuit. In particular, the protocol `OT` defined in Algorithm 13 is used. In addition, they all participate in the final resharing protocol `Reshare` in Algorithm 11 to translate the outputs of the garbled circuit back to additive shares.

The standard oblivious transfer is a two-party protocol where one party has two inputs x_0, x_1 and another party has a bit b. As a result of the protocol, the second party learns x_b and nothing about the other element. The first party learns nothing about b. The following three-party protocol requires a special flavour of assisted oblivious transfer where the first party knows the inputs, the choice is secret shared and the second party receives the output. A version of such an assisted oblivious transfer protocol is given in Algorithm 13. Note that the local computations of the algorithm are input-private and other computations are secure. Concretely, the respective protocols (multiplication, addition and subtraction) of the Sharemind framework can be used. The following security proof argues that this protocol is a suitable component for the given passively secure hybrid protocol execution. Note that the protocol until the computation of $[\![X]\!]$ is often also known under the name *oblivious choice* where the computations have already chosen the necessary encodings. The last lines are publishing this choice to $\mathcal{P}_2$ to turn the protocol into oblivious transfer.

Note that the bit-vector $\mathbf{x}$ is secret-shared in a ring that is suitably big to also fit X_i^b as elements without overflow. Conversions to and from a more efficient representation of a bit vector are possible, similarly to all kinds of transformations between secure data representations but are not of interest to the discussion here.

Recall that the garbled circuits specified in Section 2.1.5 are defined by two algorithms, $\mathsf{Gb}(f)$ for garbling and $\mathsf{Ev}(F, X_1, \ldots, F_n)$ for evaluation. The garbling

Algorithm 13: Oblivious transfer of input tokens (OT).

Input: $\mathcal{P}_1$ holds the input tokens $(X_1^0,\ldots,X_n^0,X_1^1,\ldots,X_n^1)$
The input choice bit vector $[\![\mathbf{x}]\!] = [\![x_1,\ldots,x_n]\!]$ is shared between all parties
Output: $\mathcal{P}_2$ receives input tokens $(X_1^{x_1},\ldots,X_n^{x_n})$
if *Executed by* $\mathcal{P}_1$ **then**
  $[\![\mathbf{X^0}]\!]_1 = (X_1^0,\ldots,X_n^0)$
  $[\![\mathbf{X^1}]\!]_1 = (X_1^1,\ldots,X_n^1)$
else if *Executed by* $\mathcal{P}_i$ *where* $i \neq 1$ **then**
  $[\![\mathbf{X^0}]\!]_i = (0,\ldots,0)$
  $[\![\mathbf{X^1}]\!]_i = (0,\ldots,0)$
$[\![\mathbf{X}]\!] \leftarrow [\![\mathbf{X^0}]\!] \cdot ([\![\mathbf{1}]\!] - [\![\mathbf{x}]\!]) + [\![\mathbf{X^1}]\!] \cdot [\![\mathbf{x}]\!]$
$\mathcal{P}_1$ and $\mathcal{P}_3$ send their shares of $[\![\mathbf{X}]\!]$ to $\mathcal{P}_2$
$\mathcal{P}_2$ uses all shares of $[\![\mathbf{X}]\!]$ to reconstruct $(X_1^{x_1},\ldots,X_n^{x_n})$
return $(X_1^{x_1},\ldots,X_n^{x_n})$ *to* $\mathcal{P}_2$

algorithm defines the garbled circuit F, the input encoding e and the output decoding d. The circuit is defined by a function $f = (n,m,q,A,B,G)$, where n is the number of inputs, m is the number of outputs, q is the number of gates and A, B and G determine the wiring and functionality of each gate. During the garbled circuit phase of the protocol, $\mathcal{P}_1$ is in the role of the garbler. The full description of $\mathcal{P}_1$ is given in Algorithm 14. During garbling, $\mathcal{P}_1$ generates encodings X_i^1 and X_i^0 for each input bit x_i. The evaluation of the garbled circuit is carried out by $\mathcal{P}_2$ according to the protocol in Algorithm 15. For that, $\mathcal{P}_2$ receives the input encodings $X_i^{x_i}$ for the shared input $[\![x_i]\!]$ using OT. Party $\mathcal{P}_3$ does not participate in the garbled circuit evaluation directly, but its shares are needed for the input encodings and it receives an output that is again secret-shared between three parties. The protocol for $\mathcal{P}_3$ is given in Algorithm 16. All parties participate in the final protocol step to reshare $[\![\mathbf{y}]\!]'$ to $[\![\mathbf{y}]\!]$. The protocol descriptions show the call to Reshare with the input that each party has.

As the protocol is generic, the description assumes that it takes an input vector $\mathbf{x}$ of length n and gives an output vector of length m. It is assumed that all inputs are one-bit elements. The core idea of the whole protocol is to take the function $f(\mathbf{x})$ that the parties want to evaluate and replace it with a function $f(\mathbf{x}) - \mathbf{y}_1$ where $\mathbf{y}_1$ is randomly generated by $\mathcal{P}_1$. The parties together give $\mathcal{P}_2$ the encoding for $\mathbf{x}$ so that $\mathcal{P}_2$ evaluates the circuit to learn the output $\mathbf{y}_2$ of the function. Together, $\mathbf{y}_1$ and $\mathbf{y}_2$ are additive shares for the result of $f(\mathbf{x})$. These can be interpreted as shares for three parties with $\mathbf{y}_3 = \mathbf{0}$. However, these need to be reshared in order to get random shares of $\mathbf{y}$ for three parties.

Security Proof of the Yao and Secret Sharing Combination. This section deploys the input privacy and security composition to show that the Yao-additive protocol defined by Algorithms 14, 15 and 16 is secure against passive static adversaries that corrupt at most one participant. The Yao-additive protocol can be

Algorithm 14: Hybrid protocol algorithm for $\mathcal{P}_1$.

Input: $[\![\mathbf{x}]\!]_1 = [\![(x_1,\ldots,x_n)]\!]_1$ and circuit $f = (n,m,q,A,B,G)$
Output: $[\![\mathbf{y}]\!]_1 = [\![(y_1,\ldots,y_m)]\!]_1$ such that $\mathbf{y} = f(\mathbf{x})$
$\mathbf{y}'_1 \xleftarrow{\$} \{0,1\}^m$
$(F,e,d) \leftarrow \mathsf{Gb}(f(\mathbf{x}) - \mathbf{y}'_1)$
Use e to derive all input encodings $(X_1^0,\ldots,X_n^0),(X_1^1,\ldots,X_n^1)$
$\mathtt{OT}$ $((X_1^0,\ldots,X_n^0),(X_1^1,\ldots,X_n^1),[\![\mathbf{x}]\!]_1)$ `// As a sender with messages` $(X_1^0,\ldots,X_n^0),(X_1^1,\ldots,X_n^1)$ `in Algorithm 13`
Send F,d to $\mathcal{P}_2$
Set $[\![\mathbf{y}]\!]'_1 = \mathbf{y}'_1$
$[\![\mathbf{y}]\!]_1 \leftarrow \mathtt{Reshare}([\![\mathbf{y}]\!]'_1)$`// Together the parties execute` $[\![\mathbf{y}]\!] = \mathtt{Reshare}([\![\mathbf{y}]\!]')$
return $[\![\mathbf{y}]\!]_1$

Algorithm 15: Hybrid protocol algorithm for $\mathcal{P}_2$.

Input: $[\![\mathbf{x}]\!]_2 = [\![(x_1,\ldots,x_n)]\!]_2$ and circuit $f = (n,m,q,A,B,G)$
Output: $[\![\mathbf{y}]\!]_2 = [\![(y_1,\ldots,y_m)]\!]_2$ such that $\mathbf{y} = f(\mathbf{x})$
$(X_1,\ldots,X_n) \leftarrow \mathtt{OT}$ $([\![\mathbf{x}]\!]_2)$ `// Receiver for choice` $[\![\mathbf{x}]\!]$ `in Algorithm 13`
Receive F,d from $\mathcal{P}_1$
$\mathbf{Y} = \mathsf{Ev}(F,X_1,\ldots,X_n)$
Obtain $[\![\mathbf{y}]\!]'_2$ by decoding $\mathbf{Y}$ using d
$[\![\mathbf{y}]\!]_2 \leftarrow \mathtt{Reshare}([\![\mathbf{y}]\!]'_2)$ `// Together the parties execute` $[\![\mathbf{y}]\!] = \mathtt{Reshare}([\![\mathbf{y}]\!]')$
return $[\![\mathbf{y}]\!]_2$

separated into two parts. The first part is everything in the algorithms before the final line that runs `Reshare`. This part will be called `Hybrid'` in the following. The second part is just the `Reshare` protocol on its own. In this light, the full protocol is a fully ordered composition of `Hybrid'` followed by `Reshare` (Algorithm 11). Hence, to use the composition results, this section needs to show input privacy of `Hybrid'` and that the composition is jointly output predictable. Note that `Reshare` functionality is secure against a passive adversary.

The concrete version of the Yao-additive protocol implemented in [125] used the GAXR garbling scheme from [19, 20]. However, the current exposition generalises the original protocol and the proof for any correct garbling scheme with `prv.sim` security (defined in Section 2.1.5). The original security proof in [125] relied on the concrete details of the encoding and decoding functions used in the GAXR scheme. In the version presented in this thesis, this has been replaced by the blinding of the actual circuit with the vector $\mathbf{y}'_1$. The current approach will directly embed the discussion regarding the privacy of the garbling scheme into

Algorithm 16: Hybrid protocol algorithm for $\mathcal{P}_3$.

Input: $[\![\mathbf{x}]\!]_3 = [\![(x_1,\ldots,x_n)]\!]_3$
Output: $[\![\mathbf{y}]\!]_3 = [\![(y_1,\ldots,y_m)]\!]_3$ such that $\mathbf{y} = f(\mathbf{x})$
`OT` $([\![\mathbf{x}]\!]_3)$ `// Inputting part of the shared bit vector` $[\![\mathbf{x}]\!]$ `in Algorithm 13`
$[\![\mathbf{y}]\!]'_3 \leftarrow 0^m$
$[\![\mathbf{y}]\!]_3 \leftarrow$ `Reshare`$([\![\mathbf{y}]\!]'_3)$ `// Together the parties execute` $[\![\mathbf{y}]\!] =$ `Reshare`$([\![\mathbf{y}]\!]')$
return $[\![\mathbf{y}]\!]_3$

the input privacy discussion for party $\mathcal{P}_2$ in Theorem 7. The blinding is needed to ensure the privacy of the inputs as otherwise the evaluator $\mathcal{P}_2$ would learn the output of the circuit. Such additive blinding is also convenient as the result is transformed back to additive secret sharing.

The ideal functionality $\mathcal{F}_{\texttt{Reshare}}$ for `Reshare` takes the input as shares and outputs a uniformly random sharing of the input value. The ideal input private functionality for `Hybrid'` takes the shares $[\![\mathbf{x}]\!]$ and outputs the value $\mathbf{y} = f(\mathbf{x})$ as shares $[\![\mathbf{y}]\!]$ where $[\![\mathbf{y}]\!]_3 = (0,\ldots,0)$ but $[\![\mathbf{y}]\!]_1$ and $[\![\mathbf{y}]\!]_2$ are uniformly random additive shares. The corresponding ideal functionality for the composed protocol outputs uniformly random shares of $f(\mathbf{x})$. Note that these functionalities fit the intuition where the functionality could be computed by first reconstructing the value, then computing the necessary function (identity function for `Reshare` and $f(\cdot)$ for the composed functionality), and finally, sharing the output.

As noted before, output predictability is trivial for correct input-private functionalities that are deterministic. The correctness of the private functionality does not follow from privacy. However, it is necessary to ensure the correctness of the composed protocol with respect to the composed ideal functionality. The correctness of `Hybrid'` is proven in Lemma 9.

Lemma 9. *`Hybrid'` protocol is correct if the garbling scheme and oblivious transfer are correct.*

Proof. In short, the correctness follows from the correctness of the sub-protocols used in the `OT` part and the correctness of the garbling scheme. `Hybrid'` is correct, if $[\![\mathbf{y}]\!]'$ is such that $\mathbf{y} = f(\mathbf{x})$ for the shared input $[\![\mathbf{x}]\!]$. If the oblivious transfer transfers the right keys, then the correctness of the garbling Gb and evaluation Ev algorithms ensures the correct output for the garbling of the function $[\![\mathbf{y}]\!]'_2 = f(x) - [\![\mathbf{y}]\!]'_1$. By definition $[\![\mathbf{y}]\!]'_3 = \mathbf{0}$. Hence, $[\![\mathbf{y}]\!]'_1 + [\![\mathbf{y}]\!]'_2 + [\![\mathbf{y}]\!]'_3 = f(\mathbf{x})$ and the output is correct.

The `OT` in Algorithm 13 is correct if the used subprotocols for arithmetic operations are correct. If the shared secret input bit $x_i = 0$ then $[\![X_i]\!] = [\![X_i^0]\!] \cdot (1 - 0) + [\![X_i^1]\!] \cdot 0 = [\![X_i^0]\!]$. On the other hand, if $x_i = 1$, then $[\![X_i]\!] = [\![X_i^1]\!]$. In total, $[\![X_i]\!] = [\![X_i^{x_i}]\!]$ for all i and the `OT` protocol is correct. □

Corollary 4. *The ordered composition of* `Hybrid'` *and its corresponding ideal functionality* $\mathcal{F}_{\texttt{Hybrid'}}$ *with* $\mathcal{F}_{\texttt{Reshare}}$ *have a jointly predictable outcome.*

Proof. Lemma 4 states that these ordered compositions $\texttt{Hybrid'} \to \mathcal{F}_{\texttt{Reshare}}$ and $\mathcal{F}_{\texttt{Hybrid'}} \to \mathcal{F}_{\texttt{Reshare}}$ have a jointly predictable outcome if and only if $\texttt{Hybrid'} \to \mathcal{F}_{\texttt{Reshare}}$ has a predictable outcome and `Hybrid'` is a correct implementation of $\mathcal{F}_{\texttt{Hybrid'}}$. Hence, it is sufficient for joint output predictability to show an output predictor for the ordered composition of `Hybrid'` protocol and ideal `Reshare`, as Lemma 9 shows that `Hybrid'` is correct.

The output predictor $\mathcal{F}_{\mathsf{pred}}$ for $\texttt{Hybrid'} \to \mathcal{F}_{\texttt{Reshare}}$ gets the input $[\![\mathbf{x}]\!]$ and must predict the output of $[\![\mathbf{y}]\!]$. The functionality $\mathcal{F}_{\mathsf{pred}}$ also knows the function $f(\mathbf{x})$. Hence, the predictor can reconstruct the value $\mathbf{x}$, compute $\mathbf{y} = f(\mathbf{x})$. Finally, it secret shares $\mathbf{y}$ and gives these shares as its output.

Given that the output of `Hybrid'` is correct and the functionality is deterministic (the value of $\mathbf{y}'$ is deterministic), the predictor computes the right output values. In addition, the behaviour of $\mathcal{F}_{\texttt{Reshare}}$ depends only on the value $\mathbf{y}'$ and not on the shares $[\![y]\!]'_i$ and, by definition, it outputs fresh shares of the input. Hence, the predictor gives the right output value with the right distribution of the shares and is therefore correct. □

In order to use the composition results for input-private and secure protocols, it is necessary to show that `Hybrid'` is indeed input-private. For that, a deeper look must be taken into the additive secret sharing used for inputs and the `OT` protocol and how it is combined with the garbling scheme. For input privacy of `Hybrid'`, it is intuitively clear that the output obtained by the evaluator $\mathcal{P}_2$ should not leak the actual output of $f(\mathbf{x})$. The `obv.sim` property (Figure 3 in Section 2.1.5) of the garbling scheme is quite similar to the input privacy property (Definition 20). Intuitively, the `obv.sim` property is such that the adversary sees the garbled circuit and the input encoding but has no way of decoding the output. Everything that the adversary sees is indistinguishable from the simulated circuit that is generated using the side information about the circuit. However, the following uses the `prv.sim` (Figure 3 property of the garbling in combination with the blinding of the output provided by the protocol to obtain an effect similar to that of `obv.sim`. More concretely, the following theorem uses the simulator from the `prv.sim` definition of garbled circuits to show the input privacy of `Hybrid'`. For simplicity, the computation part of the `OT` protocol in Algorithm 13 is called the oblivious choice to distinguish it from the final reconstruction part where the values of the input tokens are opened for the evaluator $\mathcal{P}_2$.

Theorem 7. *Protocol* `Hybrid'` *is perfectly input-private for passively corrupted* $\mathcal{P}_1$ *or* $\mathcal{P}_3$ *and computationally input-private against passively corrupted* $\mathcal{P}_2$ *for any garbling scheme with* `prv.sim` *security.*

Proof. The proof separately considers all parties and demonstrates an input privacy simulator that gets the inputs of the corrupted party and simulates the view

of the corrupted party. The view contains all messages received by the party.

Party $\mathcal{P}_1$ or $\mathcal{P}_3$ is corrupted. The inputs of the parties are their shares of $[\![\mathbf{x}]\!]$ and the only received communication in `Hybrid'` is in the `OT` protocol during the computation of the oblivious choice. Hence, perfect input privacy for these parties follows from the perfect input privacy of the addition, subtraction and multiplication protocols used in `OT` and the composition result for input-private protocols (Theorem 5).

Party $\mathcal{P}_2$ is corrupted. First, $\mathcal{P}_2$ is the evaluator so he is the adversary against the garbling scheme and needs to be considered with respect to the `prv.sim` property. The goal of the following is to define a simulator Sim for the input privacy property for corrupted $\mathcal{P}_2$. By definition, the garbling scheme has `prv.sim` security, hence the simulator from the `prv.sim` game in Figure 3 must exist. The following denotes this simulator from `prv.sim` definition as $\mathsf{Sim}_{\mathsf{prv}}$. The simulator Sim can use the $\mathsf{Sim}_{\mathsf{prv}}$ if it can give it the inputs of $\mathcal{P}_2$, the circuit that is garbled, the expected output and the security parameter. The input privacy simulator gets all the inputs of $\mathcal{P}_2$ from the ideal functionality so it knows the shares $[\![\mathbf{x}]\!]_2$. It knows the function $f(\mathbf{x})$ that is computed, as well as the security parameter k defining the length of the encoding. Hence, the privacy simulator Sim can run $\mathsf{Sim}_{\mathsf{prv}}$ if it can also provide the desired output for $f(\mathbf{x}) - \mathbf{y}'_1$. It can also use the simulators for the components protocols.

The input privacy simulator Sim works as follows.

- It generates a uniformly random $[\![\mathbf{y}]\!]'_2$ and uses these values as the output of $f(\mathbf{x}) - \mathbf{y}'_1$ for $\mathsf{Sim}_{\mathsf{prv}}$. It sends the result of the computation $[\![\mathbf{y}]\!]'_2$, the side information $\Phi(f(\mathbf{x}) - \mathbf{y}'_1)$ about the computed circuit, and the security parameter k to $\mathsf{Sim}_{\mathsf{prv}}$.
- It gets the garbled circuit F, the input encoding X and the output decoding information d for the garbling of $f(\mathbf{x} - \mathbf{y}'_1)$ from $\mathsf{Sim}_{\mathsf{prv}}$.
- It simulates the oblivious choice protocol with output $\mathbf{X}$and input $[\![\mathbf{x}]\!]_2$. For that, it uses the input-privacy simulators for all protocols used to compute $\mathbf{X}$ in Algorithm 13. This simulation gives the shares $[\![\mathbf{X}]\!]_2$.
- Finally, it has to simulate publishing the value $\mathbf{X}$ in Algorithm 13. It uses the shares $[\![\mathbf{X}]\!]_2$ and the desired result $\mathbf{X}$ as input. For that it generates a uniformly random $[\![\mathbf{X}]\!]_1$ and computes $[\![\mathbf{X}]\!]_3$ such that $[\![\mathbf{X}]\!]_1 + [\![\mathbf{X}]\!]_2 + [\![\mathbf{X}]\!]_3 = \mathbf{X}$.

The analysis of the correctness of the proposed privacy simulator follows.

Firstly, consider the straightforward simulation of the `OT` protocol. The oblivious choice part of this protocol is perfectly input-private because it is an ordered composition of perfectly input-private protocols for addition, multiplication and subtraction. Hence, there exists a privacy simulator $\mathsf{Sim}_{\mathsf{choice}}$ that can be used as a building block of the privacy simulator for the whole protocol. This simulator expects only the shares $[\![\mathbf{x}]\!]_2$ as inputs of the party $\mathcal{P}_2$ and the privacy simulator Sim can give it these. The reconstruction step can be simulated thanks to the additional details of the oblivious choice part. The multiplication protocol (Algorithm 12)

used in OT is secure. Hence, output shares of the multiplication are uniformly random. Hence, the final addition protocol in the oblivious choice adds to uniformly random shares and, therefore, the shares $[\![\mathbf{X}]\!]_i$ are uniformly random for $i \in \{1,2,3\}$. Hence, the privacy simulator for the oblivious choice outputs a suitable $[\![\mathbf{X}]\!]_2$, and the reconstruction is perfectly simulated by choosing uniformly random $[\![\mathbf{X}]\!]_1$ and computing $[\![\mathbf{X}]\!]_3 = \mathbf{X} - [\![\mathbf{X}]\!]_1 - [\![\mathbf{X}]\!]_2$.

As a second part, consider how the prv.sim property of the garbling scheme can be broken if the described simulator is not a valid privacy simulator. Assume, by contradiction, that there exists some environment $\mathsf{Env} = \mathsf{Env}' \cup \mathsf{Env}_\perp$ and adversary A pair where the adversary corrupts $\mathcal{P}_2$ that can distinguish the real and simulated protocol run for some function $f(\mathbf{x})$. Consider an adversary B against the prv.sim game that mostly operates as the protocol Hybrid' and interacts with A and Env as well as the prv.sim game. B needs to submit the functionality f and the input $\mathbf{x}$ to the prv.sim game GARBLE function. As B is playing the role of the protocol, it receives the shares $[\![\mathbf{x}]\!]$ of the input from Env'. For most of the protocol, it simulates the protocol for A as the privacy simulator but B uses the output of GARBLE as the message that $\mathcal{P}_1$ sends to $\mathcal{P}_2$ with the garbling result. In the end, B outputs whether Env' considered this to be the real or simulated interaction.

The rest of the simulation, other than the (F,X,d) generated by GARBLE, is perfect. Hence, the only distinguishing factor can be if the GARBLE gave the real or simulated output. In the case when GARBLE gives the real garbling output, the view of A and Env' is the same as in a real protocol execution. Therefore, B wins against prv.sim exactly when Env' can distinguish the real and simulated protocol. This is a contradiction as the garbling scheme has prv.sim security. Hence, as the secret sharing part of the privacy simulation is perfect, the privacy simulator is correct as long as the garbling scheme has prv.sim security.

Note that the computational security for $\mathcal{P}_2$ results from the security definition of the garbled circuits that only considers polynomial-time adversaries, and therefore, the simulated and real protocols are computationally indistinguishable. Also, note that if A runs in polynomial time, then so does B as the protocol and its simulation are polynomial time. □

Corollary 5. *A hybrid protocol algorithm combining Hybrid' and Reshare is perfectly secure against passively corrupted $\mathcal{P}_1$ and $\mathcal{P}_3$, and as secure against a passively corrupted $\mathcal{P}_2$, as the used garbling scheme.*

Proof. Corollary 4 says that the composition of Hybrid' and Reshare is jointly output predictable. Clearly, Hybrid' and Reshare are in ordered composition because all outputs of Hybrid' are inputs to the secure Reshare protocol, and there is no data dependency from Reshare to Hybrid'. Theorem 7 also showed that Hybrid' is input-private. From Section 4.8.1, it is known that Reshare is secure for one passively corrupted party. Therefore, the composition result that

ordered composition of input-private and secure protocols is secure if the composition has a predictable outcome (Theorem 6) can be applied. In conclusion, the full protocol proposed in Algorithm 14, Algorithm 15 and Algorithm 16 with the `OT` in Algorithm 13 is secure against one passively corrupted party. □

4.8.3. Other Works using Input Privacy Based Security Proofs

Since the original publication of [30], the approach of combining input privacy and security for MPC protocols has also been used in other papers.

New protocols for equality and comparison were proposed in [52]. They show that their preprocessing for the equality test protocol is input-private and the online phase of the protocol is secure. They also show that their composed protocol has a predictable outcome. In addition, all outputs of the preprocessing are used in the online phase meaning that the two protocols are in ordered composition and, therefore, their composition into the full equality test protocol is secure.

A language-based security framework for passive security, where the main security notion is similar to input privacy, is proposed in [3]. Hence, the composition results for input-private and secure protocols enable us to easily lift the protocols that are secure in their framework to protocols that are also secure with respect to the traditional universal composability-based security definition. They consider the ideal execution similar to the definition used in this work, where the ideal functionality always first reconstructs the input and then computes the output. The simulator for the privacy definition learns the shares of the corrupted parties and any leakage that the ideal functionality has based on the value that is shared. The main difference between their definition and input privacy in Definition 20 is that correctness in terms of values is also included in their definition, as the adversary can also see the output value computed using the desired function and the reconstructed input. However, similar to input privacy, this value is not available to the simulator.

4.9. Input Privacy and Statistical Disclosure Limitations

Statistical disclosure limitations are data processing techniques used to limit the likelihood of identifying or disclosing private information from statistical datasets. The goal is to either ensure that the results of queries do not reveal more than intended or that some dataset can be published while ensuring the data is not identifiable. This approach covers various privacy enhancing technologies like differential privacy [63], de-identification [69] or k-anonymity [131]. These methods are defined for a fundamentally different setting than the input privacy considered here. In general, disclosure limitations are used in cases where the computation gives some output that is useable and gives new information to the parties seeing it, simply not information that is deemed private. For example, for differential privacy, the definition specifies that it should be hard to decide if data from a certain participant was used to answer the query. For our version of input privacy, it is

required that the output seen by any participant in the input-private protocol does not reveal any new information to that party. Techniques providing statistical disclosure limitation are sometimes also referred to as tools providing output privacy as they ensure the outputs maintain some desired level of privacy of the inputs.

On the other hand, as discussed, the definition of input privacy does not specify what the outputs of input-private computations could reveal if distributed across the participants of the computation. This chapter discussed how to ensure the security of the protocol when using input-private components and this limits the possible leaks from individual views. However, it is still possible that the reconstructed outputs obtained from the secure computation reveal more information than desired by the input parties. In this case, secure computation should be combined with methods to limit the data disclosure. For example, the combination of MPC and differential privacy has received a lot of attention starting with the works [65, 119]. Similarly, statistical analysis tools using MPC can easily include k-anonymity principles and refuse queries that affect less than k data records as done by Rmind [29][1].

4.10. Input Privacy for Active Adversary

Input privacy formalisation and composition theorems consider input privacy for passive adversaries. It is likely that the notion and similar results can be extended to active security as well. The exact details of this extension require careful thought, but the following describes some of the main aspects that need consideration. Firstly, the proper extension requires the correct specification of the ideal functionalities to capture the adversarial capabilities. At the very least, the functionalities depend on the consistency of the inputs that they receive and, in many cases, the adversary should be given the possibility to abort the protocol execution. A straightforward view of ideal functionalities can only consider the protocols that either fail or successfully recover all of their inputs, like the canonical functionalities described in Section 3.4.

The main definition of privacy forbids cases where there can be protocol faults that depend on the inputs. This is in line with the intuition, as such faults leak some information about the inputs. However, for many secure computation frameworks, doing optimistic evaluation until some point in the execution (e.g. publishing of the outputs), privacy could be applicable until there are no consistency checks in the protocol and, therefore, no faults. Hence, the input privacy notion is likely useful for protocols that distinguish between the evaluation and verification phase. Possibly, only the verification needs to be secure.

However, to apply similar results, one also needs to consider an actively secure variant of output predictability. For active security, the adversary defines the

[1] Documentation in `https://docs.sharemind.cyber.ee/sharemind-mpc/2023.09/installation/query-interfaces.html`

inputs that are used in the system, and the predictor learns of these changes to the inputs from Env. Hence, the predictor and adversary need some communication, but it needs to be limited for the predictor to be strong enough. On the other hand, it is also possible that in the active model, instead of the general output predictability, it would be more beneficial to consider the correctness of the input-private protocol and only work with outputs that are either correct or corrupted. In addition, in the active model, care needs to be taken to make sure that the adversary cannot affect the protocol execution to use completely wrong inputs. Some similar issues are considered in Chapter 5 as tight scheduling and well-formed protocols of secure computation.

Hence, it would be most straightforward to first consider the extension of the current results to those protocols that are robust against an active adversary as, in this case, there are no faults, and the values in the protocol cannot be modified by the adversary.

A similar framework for actively secure sequential composition with mobile corruption is considered in [66]. Concretely, they focus on secure multiparty computation based on verifiable secret sharing based on commitments. The focus of the paper is on defining a framework for the proactive security model in EasyCrypt [2] [14, 15], and overall they claim a similar structure to the input privacy approach where they prove that a composition of private protocols is private and composing it with a secure protocol (random protocol in their definitions) is secure. In addition, they show that composing a protocol with randomised outputs with a proactively secure protocol gives a proactively secure protocol. Unfortunately, the paper does not provide full details of their definitions and proofs, so further similarities or differences from the approach of this thesis are difficult to analyse.

An extension of input privacy to the case with an active adversary is considered in [118]. Their work considers information-theoretic privacy and focuses on automated proofs of privacy based on the description of the protocol. Their notion of privacy is defined through an ideal functionality that only gives out the inputs of the corrupted parties, which serves the same purpose as the input privacy definition here, where the outputs are not given to the adversary. The outputs are only used by the fragmented environment and are not used to distinguish the real and ideal execution. Hence, the intuition for actively private protocols is similar to input privacy – all that the adversary learns can be learned using only the inputs of the corrupted parties. For active privacy, the latter holds even if the adversary behaves unexpectedly and sends messages that are not following the protocol. They also show that the property of the protocol that they prove is indeed composable, and two actively private protocols can be composed into a new actively private protocol. Their goal is that one should be able to show the privacy of the protocol until the final reconstruction of the outputs or some verification of the

[2] `http://www.easycrypt.info/`

protocol's correctness. However, the final part has to be handled with extra care to achieve security. Note that even if active security is not achieved, there is value in a protocol that achieves passive security with active privacy since it guarantees that even the misbehaving adversary cannot learn more information than the output of the protocol. For example, the output may not leak all details about its inputs. However, in such cases, there is no guarantee about the correctness of the output. Especially the adversary may be able to compute the correct output, but the output received by other parties is incorrect. It seems logical that if an actively private protocol is finished with a consistency check and actively secure opening, then one can also achieve security against an active adversary. While this conclusion is likely, such a result in not proven in [118] nor elsewhere. Hence, it remains an open issue to further study input privacy and its composability for active adversaries.

5. ALGORITHM SECURITY FOR MPC FRAMEWORKS

The research on secure algorithms for MPC often assumes that there are some core protocols, such as secure addition and multiplication, provided by some secure computation framework that can be used to build more complex algorithms. This chapter focuses exactly on these cases where the focus is on some new protocol Π that is built on top of a secure protection domain. In addition to this protocol, the protection domain can also carry out various other computations Π_e. The idea is to extend the current description of the protection domain (Section 3.6) with the new protocol Π that is as secure as some ideal functionality $\mathcal{F}$. It is the main job of the protocol designer to prove that Π is as secure as $\mathcal{F}$, and this chapter defines an abstract execution environment that can be used to make such proofs simpler and only focus on the protocol Π. This seems similar to the goals given by frameworks for composable security. However, the following analysis takes into account that Π and Π_e often share some setup parameters for the storage domains. In addition, Π expects inputs and outputs in some fixed storage domains whereas Π_e itself runs in a more generic context of an environment giving it plain inputs and possibly directing its computations. In addition, this chapter discusses the extendability of protection domains that is required to add the new functionality $\mathcal{F}$ to the already existing protection domain. Section 5.1 gives a more in-depth overview of the basis of the following results.

The core of this chapter focuses on defining the abstract execution model that simplifies the protocol description as well as the environment that needs to be considered in security proofs. Many low-level secure computation protocols are always running in a bigger context of other protocols with the same storage domains. For example, a sorting protocol often uses protocols for comparisons, and therefore, it is reasonable to expect that the comparison functionality can be used with the values in the same storage domain. This can make it very difficult to consider only one protocol in isolation, as its inputs may come from a big set of other existing functionalities in the protection domain. However, for many protection domains, the distribution of the values in the storage domains can be simulated in a manner specified later. If the secure computation framework is simulatable, then it is sufficient to only consider the protocol in an environment that gives public inputs and receives the outputs of the protocol also in public. This significantly simplifies analysing the protocol, as the other functionalities in the protection domain can be forgotten for the analysis. Such an isolated execution is called the abstract execution. The abstract execution itself also does not need to explicitly use the protected versions of the data. Instead, the abstract execution only considers the protocol that is executed and the storage domains for each variable in the program. The adversary can see the values if it has broken the hiding property of the given storage domain, and it can modify the values based on the limited con-

trol property of the storage domain. Hence, the focus of the security proof is on simulating the values that are made public to the adversary. All the rest is covered by the properties of the functionalities and storage domains used to compute the protocol.

Overall, the motivation for the abstract model definition is close to that of the simplified universal composability [44] defined for MPC protocols. The paper [44] also observes that proofs usually do not concern themselves with the full details of the UC framework. However, the abstract model aims to do further simplifications based on reasonable properties of the protocols and functionalities involved in secure multiparty computation. The following states the main result of this chapter.

Theorem 8 (Security in the abstract model is equivalent to security in the hybrid model, informal)**.** *If the used storage domains are hiding and modification aware and the protocol is reasonably built from meaningful local operations and canonical ideal functionalities then security in the abstract execution model is necessary and sufficient for security in the hybrid model.*

5.1. Overview of the Approach

The following sections start from the hybrid execution model and transform it into the abstract execution model. The transformation is split into several bigger steps and sections. Section 5.3 starts with considering generic adversaries against secure multiparty computation and observes that the structure of such protocols limits which actions of the adversary are meaningful. Since the separate protocols of secure computation are only evaluated if also the honest parties have given their inputs, then it is possible to consider a lazy adversary that does not send excessive messages to ideal functionalities. Section 5.4 studies the protocol structure and splits the execution of the protocol to separate machines for storing the state and for controlling the execution. It also merges the memory machines holding the states into one memory machine that is connected to all participants. Section 5.5 continues with simplifying the memory and protocol structure. It observes that honest execution only needs the values and not the protection mechanism and proceeds to make it so that only the adversary uses the protected data representations. It concludes that one only needs to consider an adversary that has some limited control over the memory. The final protocol description relies only on the values and the knowledge about the storage domains that hold these values. By this step, the protocol description is already quite abstract. The final transformations in Section 5.6 simplify the environment and arrive at the abstract execution model.

Section 5.1.1 establishes the conditions that the protocol transformations have to satisfy for the security proofs in the abstract world to also mean that the protocol before transformations is secure. Section 5.1.2 establishes the concrete view of the party using the protection domain to execute the protocol of interest. Sec-

tion 5.2 establishes why a specific protocol in a protection domain can be somewhat isolated from the rest of the protection domain. Thanks to the results there, the description of a protocol can use functionalities that only connect to this protocol and not to the rest of the protection domain. Section 5.2.2 discusses what it means for a secure computation framework to be simulatable so that it can be abstracted away and the analysis can focus on the protocol at hand.

5.1.1. Soundness and Completeness Theorems

As stated before, this chapter builds an abstract execution model. This section shows how and why the abstract model is indeed useful. Namely, if there is a security proof in the abstract model, then there also is a security proof in the initial model, and if there is an attack in the abstract model, then there is an equivalent attack in the real model. In shorter terms, the abstract model can be sound and complete if certain equivalence results specified in this section hold. The following of this section will build up the details of these equivalence results for the concrete case of real and abstract execution of secure multiparty computation protocols. This section is given in very general terms and holds for any type of protocol and environment.

Commonly, the discussion is about a protocol and its ideal implementation, but in the general case, the security proofs consider two protocols Π_1 and Π_2. Let Π_1^* and Π_2^* be the same protocol described in an abstract execution model, $\mathbb{E}$ and $\mathbb{E}^*$ be the sets of original and abstract environments and $\mathbb{A}_1, \mathbb{A}_2$, $\mathbb{A}_1^*, \mathbb{A}_2^*$ be the respective adversaries. The proofs needed to show that the abstract model is sound and complete need to show how one representation of the protocol, environment and adversary can be transformed into the other representation. The proof $\Pi_1^* \geq \Pi_2^*$ can be done in the abstract model and the result also holds in the original model. The transformations are denoted as

$$\begin{array}{lll} \psi : \mathbb{E} \to \mathbb{E}^* & \phi_1 : \mathbb{A}_1 \to \mathbb{A}_1^* & \phi_2 : \mathbb{A}_2 \to \mathbb{A}_2^* \\ \psi^* : \mathbb{E}^* \to \mathbb{E} & \phi_1^* : \mathbb{A}_1^* \to \mathbb{A}_1 & \phi_2^* : \mathbb{A}_2^* \to \mathbb{A}_2 \end{array} , \tag{5.1}$$

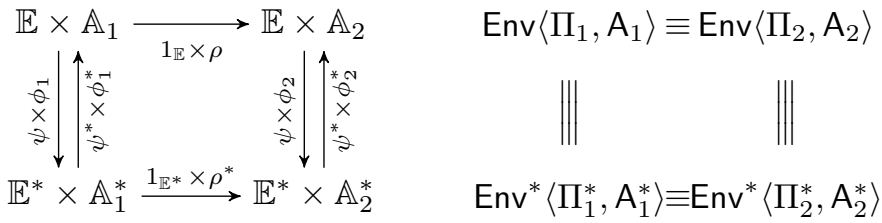
where the star denotes a semi-inverse of the transformation with the same symbol. To be meaningful, these transformations have to satisfy several equivalences

$$\begin{array}{llll} \forall \mathsf{Env} \in \mathbb{E} : & \forall \mathsf{A}_1 \in \mathbb{A}_1 : & \mathsf{Env}\langle \Pi_1, \mathsf{A}_1 \rangle \equiv \psi(\mathsf{Env})\langle \Pi_1^*, \phi_1(\mathsf{A}_1) \rangle & \\ \forall \mathsf{Env}^* \in \mathbb{E}^* : & \forall \mathsf{A}_1^* \in \mathbb{A}_1^* : & \mathsf{Env}^*\langle \Pi_1^*, \mathsf{A}_1^* \rangle \equiv \psi^*(\mathsf{Env}^*)\langle \Pi_1, \phi_1^*(\mathsf{A}_1^*) \rangle & \end{array} \tag{5.2}$$

$$\begin{array}{llll} \forall \mathsf{Env} \in \mathbb{E} : & \forall \mathsf{A}_2 \in \mathbb{A}_2 : & \mathsf{Env}\langle \Pi_2, \mathsf{A}_2 \rangle \equiv \psi(\mathsf{Env})\langle \Pi_2^*, \phi_2(\mathsf{A}_2) \rangle & \\ \forall \mathsf{Env}^* \in \mathbb{E}^* : & \forall \mathsf{A}_2^* \in \mathbb{A}_2^* : & \mathsf{Env}^*\langle \Pi_2^*, \mathsf{A}_2^* \rangle \equiv \psi^*(\mathsf{Env}^*)\langle \Pi_2, \phi_2^*(\mathsf{A}_2^*) \rangle & \end{array} \tag{5.3}$$

$$\begin{array}{llll} \forall \mathsf{Env} \in \mathbb{E} : & \forall \mathsf{A}_2 \in \mathbb{A}_2 : & \mathsf{Env}\langle \Pi_2, \mathsf{A}_2 \rangle \equiv \psi^*(\psi(\mathsf{Env}))\langle \Pi_2, \mathsf{A}_2 \rangle & \\ \forall \mathsf{Env}^* \in \mathbb{E}^* : & \forall \mathsf{A}_2^* \in \mathbb{A}_2^* : & \mathsf{Env}^*\langle \Pi_2^*, \mathsf{A}_2^* \rangle \equiv \psi(\psi^*(\mathsf{Env}^*))\langle \Pi_2^*, \mathsf{A}_2^* \rangle & . \end{array} \tag{5.4}$$

Equivalences (5.2) and (5.3) define relations between the real and abstract models for the two protocols. Equivalence (5.4) defines that the transformations



(a) Relations between protocol environments and adversaries.

(b) Resulting equivalent protocol executions.

Figure 31: Equivalence guarantees and their relations to ρ and ρ^*.

on the environments are inverses in the sense of the capabilities of the respective environments distinguishing the real and abstract execution of Π_2.

Let ρ be the transformation $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ and, respectively, $\rho^* : \mathbb{A}_1^* \to \mathbb{A}_2^*$ is its counterpart in the abstract world. This is the transformation defined in the security proof to transform any adversary against Π_1 to an equivalent adversary against Π_2, showing $\Pi_1 \geq \Pi_2$. The previous constructions from (5.1) with the construction ρ define an equivalence square illustrated in Figure 31 if the relations (5.2)–(5.4) hold. Figure 31a shows how the transformation works and the arrows show the direction of the transformation. The relation $1_{\mathbb{E}}$ is the identity transformation, as the security proofs do not change the environment, but the adversary is usually changed by defining the simulator. If the up and down relations hold, then the existence of ρ implies the existence of ρ^* and vice versa. Note that all the results in Figure 31b are indeed equivalences. The vertical equivalences come from the necessary properties and the horizontal ones come from the security proof, as the respective environment must be unable to distinguish the protocol running with an equivalent adversary where $\mathsf{A}_2 = \rho(\mathsf{A}_1)$ and $\mathsf{A}_2^* = \rho^*(\mathsf{A}_1^*)$. Often, the security proofs consider some restricted adversary, for example, a polynomial-time adversary. Restricted versions of the equivalences can also be considered if ρ is suitable and $\phi_2^* \circ \rho^* \circ \phi_1$ and $\phi_2 \circ \rho \circ \phi_1^*$ preserve these restrictions. The following theorem proves the equivalence results illustrated by Figure 31.

Theorem 9. *Let $\psi : \mathbb{E} \to \mathbb{E}^*$, $\phi_1 : \mathbb{A}_1 \to \mathbb{A}_1^*$ and $\phi_2 : \mathbb{A}_2 \to \mathbb{A}_2^*$ be constructions with semi-inverses $\psi^*, \phi_1^*, \phi_2^*$ that satisfy the equivalence relations* (5.2)–(5.4)*. Then, $\Pi_1 \geq \Pi_2$ for environments $\mathbb{E}$ if and only if, $\Pi_1^* \geq \Pi_2^*$ for environments $\mathbb{E}^*$.*

Proof. This proof is essentially an analysis that establishes that the equivalences in Figure 31 hold.

SOUNDNESS. For soundness, the assumption is that $\Pi_1^* \geq \Pi_2^*$. Hence, there exists $\rho^* : \mathbb{A}_1^* \to \mathbb{A}_2^*$ defined by the security proof such that

$$\mathsf{Env}^*\langle \Pi_1^*, \mathsf{A}_1^* \rangle \equiv \mathsf{Env}^*\langle \Pi_2^*, \rho^*(\mathsf{A}_1^*) \rangle \tag{5.5}$$

for all $\mathsf{A}_1^* \in \mathbb{A}_1^*$ and $\mathsf{Env}^* \in \mathbb{E}^*$. The goal of the proof is to define a suitable ρ. Hence, the starting point is the collection $\mathsf{Env}\langle \Pi_1, \mathsf{A}_1 \rangle$ which has to be equivalent

to $\mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$. This can be derived using first the first equivalence of (5.2), then (5.5), then the second equivalence of (5.3) and finally the first equivalence of equivalence (5.4) as

$$\begin{aligned}
\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle &\equiv \psi(\mathsf{Env})\langle\Pi_1^*, \phi_1(\mathsf{A}_1)\rangle \\
\psi(\mathsf{Env})\langle\Pi_1^*, \phi_1(\mathsf{A}_1)\rangle &\equiv \psi(\mathsf{Env})\langle\Pi_2^*, \rho^*(\phi_1(\mathsf{A}_1))\rangle \\
\psi(\mathsf{Env})\langle\Pi_2^*, \rho^*(\phi_1(\mathsf{A}_1))\rangle &\equiv \psi^*(\psi(\mathsf{Env}))\langle\Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1)))\rangle \\
\psi^*(\psi(\mathsf{Env}))\langle\Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1)))\rangle &\equiv \mathsf{Env}\langle\Pi_2, \phi_2^*(\rho^*(\phi_1(\mathsf{A}_1)))\rangle \ .
\end{aligned}$$

From this derivation, it is clear that ρ can be defined as $\rho = \phi_2^* \circ \rho^* \circ \phi_1$ and it is straightforward to see that $\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle$.

COMPLETENESS For completeness, it is required that if the relation $\Pi_1 \geq \Pi_2$ holds for the real case, then there exists ρ^* proving $\Pi_1^* \geq \Pi_2^*$. If $\Pi_1 \geq \Pi_2$, then there exists $\rho : \mathbb{A}_1 \to \mathbb{A}_2$ such that

$$\mathsf{Env}\langle\Pi_1, \mathsf{A}_1\rangle \equiv \mathsf{Env}\langle\Pi_2, \rho(\mathsf{A}_1)\rangle \ , \tag{5.6}$$

for all $\mathsf{A}_1 \in \mathbb{A}_1$ and $\mathsf{Env} \in \mathbb{E}$. Similarly to the previous half of the proof, the required relations can be used to derive the result as

$$\begin{aligned}
\mathsf{Env}^*\langle\Pi_1^*, \mathsf{A}_1^*\rangle &\equiv \psi^*(\mathsf{Env}^*)\langle\Pi_1, \phi_1^*(\mathsf{A}_1^*)\rangle \\
\psi^*(\mathsf{Env}^*)\langle\Pi_1, \phi_1^*(\mathsf{A}_1^*)\rangle &\equiv \psi^*(\mathsf{Env}^*)\langle\Pi_2, \rho(\phi_1^*(\mathsf{A}_1^*))\rangle \\
\psi^*(\mathsf{Env}^*)\langle\Pi_2, \rho(\phi_1^*(\mathsf{A}_1^*))\rangle &\equiv \psi(\psi^*(\mathsf{Env}^*))\langle\Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*)))\rangle \\
\psi(\psi^*(\mathsf{Env}^*))\langle\Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*)))\rangle &\equiv \mathsf{Env}^*\langle\Pi_2^*, \phi_2(\rho(\phi_1^*(\mathsf{A}_1^*)))\rangle \ .
\end{aligned}$$

This derivation is almost the same as the previous and uses the second equivalence of (5.2), then (5.6), then the first equivalence of (5.3) and finally the second equivalence of (5.4). As a result, a suitable relation ρ^* can be defined as $\rho^* = \phi_2 \circ \rho \circ \phi_1^*$, and it is clear from the derivations, that $\mathsf{Env}^*\langle\Pi_1^*, \mathsf{A}_1^*\rangle \equiv \mathsf{Env}^*\langle\Pi_2^*, \rho^*(\mathsf{A}_1^*)\rangle$ as required. □

This theorem forms the basis for the rest of this chapter of the thesis. The following sections define the transformations ϕ_1, ϕ_2 and ψ with their semi-inverses and prove that the conditions in equivalences (5.2)–(5.4) hold. These transformations take the real protocol description to the abstract description and, from Theorem 9, it is then clear that a security proof defining ρ^* in the abstract world exists if and only if a security proof defining ρ exists in the real world. For specific purposes, Π_1 is the real protocol and Π_2 is the ideal version of it. Due to the equivalences of different ways to specify the ideal functionalities discussed in Section 3.4.3, the real and ideal world descriptions can have a very similar structure. Due to that, ϕ_1 and ϕ_2 are also the same transformations.

The necessary transformations are defined in smaller steps throughout the chapter. First, Section 5.3 limits the set of adversaries that should be considered and defines the first part of ϕ_1 and ϕ_2. Relevant ϕ_1^* and ϕ_2^* are identity functions and the

protocol description is not changed. The environment is not transformed in this step. Hence, ψ and ψ^* are also identity functions. Hence, only the equivalences concerning ϕ_1 and ϕ_2 need to be proven from (5.2) and (5.3).

Section 5.4 separates protocol state and execution and refines how the adversary can modify the messages sent in the protocol. These changes affect the protocol description and adversary. The environment remains the same again. Hence, this section explicitly defines ϕ_1 and ϕ_2 as well as ϕ_1^* and ϕ_2^* and proves the necessary equivalence relations (5.2) and (5.3).

Section 5.5 further decomposes how secure data is stored in the protocol and considers the properties of ideal functionalities to limit the adversary even more. Again, this changes the protocol description and the adversary, hence further detailing ϕ_1 and ϕ_2, as well as ϕ_1^* and ϕ_2^*, and requiring the same proofs as the previous section.

Section 5.6 finally focuses on the environment to simplify the general environment of the protection domain to the simple case that only executes the desired sub-protocol from the protection domain. Hence, this section focuses on defining ψ and ψ^* and requires proving equivalence relations (5.4). However, modifications also affect the adversary's communication with the environment and the protocol. Hence, also relations (5.2)–(5.3) must be shown to hold.

5.1.2. Protocol Description

A protocol Π is executed by participants interacting with the protection domain. The exact specification of the participant is developed throughout this chapter and has to cover the specifics of the protocol execution as well as the adversarial communication. This section introduces the basics used in the following.

A participant $\mathcal{P}_i$ is represented as a collection of two machines $\mathcal{I}_i$ for code interpretation and $\mathcal{Z}_i$ for corruption management as illustrated in Figure 32. Most of the following does not use port numbering, but it is necessary to define the interfaces of the machines that are involved. The interpreter $\mathcal{I}_i$ is responsible for all computations taking part on the party side and for following the protocol to send out calls to the next sub-protocols. For each input that $\mathcal{I}_i$ receives, it may send out any number of follow-up messages, but it cannot start or compute anything more before receiving the next input from Π_{e} or some response from $\mathcal{F}_p$ that it has called.

Port pairs $1,\ldots,k$ between $\mathcal{I}_i$ and $\mathcal{Z}_i$ are for forwarding messages to and from subprotocols $\mathcal{F}_1,\ldots,\mathcal{F}_k$. The zeroth pair is for adversarial actions and the $k+1$ port pair is for communicating with Π_{e} that is calling out the protocol executed by $\mathcal{P}_i$. All these ports are matched by the ports $\mathcal{Z}_i$ used to communicate with the protection domain or A. Hence, $\mathcal{Z}_i$ acts like a switch. Port pair $k+2$ in $\mathcal{I}_i$ is intended for receiving the setup information.

If $\mathcal{Z}_i$ is not corrupted, then it simply forwards the communication between the matching ports, interaction on port $0 \leq m \leq k+1$ for $\mathcal{I}_i$ is sent to $m+k+2$ to

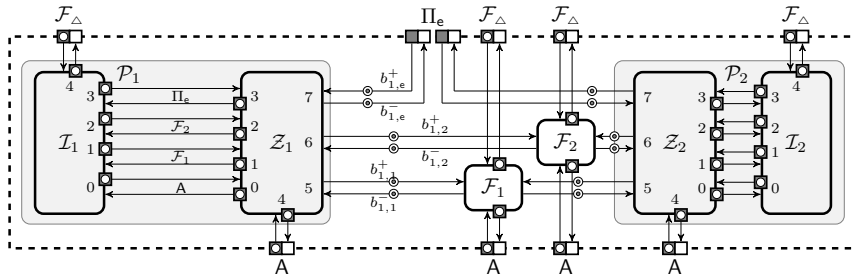


Figure 32: The structure of parties in a two-party protocol interacting with functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$.

the outside of party $\mathcal{P}_i$ and vice versa. Every time $\mathcal{Z}_i$ receives a message from outside, it writes it to $\mathcal{I}_i$ and clocks the channel. When receiving a message from $\mathcal{I}_i$, $\mathcal{Z}_i$ gives control back to $\mathcal{I}_i$ after writing the message to the respective outgoing buffer. The control is given back using an empty message of the same port pair as used by $\mathcal{I}_i$. $\mathcal{I}_i$ includes an end token when it does not need to get the control back. $\mathcal{Z}_i$ gets the control and can then send inputs from $\mathcal{I}_i$ to A. If $\mathcal{Z}_i$ is corrupted, then A can order it to write any message to any of its output ports. All messages arriving at $\mathcal{Z}_i$ are written to A together with the port label. Messages from $\mathcal{I}_i$ are grouped together for all messages between sending the input to $\mathcal{I}_i$ and $\mathcal{I}_i$ sending the message with the end token. A can issue a REVEAL call to $\mathcal{I}_i$ through $\mathcal{Z}_i$. As a response, A receives the internal state of $\mathcal{I}_i$.

Each message sent between $\mathcal{P}_i$ and $\mathcal{F}_p$ is a triple (t_1, t_2, m). Here, t_1 is the instance of the protocol Π in $\mathcal{P}_i$ and t_2 is the instance of $\mathcal{F}_p$. As this protocol acts analogously to $\mathcal{F}_p$ for the protocol Π_e, then for a message between $\mathcal{P}_i$ and Π_e, t_2 specifies the instance of the protocol Π and t_1 the instance of the super-protocol Π_e. The tags t_1 and t_2 are leaked by the leaky buffers, whereas m is kept private and authentic. For simplicity, the leaky buffers can be denoted as shorthands where $b^+_{i,p}$ is outgoing from $\mathcal{P}_i$ and $b^-_{i,p}$ is incoming to the party for functionality $\mathcal{F}_p$. From the viewpoint of Π_e, the protocol Π is similar to the role of $\mathcal{F}$ inside Π. Hence, the buffers between Π_e and $\mathcal{P}_i$ are denoted differently, here $b^-_{i,e}$ is the reply to Π_e and $b^+_{i,e}$ is the input for $\mathcal{P}_i$.

5.2. Security of Protection Domain Extensions

The protection domains were introduced in Section 3.6, and the security of the protection domain was given as Definition 17. Adding an algorithm to a secure protection domain means describing the ideal functionality corresponding to the algorithm and then extending the set of ideal functionalities used by the domain. For a modular protection domain (Definition 16), the new protocol is therefore $\Pi\langle\Pi_1,\ldots,\Pi_k\rangle$ where $\Pi_1,\ldots,\Pi_k$ are the protocols available in the protection domain. The protocol designer proves that the new protocol is as secure as some ideal functionality, establishing $\mathcal{F}_\triangle\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle\rangle \geq \mathcal{F}_\triangle\langle\mathcal{F}\rangle$. The security def-

inition of the protection domain establishes $\mathcal{F}_{\triangle}\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\Pi\langle\mathcal{F}_{\mathsf{pd}}\rangle\rangle$, for any protocol Π allowed by the protection domain. Hence, it is left to prove that $\mathcal{F}_{\triangle}\langle\Pi\langle\mathcal{F}_{\mathsf{pd}}\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\mathcal{F}\rangle$. Hence, the main proof is in the hybrid model, where for the modular protection domain, $\mathcal{F}_{\mathsf{pd}}$ is defined as a collection of $\mathcal{F}_1,\ldots,\mathcal{F}_k$ and the protocol is therefore described as $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$.

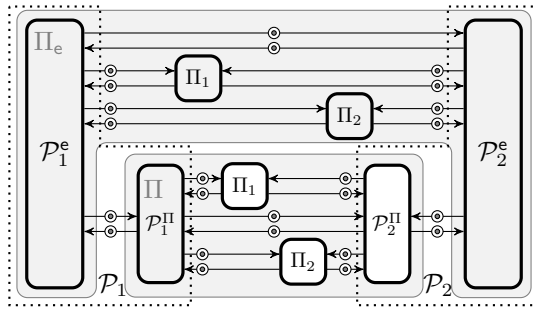
Once it is proven that $\mathcal{F}_{\triangle}\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\mathcal{F}\rangle$, the next question is, if the collection $\mathcal{F},\mathcal{F}_1,\ldots,\mathcal{F}_k$ is a valid definition for $\mathcal{F}_{\mathsf{pd}}$. The proof that Π is as secure as $\mathcal{F}$ is done in isolation from the rest of the computations that may happen in the protection domain. Hence, the main question is to consider if a list of protocols $\Pi,\Pi_1,\ldots,\Pi_k$ is a secure protection domain for a class of protocols $\mathbb{P}$ if $\Pi_1,\ldots,\Pi_k$ was. In order to prove it, one must consider the compound protocols $\Pi_* \in \mathbb{P}$ for $\Pi_*\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle,\Pi_1,\ldots,\Pi_k\rangle$. The protocols Π_* and Π can be merged to form a new protocol $\Pi_{**}\langle\Pi_1,\ldots,\Pi_k,\Pi_1,\ldots,\Pi_k\rangle$. However, the new compound protocol has access to two copies of each of the original protocols, and therefore, it is not clear if it is as secure as $\mathcal{F}_{\mathsf{pd}}$. Firstly, define such property of the protection domain as extendability (Definition 30). Under reasonable assumptions discussed in Section 5.2.1, a secure protection domain is extendable.

Definition 30 (Extendable protection domain)**.** A list of protocols $\Pi_1,\ldots,\Pi_k$ with a shared setup $\mathcal{F}_{\triangle}$ is a securely extendable protection domain if the list of protocols with duplicates $\Pi_1,\Pi_1,\ldots,\Pi_k,\Pi_k$ with a shared setup $\mathcal{F}_{\triangle}$ is also a secure protection domain.

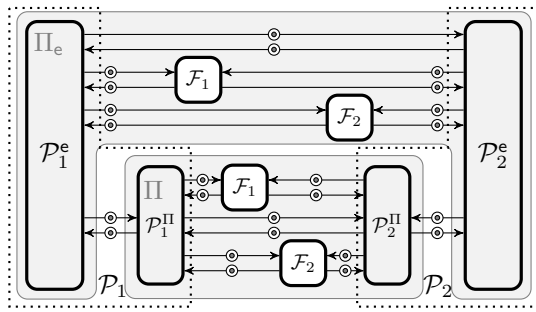
Note that a modular protection domain contains ideal functionalities as well as the secure setup functionality. However, its main property is that it can be considered mainly as the collection of its internal functionalities that are ideal functionalities as specified in Definition 14. This may not be true for all possible secure computation frameworks. However, this thesis and the following theorem focus on only such cases.

Theorem 10. *Let $\Pi_1,\ldots,\Pi_k$ be a securely extendable modular protection domain with a shared setup $\mathcal{F}_{\triangle}$ for protocols $\mathbb{P}$. Then $\Pi\langle\Pi_1,\ldots,\Pi_k\rangle,\Pi_1,\ldots,\Pi_k$ is a secure protection domain for compound protocols $\mathbb{P}$ provided that $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ is a secure protection domain with modular representation with one functionality $\mathcal{F}$ for the set of compound protocols $\mathbb{P}_* = \{\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle : \Pi_{\mathsf{e}} \in \mathbb{P}\}$ and coherent adversaries.*

Proof. The desired signature for the extended protection domain is $(\mathbb{E},\mathbb{P},\mathcal{F},\mathcal{F}_1,$ $\mathcal{F}_2,\ldots,\mathcal{F}_k)$ and $\Pi_{\mathsf{e}}\langle\cdot\rangle \in \mathbb{P}$ is one of the compound protocols for this protection domain. Let $\mathbb{A}_1$ be the class of adversaries against the real implementation of the extended protection domain $\Pi_{\mathsf{e}}\langle\Pi\langle\Pi_1,\ldots,\Pi_k\rangle,\Pi_1,\ldots,\Pi_k\rangle$. By definition, $\Pi_1,\ldots,\Pi_k$ can be extended to a protection domain that doubles all the protocols. Any compound protocol Π_{e} interacting with the extended protection domain can be restructured to separate the protocol Π from the rest of Π_{e} as in Figure 33a. The protocol is executed by parties $\mathcal{P}_i$. Let $\mathcal{P}_i^{\mathsf{e}}$ be executing the rest of the Π_{e} and $\mathcal{P}_i^{\Pi}$ be the parties executing Π.



(a) Ways to think of composed protocol $\Pi_e\langle\Pi\langle\Pi_1,\Pi_2\rangle,\Pi_1,\Pi_2\rangle$.



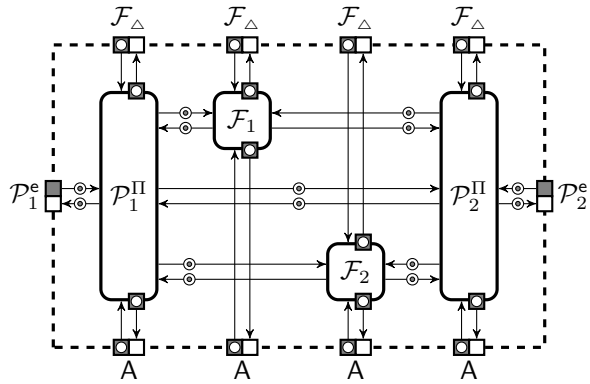
(b) Extended hybrid collection $\Pi_e\langle\Pi\langle\mathcal{F}_1,\mathcal{F}_2\rangle,\mathcal{F}_1,\mathcal{F}_2\rangle$.

Figure 33: Two-party protection domain extension.

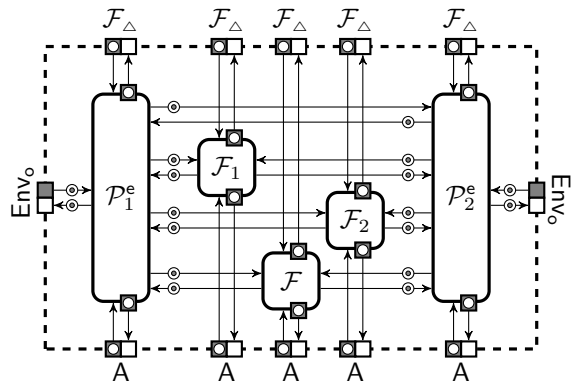
Separating the party into two components introduces a new leaky buffer that is connecting $\mathcal{P}_i^e$ to Π. For the case of honest parties, this introduces a new adversary capability to control the inner timing of party $\mathcal{P}_i$. Now the adversary can directly control when the execution of Π starts. By definition of the protocol execution, Π is started by a party $\mathcal{P}_i$ when it receives or computes the necessary inputs for the protocol and reaches the relevant call in its program. Π sends the results back when it finishes computing any output. However, the only effect of separately clocking Π is equivalent to that of clocking the buffers to and from $\mathcal{F}_p$ that need to finish before Π is called or before Π produces outputs. A coherent adversary always corrupts both machines representing $\mathcal{P}_i$ at the same time and assumes total control over that party. Hence, this leaky buffer is equivalent to the joint machine in the case of corrupted parties.

The security definitions of the functionalities give $\Pi_p \geq \mathcal{F}_p$. Hence, it is allowed to replace each real protocol with the ideal implementation to define the hybrid model. This is shown in Figure 33. The theorem assumption stated that $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ is a secure protection domain with one functionality $\mathcal{F}$ for the set of outer protocols Π_e. This allows us to do the steps in Figure 34 to replace Π with $\mathcal{F}$. It remains to show that these substitutions are valid and that the resulting configuration is a valid protection domain for protocols Π_e.

Denote by $\mathbb{E}_{pd}$ the outer environments that are interacting with the whole pro-



(a) $\Pi\langle\mathcal{F}_1,\mathcal{F}_2\rangle$ running in the context of $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle \in \mathbb{P}_*$.



(b) $\mathcal{F}_{\mathsf{pd}}$ containing $\mathcal{F}_0,\mathcal{F}_1,\mathcal{F}_2$.

Figure 34: Hybrid protocol and the respective ideal functionality $\mathcal{F}$ inserted into the protection domain.

tection domains. In the hybrid model, it is then necessary to show that, for environments $\mathbb{E}_{\mathsf{pd}}$ and adversaries $\mathbb{A}_2$, we have

$$\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle,\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F},\ldots,\mathcal{F}_k\rangle\rangle \ .$$

One copy of the functionalities $\mathcal{F}_1,\ldots,\mathcal{F}_k$ can be pushed into Π_{e} to define a new compound protocol $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle \in \mathbb{P}_*$. $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ only calls the sub-protocol $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ (Figure 34a), as in Figure 33b. By the theorem assumption, $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\cdot\rangle$ is a valid compound protocol for $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ meaning that $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ is as secure as $\mathcal{F}$ when used in this protocol. By the definition of secure protection domain, we then have

$$\mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\rangle\rangle \geq \mathcal{F}_{\triangle}\langle\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\mathcal{F}\rangle\rangle \ .$$

Finally, the functionalities $\mathcal{F}_1,\ldots,\mathcal{F}_k$ can be again considered as simply being outside of Π_{e} and hence, the $\Pi_{\mathsf{e}}\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\langle\mathcal{F}\rangle$ is equivalent to $\Pi_{\mathsf{e}}\langle\mathcal{F},\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$

as in Figure 34b, which proves the theorem. The environment in this figure is the outer environment introduced in Section 3.6.2 as the environment against the protection domain. □

This theorem essentially specifies that it is sufficient to prove the security of $\Pi\langle\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle$ in the context of compound protocols $\mathbb{P}_*$ that the protection domain that is extended can run. The protocols can use the functionalities $\mathcal{F}_1,\ldots,\mathcal{F}_k$ that the protection domain contains. In practice, this means that the class of environment in which the security of this protocol must be proven is the combination of the compound protocols $\mathbb{P}_*$ and general environments $\mathbb{E}_{\mathsf{pd}}$ (Section 3.6.2) giving inputs and receiving outputs from the parties running the protocols $\mathbb{P}_*$.

5.2.1. Extendable Protection Domains

Definition 30 specified that a protection domain is extendable if the collection of protocols, where each original protocol is duplicated, is also a secure protection domain. This section considers the case when it is a secure protection domain for functionally the same classes of protocols. It is most straightforward to consider the case when the two collections are equivalent to each other.

Lemma 10. *A secure modular protection domain with shared setup $\mathcal{F}_\triangle$ for protocols $\mathbb{P}$ is securely extendable to a secure modular protection domain with shared setup $\mathcal{F}_\triangle$ for protocols $\mathbb{P}_*$ if the following conditions hold.*

- *$\mathcal{F}_\triangle$ gives consistent setup information.*
- *One instance of a protocol Π_p of the protection domain is independent both in functionality and adversary capabilities of the other instances of the same protocol (other than the shared setup information).*
- *The protocols $\mathbb{P}_*$ are functionally the same as $\mathbb{P}$, but each new instance of a sub-protocol call chooses which copy of the sub-protocol to use and uses disjoint sets of instance tags for the two copies of the same functionality.*

Proof. Assume, by contradiction, that such a protection domain is not securely extendable. Hence, there exists an environment Env and adversaries A and $\rho(\mathsf{A})$ such that that A running with $\Pi_1,\ldots,\Pi_k,\Pi_1,\ldots,\Pi_k$ is distinguishable from $\rho(\mathsf{A})$ running with the protection domain $\mathcal{F}_{\mathsf{pd}}$ consisting of $\mathcal{F}_1,\ldots,\mathcal{F}_k,\mathcal{F}_1,\ldots,\mathcal{F}_k$. Note that the environment always communicates with the parties $\mathcal{P}_i$ running the protocol. Therefore the interface to the environment is unchanged by the extension. Adversary A can then be turned into an adversary B against the initial protection domain with protocols $\Pi_1,\ldots,\Pi_k$. The difference between the two collections is that A has two copies of each functionality that it can interact with, and, therefore, it also has separate leaky buffers to clock to either copy. To use adversary A against the initial protection domain without modifying its outcome, there must be a simulator that can simulate the run of every single functionality $\mathcal{F}_p$ as two copies of it. By the conditions in the theorem, this is straightforward to do. First, the two sets of protocols allowed by the initial and extended protection domains

are functionally the same. Hence, a protocol that runs a single copy can be simulated as executing with two separate copies of functionalities. The simulator can see all the leaks in the buffers to and from $\mathcal{F}_p$ and can distribute these to two simulated buffers as the two buffers contain a disjoint set of tags. Since messages with one instance tag always go to one copy of the functionality, it is also easy to distribute the messages like they are in two different buffers and manage the clocking of messages in these buffers. The adversary actions can safely be done in a single copy of the functionality as they, by definition, only affect the given protocol instance. Hence, the adversary A running with the simulated version of the protection domain $\Pi_1, \ldots, \Pi_k$ has the same view as when running with the extended protection domain. Hence, the existence of such adversary A gives rise to an adversary B against the original protection domain.

The adversary $\rho(A)$ can undergo a similar simulation to be turned into an adversary $\rho(\mathsf{B})$ against the protection domain with $\mathcal{F}_1, \ldots, \mathcal{F}_k$. By definition of A and $\rho(\mathsf{A})$ can distinguish the extended protection domains and, therefore, the existence of B and $\rho(\mathsf{B})$ invalidates the assumption that the original protection domain was a secure modular protection domain. □

The conditions in Lemma 10 are simple to enforce for any protocol in $\mathbb{P}_*$ derived from $\mathbb{P}$, and this document already assumes a consistent setup inside the protection domain. Hence, the main restriction posed by this lemma for the rest of this chapter is the natural requirement that the protocols Π_p are such that they do not share states other than the setup across instances.

5.2.2. Simulatable Protection Domains

Usually, the overall simulation strategy for a real implementation of a secure computation framework specified as an ABB is very simple. The simulator runs a simulated setup phase and then plays the roles of all parties in the protocol and interacts with the adversary the same as the real protocol would. The simulator just uses dummy inputs for all inputs from the honest parties and relies on the hiding property of the used data storage. The tricky parts of such simulation are extracting the inputs of the corrupted parties so that the simulator can give these to the ABB as inputs and making sure that if the protocol succeeds, then the simulated protocol gives the same output as the one the simulator receives from the ABB. The decision of success or failure is made based on the success or abort of the simulated protocol. If the protocol tolerates adaptive corruption, then extra care must be taken so that the simulated view can always be modified to accommodate the correct values for the newly corrupted parties as highlighted in [56].

The same simulation strategy is successful when the trusted setup and the full protection domain are considered as the ideal functionality corresponding to some real implementation. However, the strength of the formalisation in Chapter 3 is its modularity and the possibility to only consider some sub-protocol inside the protection domain and to extend the protection domain with this functionality. It

is easier to consider the protocol Π in isolation from Π_e if the Π_e can be simulated.

In the following transformations, simulatability is required in two places. Firstly, consider the protection domain inside the hiding game for the storage domain in Section 5.5. Secondly, to simplify the execution environment and embed the Π_e into $\mathsf{Env}_{\mathsf{pd}}$ in Section 5.6. The main difficulty is that Π and Π_e share the setup parameters and that, in the general case, the values moving between Π and Π_e are in some hiding protection domain rather than the plain values output by the ABB. Due to the shared setup, the straightforward simulation method of running the setup phase and using these parameters for simulating the work of Π does not work because, in many cases, the adversary can notice the difference in parameters in Π and Π_e. Hence, the following relies on the fact that the execution of Π_e is simulatable without specifying the simulation strategy.

Simulatability of protection domain execution is defined through the collections in Figure 35. The idea is that the simulator Sim has to play the part of Π_e in a setting where it knows all the plain inputs. Therefore, Sim can execute the same program as Π_e in public and compute the correct values that are given to Π or $\mathsf{Env}_{\mathsf{pd}}$. However, the simulator has to also be able to create a corrupted view of the stored data. The simulator has the setup parameters that are public or known to the corrupted parties, but it does not have the private parameters of the honest parties. However, the protocol Π needs to get inputs that are correct with respect to the real parameters. Hence, there must be a valid sharing functionality $\mathcal{S}$ to generate the input representation for Π. Similarly, the outputs from Π come in some protection domain and need to be reconstructed by $\mathcal{R}$ to be used inside the simulator.

However, simply sharing and reconstructing is not sufficient to unify the representation that the private data has for the values that appear both in Π and Π_e. For example, if the adversary knows that the corrupted party in Π outputs a secret value x, then it expects the party in Π_e to also receive x exactly. Hence, both $\mathcal{S}$ and $\mathcal{R}$ functionalities are slightly enhanced to support the simulation and distribute the shares of the corrupted parties.

$\mathcal{S}_e^{\mathsf{Sim}}$ contains the sharing functionalities for all storage domains needed in Π. More precisely, Sim can send a value to $\mathcal{S}_e^{\mathsf{Sim}}$ that shares this value. Upon receiving the value and label pair (ℓ_x, x) from Sim, $\mathcal{S}_e^{\mathsf{Sim}}$ shares the value x and sends the shares of the corrupted parties back to Sim. Sim also stores all shares and labels. Sim can then send forwarding commands $(forward, \ell_x, \mathcal{P}_i)$ upon which Sim writes the share of $\mathcal{P}_i$ to the buffer to $\mathcal{P}_i$ in Π. Hence, Sim can learn corrupted shares before they are sent to Π but is only allowed to query values that are really intended to be sent to Π. If a new party $\mathcal{P}_i$ becomes corrupted, then Sim sends this party identifier i to $\mathcal{S}^{\mathsf{Sim}}$. Sim responds with all the shares generated for this party so far.

The functionality $\mathcal{R}$ is extended similarly to $\mathcal{S}$. The functionality $\mathcal{R}_e^{\mathsf{Sim}}$ also receives the set of corrupted parties from Sim and stores all values it gets from Π. If it collects all shares needed for reconstructing a value, then it sends the value to

Sim. If it receives a share of a corrupted party from Π or learns that a new party $\mathcal{P}_i$ is corrupted from Sim, then it sends the shares of $\mathcal{P}_i$ to Sim.

Most of the buffers drawn in Figure 35 represent a set of buffers. $\mathcal{F}_{\triangle}$ has buffers to each machine in Π and Π_{e} that need setup data. In addition, all leaky buffers are there for each participant in Π and Π_{e}, and the adversary has connections to all parties and functionalities that it usually has inside Π and Π_{e}. Sim has the same interface for the adversary as Π_{e}.

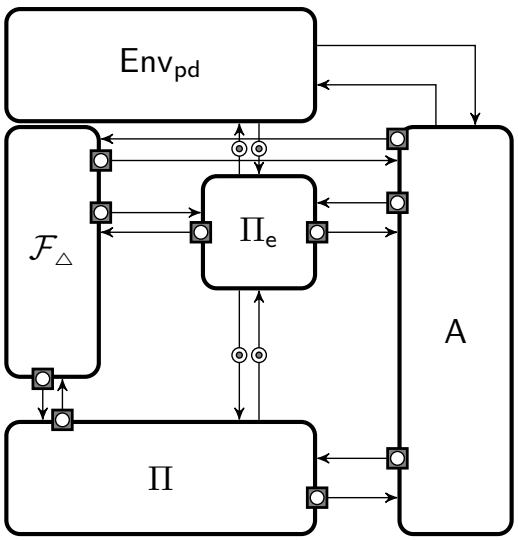
Definition 31 (Simulatable protection domain)**.** A protection domain is simulatable if there exists a simulator Sim such that the collections in Figure 35 with $\mathcal{S}_{\mathsf{e}}^{\mathsf{Sim}}$ and $\mathcal{R}_{\mathsf{e}}^{\mathsf{Sim}}$ specified above are indistinguishable.

The first occurrence where simulatability is needed is in Lemma 13. Note that some of the following might be simplified if simulatability is taken as a requirement for the protection domain. However, simulatability is also a complex notion and it is useful to derive more explicit constraints for protocols Π by not assuming simulatability where it is not crucial.

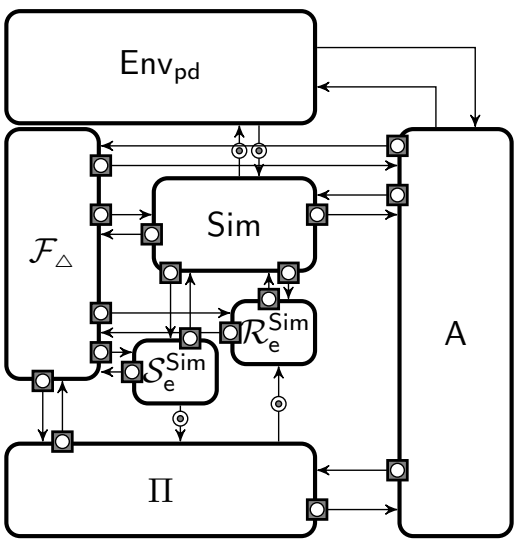
Definition 50 in Section 5.6 defines a simpler version of simulatability called embeddability. Embeddability is used in a setting where only the public outputs of the computation are considered and the protocol Π_{e} is part of $\mathsf{Env}_{\mathsf{pd}}$ that also has all the plain inputs. Hence, the input-output simulation part is trivial. Any simulatable protection domain is also embeddable. In addition, the definition depends on Π, but in the basic case, simulatability can be proven for a set of protocols executed by Π if one assumes that all functionalities of the protection domain are also used by Π.

The main difference between the simulation and the real execution is that the simulator only knows the setup parameters known to the corrupted parties. The question of simulatability, therefore mainly focuses on the question of being able to simulate the computations without knowing the private setup parameters but in a manner that is indistinguishable from the setting with real parameters. Hence, if there are no private parameters, then the simulator can run a copy of the Π_{e} almost honestly because it can execute $\mathcal{S}$ and learn the shares of all parties. The only difference from a full honest execution is that Sim has to be able to adapt shares that are generated by $\mathcal{S}_{\mathsf{e}}^{\mathsf{Sim}}$ or received from $\mathcal{R}_{\mathsf{e}}^{\mathsf{Sim}}$ to use as the simulated values of the corrupted parties in an execution that otherwise follows the honest execution of Π_{e}.

If there are private parameters, then the simplest case for simulation is the one where the simulator still behaves as the honest protection domain but with simulated parameters. However, the simulated parameters must be generated so that the real and simulated parameters of the corrupted parties are the same. In the case of adaptive corruption, the parameters must be adjustable to recompute the simulated parameters of the honest parties when new parameters of the corrupted parties become available. This is similar to the patching defined in [56] needed for the simulation of other values in the protocol in case of adaptive corruption. How-



(a) Real protection domain execution.



(b) Simulated protection domain execution.

Figure 35: Simulatability of Π_{e}. Drawn buffers represent a set of buffers between parties and functionalities in protocols Π and Π_{e}.

ever, the definition does not restrict the simulator and other simulation techniques could be used.

The basic observation of this definition is that the shares of the corrupted party should not leak the private parameters of the honest party. Similarly, the values sent to Env should not reveal the setup parameters.

5.3. Simplified Message Scheduling and Lazy Adversaries

This section shows that it is natural to restrict the adversary interactions with a protocol Π if the latter satisfies some natural properties. In fact, all adversarial actions can be accomplished by only modifying the state of the corrupted parties. It is possible to allow the parties to behave honestly themselves and just modify the messages that the parties send out in the protocol execution.

In more formal terms, this section defines a construction ϕ_{lazy} such that

$$\forall \Pi \in \mathbb{P}_{\mathsf{bb}} : \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \forall \mathsf{A} \in \mathbb{A}_{\Pi,\mathsf{Env}} : \quad \mathsf{Env}\langle \Pi, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \Pi, \phi_{lazy}(\mathsf{A}) \rangle \quad , \quad (5.7)$$

where $\mathbb{P}_{\mathsf{bb}}$ is the set of protocols that use the ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ of the protection domain in black-box manner and $\mathbb{E}_{\Pi}$ is the set of compatible environments. The protocols and environments also have to jointly satisfy the assumptions leading to Theorem 11 in this section. Only the adversary is changed, but the new adversary $\phi_{lazy}(\mathsf{A})$ is also a generic adversary. Hence, the semi-inverse of ϕ_{lazy} is an identity function.

5.3.1. Tight Message Scheduling

The protocol modelling in the given formalism of protection domains does not guarantee that a message is delivered or in which order messages are delivered if several of them are sent out on the same buffer. However, protocols Π_p and their respective ideal specifications $\mathcal{F}_p$ can limit how much the adversary can affect the delivery of these messages. In addition, communication between $\mathcal{P}_i$ and $\mathcal{F}_p$ is necessary for party $\mathcal{P}_i$ to know that $\mathcal{F}_p$ has received enough messages to proceed with the execution. In theory, sending a stream of messages $x_1, \ldots, x_\ell$ has no benefit over sending them as one message because execution only proceeds if all inputs are gathered. Hence, for simplicity, the following assumes that if there are inputs that are sent together or if $\mathcal{F}_p$ sends many outputs out in one go to one recipient, then they are delivered as one message. This is specified so that either party only sends a new message in a protocol instance t if it has received a reply from the receiver of the message. Hence, there is a clear time point when a sender knows that its input to the protocol has been fixed and cannot be modified.

Secondly, the modelling of the protection domain does not limit the set of input and output parties of a functionality $\mathcal{F}_p$. Hence, a party may receive an output from a functionality even if it did not send any inputs. In principle, this could be a desired property, as the output may indeed be necessary to some party that

does not have meaningful inputs to a protocol. However, it is easier to describe the interpreter $\mathcal{I}_i$ of a party that does not receive unexpected messages and also, in practical protocols, a party should have some knowledge about participating in a protocol. Hence, it is reasonable to assume that all parties provide some input to a protocol $\mathcal{F}_p$ before receiving any outputs. This input may be a dummy input just showing the readiness for receiving outputs. As a side effect, all parties must agree on the instance of the protocol $\mathcal{F}_p$.

These ideas are formalised as the tight scheduling property of a protocol. Note that it still allows many rounds of interactions with $\mathcal{F}_p$ and the next inputs may depend on previously received outputs. Note that a round here applies to the notion of communication between a party $\mathcal{P}_i$ and $\mathcal{F}_p$. The rounds of interaction may be different for different parties. If the input and output signatures of all steps of the ideal functionality include all parties, as is common for secret-sharing-based secure computation, then the rounds of different parties are also aligned. In addition, a round from the perspective of $\mathcal{F}_p$ can send messages to one or more parties.

Definition 32 (Tight message scheduling)**.** An ideal functionality $\mathcal{F}_p$ has tight message scheduling if it does not accept nor send two consecutive messages from/to $\mathcal{P}_i$ for the same protocol instance. $\mathcal{P}_i$ uses functionality with tight scheduling if it does not try to send two consecutive messages to the same functionality. Additionally, $\mathcal{P}_i$ must send the first message before receiving anything from $\mathcal{F}_p$ and both $\mathcal{F}_p$ and $\mathcal{P}_i$ must know when the other stops sending messages for a given protocol instance.

For canonical ideal functionalities, it is clear that they have to receive some message before they can start. Tight scheduling restricts them to not send outputs to parties that did not send any inputs. However, it is possible not to give outputs to all parties that sent inputs. The need for a clear end of the protocol is necessary for protocols where the number of rounds may vary. However, this can easily be added to any protocol implementation as an explicit end message. The end is clearly fixed if the number of rounds of interactions is known by the definition of the functionality. Hence, this requirement is not limiting the set of protocols that can be considered but gives some structure that can be used when discussing the message exchanges in protocols.

5.3.2. Semi-Simplistic Adversaries

The previous section considered tight scheduling, which relates the stream of honest messages and the behaviour of the functionality $\mathcal{F}_p$. The generic adversary can order the corrupted party to send any messages and can, therefore, also affect the execution of $\mathcal{F}_p$ in various ways. In addition, for a corrupted party in general, the interpreter $\mathcal{I}_i$ becomes irrelevant since the adversary can process all incoming messages itself. However, it is more straightforward to think of the protocol execution when it can be followed using the protocol specification in $\mathcal{I}_i$. This section

defines a semi-simplistic adversary that always keeps the $\mathcal{I}_i$ running honestly and in a timely manner, whereas it can still order the corruption module $\mathcal{Z}_i$ to send any message and ignore the values from $\mathcal{I}_i$.

Definition 33 (Semi-simplistic adversary)**.** An adversary is semi-simplistic if it fulfils the following conditions for all corrupted parties $\mathcal{P}_i$.

(a) The adversary clocks any outgoing buffer $b^+_{u,p}$ or $b^-_{u,\mathsf{e}}$ and any incoming buffer $b^-_{j,p}$ or $b^+_{j,\mathsf{e}}$ for honest $\mathcal{P}_j$ only when all incoming buffers $b^-_{i,p}$ or $b^+_{i,\mathsf{e}}$ to corrupted parties $\mathcal{P}_i$ are empty.

(b) Upon receiving a message from $\mathcal{Z}_i$ that comes from $\Pi_\mathsf{e}, \mathcal{F}_1, \ldots, \mathcal{F}_k$, the adversary immediately orders $\mathcal{Z}_i$ to forward it to $\mathcal{I}_i$ on the port matching the input port and without modifications.

(c) The adversary can fetch the state of $\mathcal{I}_i$, using REVEAL instruction forwarded to $\mathcal{I}_i$ by $\mathcal{Z}_i$.

(d) The adversary can send arbitrary messages to $\Pi_\mathsf{e}, \mathcal{F}_1, \ldots, \mathcal{F}_k$, i.e., anything can be written to output ports $p \in \{k+2, \ldots, 2k+2\}$ of $\mathcal{Z}_i$. The adversary gives no other orders to $\mathcal{Z}_i$.

(e) The adversary is coherent.

Note that condition (b) means that if A receives the given message then its only action is that it sends the message to $\mathcal{Z}_i$ to forward this message to $\mathcal{I}_i$ and it clocks this message to $\mathcal{Z}_i$. Hence, $\mathcal{Z}_i$ becomes the new active machine and can indeed send the message to $\mathcal{I}_i$. Conditions (a) and (b) formalise that the interpreter $\mathcal{I}_i$ always receives all messages intended for $\mathcal{P}_i$. In addition, corrupted $\mathcal{I}_i$ receives messages written by $\mathcal{F}_p$ in one activation before the honest interpreters receive the respective messages. In other words, the incoming buffers from $\mathcal{F}_p$ to corrupted parties are kept empty. Conditions (c) and (d) reflect that the adversary does not modify the state of $\mathcal{I}_i$. Hence, $\mathcal{I}_i$ is running honestly. Condition (e) mainly simplifies further discussions as any generic adversary can be replaced by a coherent adversary, as shown in Lemma 1.

Lemma 11. *For any adversary, there exists a semi-simplistic adversary achieving the same goal with unbounded computational overhead.*

Proof. Let A be the initial adversary and let A^* be the semi-simplistic adversary that internally runs A. The proof idea is illustrated in Figure 36. From Lemma 1, we know that it is sufficient to only consider coherent A.

ADVERSARIAL BEHAVIOUR. If a party $\mathcal{P}_i$ is not corrupted, then A^* just follows the clocking of A for all buffers connected to $\mathcal{P}_i$. When A corrupts $\mathcal{P}_i$, then so does A^*. A^* also immediately sends the REVEAL command to learn the internal state of $\mathcal{I}_i$ and creates a simulated copy $\mathcal{I}^*_i$ of $\mathcal{I}_i$ with the same state.

When $\mathcal{Z}_i$ sends a message to A^* that is received from Π_e or $\mathcal{F}_p$ and, therefore, intended for $\mathcal{I}_i$, then A^* orders $\mathcal{Z}_i$ to send this to $\mathcal{I}_i$ without changes. If $\mathcal{I}_i$ sends a new message, then $\mathcal{Z}_i$ collects these until $\mathcal{I}_i$ finishes and gives the collected messages to A^* who ignores them.

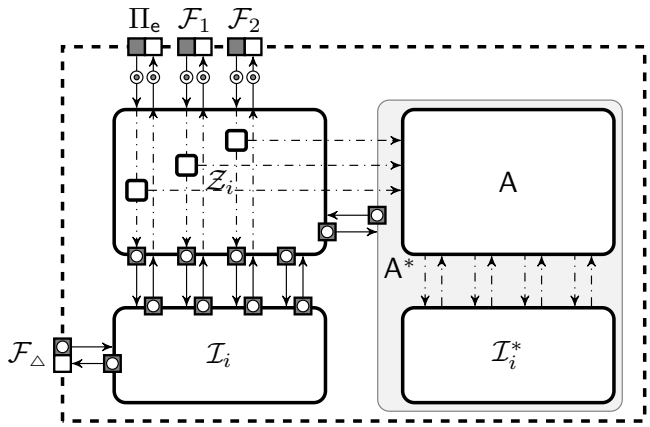


Figure 36: The construction of a semi-simplistic adversary A^* using a generic coherent adversary A and a copy of the honest interpreter. Dashed lines illustrate the movement of the messages, whereas A^* manages the timing and actual appearance of these messages for the buffers connected to A.

Then, A^* can pass the original input to A as a message from $\mathcal{Z}_i$. Whenever A orders $\mathcal{Z}_i$ to send anything to $\mathcal{I}_i$, A^* redirects this to $\mathcal{I}_i^*$. A^* passes all replies from $\mathcal{I}_i^*$ to A. If A orders $\mathcal{Z}_i$ to send a message to $\mathcal{F}_p$ or Π_e then A^* forwards this message to $\mathcal{Z}_i$.

As a result, all incoming messages reach $\mathcal{I}_i$ without changes and right after being received by $\mathcal{Z}_i$, fulfilling condition (b) of semi-simplistic adversaries. A^* orders $\mathcal{Z}_i$ to send out exactly the same messages as A and therefore achieves the same effect on the protocol as A. The adversary A^* only forwards inputs of $\mathcal{Z}_i$ to $\mathcal{I}_i$ or sends REVEAL messages to $\mathcal{I}_i$, hence fulfilling (c). By definition, the $\mathcal{Z}_i$ only receives commands to write messages so that the adversary communication satisfies condition (d). In addition, it follows the same corruption pattern as A and, for a coherent A, also A^* is coherent.

MODIFIED CLOCKING. Another modification is necessary to guarantee that A^* can always empty a buffer $b_{i,p}^-$ to corrupted $\mathcal{P}_i$. The adversary A^* always clocks a leaky buffer $b_{i,p}^-$ to corrupted $\mathcal{P}_i$ as soon as $\mathcal{F}_p$ writes to it. This fulfils the condition (a) of the simplistic adversary. The timing of $\mathcal{I}_i$ execution changes, but since all outputs of $\mathcal{I}_i$ are ignored by A^*, then this does not affect the following protocol execution. However, A^* must preserve the right clocking and timing of $\mathcal{I}_i^*$ for A. For that, A^* internally simulates the buffers $b_{i,p}^-$ for A using the real messages it has received. A^* delivers these messages to A when A clocks the message out of the buffer. Note that this change does not invalidate any of the semi-simplicity conditions achieved by the A^* construction.

COMPLEXITY. The adversary A^* has two big tasks – copying the state of $\mathcal{I}_i$ to make $\mathcal{I}_i^*$ and keeping $\mathcal{I}_i$ running. Copying can be done with polynomial overhead if the state is of polynomial size. However, $\mathcal{I}_i$ may receive unexpected inputs since

A can send arbitrary messages to $\mathcal{F}_p$ and, hence, the runtime of $\mathcal{I}_i$ is not bounded.

OUTSIDE VIEW. So far, the proof has shown that the adversary A^* is semi-simplistic. In addition, it was argued that the view of A inside the A^* remains the same as when running with the real protocol. It remains to argue that such A^* has the same effect on the environment Env where the protocol is running as A. Since A has the same view, then also the messages passed between A to Env remain the same. The messages to Π_e or $\mathcal{F}_p$ are all defined by A in the construction of A^*. Hence the view of all components outside of the $\mathcal{P}_i$ remains the same and the new adversary A^* achieves the same effect on the overall execution as A. □

The main limitation of the previous theorem is that the interpreter $\mathcal{I}_i$ may not terminate or its runtime may not be reasonably bounded if it receives unexpected inputs. It is reasonable to assume that it is possible to guard against such occurrences when designing the protocol. For example, $\mathcal{I}_i$ should ignore unexpected messages. It should also know the length of the expected messages and not process inputs longer than the maximal length. The remaining concern is that expected and valid messages could also trigger expensive computations in the interpreter. This is something the protocol designer should seek to avoid. The following specifies a protocol that has such good implementation.

Definition 34 (Robustness against malformed inputs)**.** A protocol is robust against malformed inputs if the running time of all interpreter components $\mathcal{I}_i$ is polynomial for all semi-simplistic adversaries.

Corollary 6. *If a protocol is robust against malformed inputs, then classes of semi-simplistic and generic adversaries are equivalent to each other.*

Proof. The robustness guarantees that the construction introduced in Lemma 11 has a polynomial overhead. Each time A^* invokes $\mathcal{I}_i$, it is guaranteed to stop and pass the control back to A^*. As the number of times A^* invokes $\mathcal{I}_i$ is bounded by the running time of A, the total running time of $\mathcal{I}_i$ can be only polynomial times bigger than the running time of A. Hence, any adversary can be converted to an equivalent semi-simplistic adversary. A semi-simplistic adversary is also a generic adversary. Hence explicit reverse conversion is not necessary. □

5.3.3. Security Against Rushed Messages And Lazy Adversaries

A semi-simplistic adversary may create messages that are dropped by recipients as they are not ready to process them. However, in most cases, these are unnecessary, and so the adversary can be modified not to send such messages.

Definition 35 (Input and output signature)**.** Let the input signature for a particular round π of computations inside a decomposed canonical $\mathcal{F}_p$ be the set of parties that must provide inputs before $\mathcal{T}_\mathcal{R}$ can run $\mathcal{R}$ and forward the recovered inputs to $\mathcal{F}_p$ and let the output signature be the set of parties that receive output shares from $\mathcal{T}_\mathcal{S}$.

It is difficult to analyse the execution of the protocol when the adversary can execute something too early or cause the execution of something not expected or clearly meaningful in the protocol. If some functionality only expects inputs from corrupted parties, then the adversary can execute this functionality whenever it wishes. In such cases, the adversary also has the capability to reconstruct all secure data used in this execution. It is also problematic if the adversary can mix up inputs of different instances. This can cause an execution of $\mathcal{F}_p$ where it is unclear which input was used and then this may derail the whole execution. For example, consider a possible functionality for oblivious transfer that has already received the inputs $x_1, \ldots, x_\ell$ and expects an index i. Upon receiving i, the functionality gives shares of x_i to all parties. If an honest party should send i, but the adversary can instead send i on behalf of some corrupted party, then the protocol may continue with shares of the wrong values. Hence, it is important to keep track of the right input signature and to ensure that the adversary executes only functionalities expected by the interpreter code.

The following definition formalises this requirement as rushing. It is rushing where the adversary may execute something too early. Rushing may cause parties to receive values from $\mathcal{F}_p$ sooner than they expect, and these messages may be dropped or wrongly used in the following execution.

Definition 36 (Rushed computation)**.** We say that a round π of computation is rushed if the ideal functionality $\mathcal{F}_p$ executes the computation before some interpreter $\mathcal{I}_i$ in the input signature has computed its input to π. A protocol Π is secure against rushed execution for the set of environments $\mathbb{E}$ if no semi-simplistic adversary from the class of adversaries $\mathbb{A}$ can rush a round of computation.

Note that the definition of rushing in this format is meaningful only when considering semi-simplistic adversaries. In other cases, it is not reasonable to expect the interpreter $\mathcal{I}_i$ of a corrupted party $\mathcal{P}_i$ to compute anything at all. Security against rushing means that the adversary has no point in trying to send any message earlier than the protocol execution has reached the place for this message. The following defines such an adversary as a lazy adversary.

Definition 37 (Lazy adversary)**.** A semi-simplistic adversary is lazy if it always waits for $\mathcal{I}_i$ to write a message (t_1, t_2, m) to $\mathcal{F}_p$ (or Π_e) to clock a message (t_1, t_2, m') out of the buffer $b^+_{i,p}$ (or $b^-_{i,\mathsf{e}}$) and it always clocks at most one message with the right tags per message from $\mathcal{I}_i$.

Lemma 12. *Assume that ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ and protocol Π using these functionalities have tight scheduling. If a protocol Π is secure against rushed execution, then any semi-simplistic adversary against this protocol can be converted to an equivalent lazy semi-simplistic adversary.*

Proof. Let A^* be the lazy adversary constructed from the semi-simplistic adversary A. A^* modifies which messages are written to the outgoing buffers of $\mathcal{P}_i$ and when they are clocked. This allows A^* to catch all events where A clocks a message (t_1, t_2, m) out of a buffer $b^+_{i,p}$ (or $b^-_{i,\mathsf{e}}$) before the interpreter $\mathcal{I}_i$ has produced

the respective message. In all these cases, A^* delays the clocking signal until the $\mathcal{I}_i$ has produced the relevant message. Due to tight scheduling, t_1 and t_2 uniquely determine the role of the message for the functionality $\mathcal{F}_p$. Hence, from the message and port used by $\mathcal{I}_i$, it is clear which message can be clocked to which $\mathcal{F}_p$. The overhead introduced by delaying the messages is bounded by the runtime of A. The following proof calls this message generation by $\mathcal{I}_i$ the send event.

UNLIMITED BUFFERING. Assume that $\mathcal{F}_p$ has unlimited buffering for all inputs. If no other party sends an input after the send event, then A^* has stopped a rushing event. However, as by assumption, Π is secure against rushing, then it must be that $\mathcal{F}_p$ is still waiting for some other inputs when A^* finally clocks (t_1, t_2, m). Hence, the overall execution does not change even though A^* delayed the message. As there is unlimited buffering, then $\mathcal{F}_p$ would have stored all the necessary inputs if it had received them earlier and these inputs cannot be changed. On the other hand, if the send event never occurs, then either this round is never completed anyway because of security against rushing, or this party was not in the input signature of this round and the $\mathcal{F}_p$ can proceed even though A^* does not try to clock this message.

GENERAL CASE. In the general case, $\mathcal{F}_p$ may drop some messages that it receives. Therefore, a delay in a message delivery may change which other messages are available at the time of receiving this message or how this message is processed (for example, in which round of $\mathcal{F}_p$ it is used). For example, $\mathcal{F}_p$ may drop the message when it is delivered according to the clocking of A and keep and use it when it is delivered according to A^*. Note that tight scheduling ensures that no message can replace (t_1, t_2, m) in the execution of $\mathcal{F}_p$ when the delivery is delayed because $\mathcal{P}_i$ can only have one message in the buffer for protocol instance t_2 of $\mathcal{F}_p$. In addition, thanks to tight scheduling and leaky buffers, A^* has complete knowledge about which rounds of which instances of $\mathcal{F}_p$ are pending and which are completed. If the buffering behaviour of $\mathcal{F}_p$ is known, then A^* therefore also knows which instances have messages in the buffer and when these will be dropped. Hence, if A^* knows that the message (t_1, t_2, m) would have been dropped if it was clocked according to A. For dropped messages, A^* can simply not clock them at all, even if it gets the signal from $\mathcal{I}_i$. If the message is delivered, then a similar analysis as for the unlimited buffering applies to show that this delay has no other effects on the execution.

Note that the communication between the protocol Π and Π_{e} does not differ much from the communication between $\mathcal{P}_i$ and $\mathcal{F}_p$. However, the roles are switched and Π_{e} behaves as the party $\mathcal{P}_i^*$. The adversary is coherent. Hence, it can only create and try to rush messages between corrupted $\mathcal{P}_i^*$ to $\mathcal{P}_i$. Hence, any delays introduced by the lazy adversary only affect the corrupted communication. Without lessening of generality, any action that the corrupted party would have upon receiving the delayed message can be performed by A^* also without delivering this message at the time when A clocked it. □

Theorem 11. *If a protocol is robust against malformed inputs (Definition 34) and is secure against rushed execution (Definition 36), then lazy semi-simplistic (Definition 37) and generic adversaries (Definition 18) are equivalent to each other.*

Proof. We know from Corollary 6 that generic adversaries are equivalent to semi-simplistic adversaries. Lemma 12 proves that any semi-simplistic adversary can be transformed into a lazy semi-simplistic adversary. Any lazy semi-simplistic adversary is still a semi-simplistic adversary. Hence, the two classes are equivalent. □

If a protocol can be rushed, then the adversaries most likely cannot be converted to lazy adversaries. However, for secure multiparty computation, it is reasonable to expect that the protocols themselves are secure against rushing. Commonly, all parties need to give input to $\mathcal{F}_p$, and then an honest party has some timing control over all protocols in which it participates. In fact, it is reasonable to consider cases where, for each interpreter $\mathcal{I}_i$, the code is the same or almost the same. If it is almost the same, then the main interest is when and in which order the ideal functionalities $\mathcal{F}_p$ are executed. A protocol is symmetric for all interpreters if, for all of them, the instructions to call $\mathcal{F}_p$ are the same, and they are in the same dependency graph. For example, no interpreter calls $\mathcal{F}_2$ instance t_2 before $\mathcal{F}_1$ instance t_1 has given its outputs. For sequential program execution, all the calls are also in the same order. For parallel execution, the dependency rules must be the same, but the order, in which the calls that are not depending on each other are made, may vary. A symmetric program implies that all parties are in the input signature of all functionalities.

The following theorem considers a more general approach to characterise protocols that avoid rushing. The code interpreter $\mathcal{I}_i$ will be introduced in more detail in Section 5.4.1. However, the theorem uses the fact that the functionality calls out new protocols $\mathcal{F}_p$ when it has received the answers to previous calls to ideal functionalities and has computed the necessary input for the new protocol.

Theorem 12 (Characterisation of rushing)**.** *A semi-simplistic adversary can rush a round of computation π for a canonical ideal functionality with tight scheduling only if, for some corrupted $\mathcal{P}_i$ in the input signature, one of the following statements holds.*

(a) The round π is not in the program code of the interpreter $\mathcal{I}_i$.

(b) The interpreter $\mathcal{I}_i$ needs an input from Π_{e} to submit an input to π.

(c) The interpreter $\mathcal{I}_i$ needs an output from a round of computation π' to submit an input to π and π' is executed concurrently or after π.

Proof. For a rushing event to occur, there has to be some party $\mathcal{P}_i$ that is in the input signature for the rushed round of computations π such that the computation in $\mathcal{F}_p$ is completed before $\mathcal{I}_i$ has computed its input. Then $b_{i,p}^{+}$ is the buffer where $\mathcal{P}_i$ should write its input to $\mathcal{F}_p$. If a party is in the input signature, then, by

definition, the input collector $\mathcal{T}_{\mathcal{R}}$ of $\mathcal{F}_p$ does not proceed with the round before receiving the input from $b^+_{i,p}$. If $\mathcal{P}_i$ is honest, then only $\mathcal{I}_i$ could have computed this input. Hence, rushing is only possible if $\mathcal{P}_i$ is corrupted at the time when the input was written to $b^+_{i,p}$ and clocked to $\mathcal{F}_p$.

There are two possible reasons why the interpreter of a corrupted party has not yet computed the necessary input. The first reason is that the interpreter code does not have an instruction to compute this input and call this instance of $\mathcal{F}_p$. Second, $\mathcal{I}_i$ has this instruction in its program but has not reached it yet because it waits for some previous calls to complete. The only valid reason for not yet completing a computation is if there is some input m from some $\mathcal{F}_q$ or Π_{e} that is necessary, and this input has to arrive on some buffer $b^-_{i,q}$. A semi-simplistic adversary would always empty the buffer $b^-_{i,q}$ before clocking $b^+_{i,p}$ if there was anything in the buffer $b^-_{i,q}$. Hence, this situation is only possible if the message m is computed by some round of some $\mathcal{F}_q$ that has not yet written the output or if it should arrive from Π_{e}. □

The conditions that characterise rushing can be managed to avoid rushing. For example, Byzantine agreements could be used in protocol Π to ensure that no honest party starts a round before all parties have their inputs from Π_{e}. Alternatively, in some cases, it might be reasonable to only consider Π_{e} that gives inputs in one go to all parties. This is similar to how $\mathcal{F}_p$ writes all outputs to the buffers at the same time.

The previous theorem can be used to consider rushing in different execution contexts. For example, if Π implements an ideal functionality $\mathcal{F}$ then it is necessary to analyse rushing $\mathcal{F}$ in $\mathsf{Env}\langle\mathcal{F}_{\triangle}\langle\mathcal{F},\mathcal{F}_1,\ldots,\mathcal{F}_k\rangle\rangle$. If this is impossible, then it is clear that Π cannot be rushed from the Π_{e} in the context where it is used and condition (b) is satisfied. It is then further necessary to consider the rushing conditions (a) and (c) from Theorem 12 for the protocol Π.

Going back to the ideas of common protocols for secure multiparty computation, it is clear that each round is in the program of all interpreters if the program is symmetric. For a symmetric program, all parties have the same dependency graph for all their ideal functionality calls. Hence, condition (a) cannot occur. Also, all ideal functionalities give outputs at the same time and a symmetric program has the same data dependencies. Therefore, condition (c) is also not possible. Hence, the only possibility to rush such protocols comes from the inputs given by Π_{e}. In the context of protection domains, the Π_{e} stands for the other computations taking part inside the protection domain. Hence, Π_{e} can be expected to behave quite similarly to the actual protocol Π. Hence, we could also expect that it is running a symmetric program and can deliver inputs promptly to Π. Hence, it is reasonable to consider lazy semi-simplistic adversaries in the following discussion.

5.4. Shared Memory and Simplistic Adversaries

Lazy semi-simplistic adversaries are restricted to clocking all messages to $\mathcal{F}_p$ according to the protocol specification in the interpreters. However, the adversaries can generate more messages to the buffers, which significantly complicates the understanding of the protocol execution. This section specifies the interpreters and defines a shared memory model with a simplistic adversary that does not generate unnecessary messages. Instead, the simplistic adversary can only modify the memory locations of outgoing messages.

This section defines a $\bowtie$-operator that acts on protocols and their components together with a universal construction $\phi_{\bowtie}$ and its semi-inverse $\phi^*_{\bowtie}$ that achieves

$$\forall \Pi \in \mathbb{P}_{\mathsf{c}} : \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \forall \mathsf{A} \in \mathbb{A}^{lazy}_{\Pi,\mathsf{Env}} : \mathsf{Env}\langle \Pi, \mathsf{A}\rangle \equiv \mathsf{Env}\langle \Pi^{\bowtie}, \phi_{\bowtie}(\mathsf{A})\rangle \quad (5.8)$$

$$\forall \Pi \in \mathbb{P}_{\mathsf{c}} : \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \forall \mathsf{A} \in \mathbb{A}^{\bowtie}_{\Pi,\mathsf{Env}} : \mathsf{Env}\langle \Pi^{\bowtie}, \mathsf{A}^{\bowtie}\rangle \equiv \mathsf{Env}\langle \Pi, \phi^*_{\bowtie}(\mathsf{A}^{\bowtie})\rangle, \quad (5.9)$$

where $\mathbb{P}_{\mathsf{c}}$ is the set of protocols executing a well-formed programs (Definition 38) with canonical specification (Definition 42) and black-box use of the functionalities $\mathcal{F}_p$. $\mathbb{E}_{\Pi}$ is the set of environments compatible with the protocol. The first bigger result is the equivalence of lazy semi-simplistic adversary (Definition 37) and simplistic adversary (Definition 39) in Theorem 13 and 14 for the $\diamond$-operator. The last transformation of this section simplifies the memory model and extends the $\diamond$-operator to the $\bowtie$-operator in Theorem 15. The transformations also define how $\Pi^{\bowtie}$ is derived from Π.

5.4.1. Details of Code Interpretation and Well-Formed Programs

This section specifies the version of code interpretation used in this chapter of the thesis. It is quite restricted and mostly focused on the places where the interpreter sends out a message. In general, the interpreter has two actions, either performing local computations or sending messages. In more detail, interpreter $\mathcal{I}_i$ for party $\mathcal{P}_i$ is a universal random-access machine with communication instructions DMACALL and SEND. The first is for communication expecting a reply and the latter is for one-directional communication. The interpreter executes code $\mathfrak{p}$ and can execute many instances of this protocol simultaneously. The instances are identified by their tags t. The internal state of the interpreter is an array $\mathfrak{s}[t,\delta,\ell]$ where ℓ is the memory location and δ the storage domain where the value stored in location ℓ for protocol instance t belongs to. Any protocol instance t can only access its values in $\mathfrak{s}[t,\cdot,\cdot]$. The initial state of $\mathfrak{s}$ is empty.

A new protocol instance is started when $\textsc{Init}(t_1,t_2,\delta,\mathbf{m})$ is received from Π_{e}. Upon receiving it, $\mathcal{I}_i$ starts the execution of instance t_2 by storing the values $\mathbf{m}$ with domains δ. The instance t_1 is stored as the parent protocol instance and only this instance can be used when responding to Π_{e} from protocol instance t_2.

An instruction $\textsc{DmaCall}(t_2,p,\alpha,\beta)$ is meant for port b_p^+ to send the $\mathfrak{s}[t_2,\delta_i,\ell_i]$, where $\alpha = ((\delta_1,\ell_1),\ldots,(\delta_u,\ell_u))$. The vector β specifies where the responses are

stored and the instruction is not complete before the answer is received. The party does not send a new instruction with the same tags to the same $\mathcal{F}_p$ before the instruction is complete. $\text{SEND}(t_2, p, \alpha)$ behaves similarly but does not expect a response. Outgoing messages to functionalities $\mathcal{F}_p$ are in format (t_2, t_3, m), where t_2 is the instance of $\mathcal{I}_i$ and t_3 is the instance of $\mathcal{F}_p$, and m is the message collected from the memory locations specified in α. Messages to Π_{e} are in the format (t_1, t_2, m).

The computations that $\mathcal{I}_i$ performs locally are local memory manipulation and program control flow. The conditional jumps in the program can only occur using variables in public or local storage domains. In addition, the interpreter stores its state and can forward it to the adversary when receiving a REVEAL call. It is reasonable to expect the programs that the interpreters run to satisfy some structural properties for easier analysis. These conditions are defined as well-formed programs.

Definition 38. A program $\mathfrak{p}$ is well-formed if the following holds.

(a) Each memory location $\mathfrak{s}[t, \delta, \ell]$ can be assigned only once.

(b) No instruction can read a memory location before it is assigned.

(c) A new message with tag (t_1, t_2) is never written to the output port $k+1$ to Π_{e} before reading a message with tag (t_1, t_2) from the input port $k+1$ from Π_{e}. A message with a tag (t_2, t_3) is newer written to the output port p for $\mathcal{F}_p$ before receiving (t_1, t_2) from port $k+1$ from Π_{e}.

(d) For instructions $\text{DMACALL}(t, p, \alpha, \beta)$ and $\text{SEND}(t, p, \alpha)$ no other program instruction can read memory locations in the vector α and write the memory location β.

(e) The order in which the responses to DMACALL are received does not affect how the received values are used.

The constraints on the memory access are necessary to modify the execution so that the ideal functionalities $\mathcal{F}_p$ read the inputs from the same memory and the adversary A can rewrite α part of the memory to change the inputs to the functionalities. The adversary will be restricted so that only $\mathcal{F}_p$ or Π_{e} can write the location β that is in the DMACALL.

Such memory management is wasteful since the easiest way to guarantee (d) is to copy each input to a new location before DMACALL or SEND, but it can be achieved in polynomial time in the runtime of the interpreter. The property (c) is there to ensure that all sub-protocols are executed as calls to $\mathcal{F}_p$ and messages to Π_{e} are responses to received initialisation messages. Especially, the program $\mathfrak{p}$ cannot call new instances of Π_{e}. Overall, it is also reasonable to say that the following considers programs that have a well-formed implementation, as any program can be transformed to satisfy the rules of the definition. The property (e) limits the effect that the adversary could have on programs that can send many calls at once. It also disallows programs that, for example, only use the first response that they receive and discard other responses. The adversary can change

the order in which messages are received. Hence it is important to ensure that each message is expected and has a fixed use. The adversary cannot mix inputs simply by changing the clocking. Ensuring property (e) is especially relevant if the value is in a storage domain where pieces are held by several parties, as the execution has to align so that the parties continue the evaluation with the same values.

5.4.2. Shared Memory Model

This section extracts the memory component from the interpreter specification and replaces some of the message communication with communicating the message locations and reading values from shared memory. This creates an adversary that can only modify messages that are part of the protocol and not create fraudulent messages or early messages. It will also allow the transformations to later merge the memory components to discuss the values rather than each party's view.

The interpreter $\mathcal{I}_i$ is replaced by a stateless interpreter $\mathcal{I}_i^\diamond$ and a memory $\mathcal{M}_i^\diamond$ that keeps all the interpreter state. In addition, the input-output functionalities that communicate with Π_e are separated from all interpreters and collected to form a new dedicated machine $\mathcal{F}_{\text{io}}^\diamond$. The ideal functionalities $\mathcal{F}_p$ are modified to $\mathcal{F}_p^\diamond$ that can read the values from $\mathcal{M}_1^\diamond, \ldots, \mathcal{M}_k^\diamond$. This setup that forms $\Pi^\diamond$ is illustrated in Figure 37.

When $\mathcal{F}_{\text{io}}^\diamond$ receives an input for $\mathcal{I}_i^\diamond$ from Π_e, then $\mathcal{F}_{\text{io}}^\diamond$ writes it to $\mathcal{M}_i^\diamond$ and clocks the notification about the memory location to $\mathcal{I}_i^\diamond$. Analogously, when $\mathcal{F}_{\text{io}}^\diamond$ receives an output from $\mathcal{I}_i^\diamond$ to Π_e, then it reads the value from memory and writes and clocks the value to Π_e with the right tags. The DMACALL and SEND contain all information needed to compose the right message. $\mathcal{F}_{\text{io}}$ also forwards $\text{INIT}(t_1, t_2, \delta, m)$. $\mathcal{I}_i^\diamond$ processes $\text{INIT}(t_1, t_2, \delta, m)$ the same as $\mathcal{I}_i$ did before. $\mathcal{I}_i^\diamond$ reads all necessary values from $\mathcal{M}_i^\diamond$ and writes all computed values to $\mathcal{M}_i^\diamond$. $\mathcal{M}_i^\diamond$ contains the state $\mathfrak{s}$ that was previously stored in $\mathcal{I}_i$.

The messages between $\mathcal{I}_i^\diamond$ and $\mathcal{F}_p^\diamond$ consider memory addresses rather than the actual messages. The message (t_1, t_2, m) between $\mathcal{I}_i$ and $\mathcal{F}_p$ is replaced by $(t_1, t_2, \alpha, \beta)$ that is defined by the DMACALL. Similarly, (t_1, t_2, α) is used for SEND. $\mathcal{F}_p^\diamond$ reads the indicated memory location from $\mathcal{M}_i^\diamond$ to get the same message m as $\mathcal{F}_p$ gets from $\mathcal{I}_i$. $\mathcal{F}_i^\diamond$ also stores the response to $\mathfrak{s}[t_1, \beta'_j, \ell'_j]$ and only writes a default message with instances (t_1, t_2) to $\mathcal{I}_i^\diamond$. $\mathcal{I}_i^\diamond$ knows which address in $\mathcal{M}_i^\diamond$ contains the actual response value as the response location was indicated in the DMACALL.

The corruption manager $\mathcal{Z}_i$ is also modified to $\mathcal{Z}_i^\diamond$ in Figure 37. The machine $\mathcal{Z}_i$ managed the access to the interpreter state that the adversary could have. Currently, $\mathcal{Z}_i^\diamond$ does the same, but it does not have the power to write new messages that $\mathcal{Z}_i$ has. In a way, $\mathcal{Z}_i^\diamond$ can still simulate the execution of $\mathcal{I}_i$ and $\mathcal{Z}_i$ to the adversary A. In order to do so, $\mathcal{Z}_i^\diamond$ plays the missing role of $\mathcal{Z}_i$. Mainly, $\mathcal{Z}_i^\diamond$ translates the missing buffers $b_{i,p}^-, b_{i,p}^+$ and their clocking to clocking $c_{i,p}^-, c_{i,p}^+$. The commands

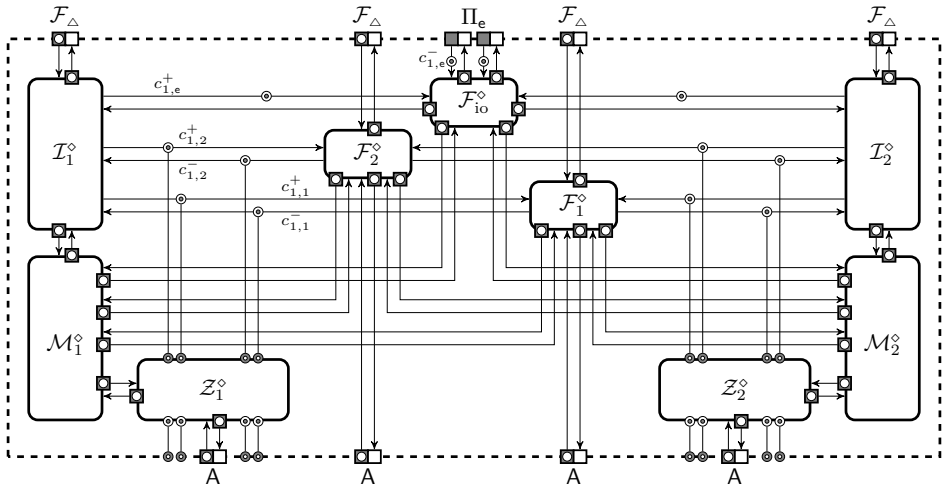


Figure 37: Separating the interpreter state to $\mathcal{M}_i^\diamond$ and input-output behaviour to $\mathcal{F}_{\text{io}}^\diamond$ to define $\Pi^\diamond$ with interpreters $\mathcal{I}_i^\diamond$.

to send new messages to $b_{i,p}^+$ are replaced with memory modifications. $\mathcal{Z}_i^\diamond$ only accesses $\mathcal{M}_i^\diamond$ if the party $\mathcal{P}_i$ is corrupted. For corrupted parties, $\mathcal{Z}_i^\diamond$ can modify the memory locations α of incomplete $\text{DMACALL}(t,p,\alpha,\beta)$ and $\text{SEND}(t,p,\alpha)$ calls. In the following, $\phi_\diamond(\mathsf{A})$ denotes the new adversary consisting of the A against the previous protocol and the modules $\mathcal{Z}_i^\diamond$ for all parties.

Theorem 13. *Let Π be the protocol with a well-formed implementation (Definition 38) and let $\mathbb{E}_\Pi$ be the set of compatible environments. Then*

$$\forall \mathsf{Env} \in \mathbb{E}_\Pi : \quad \forall \mathsf{A} \in \mathbb{A}^{lazy}_{\Pi,\mathsf{Env}} : \qquad \mathsf{Env}\langle \Pi, \mathsf{A}\rangle \equiv \mathsf{Env}\langle \Pi^\diamond, \phi_\diamond(\mathsf{A})\rangle \ ,$$

where and $\mathbb{A}^{lazy}_{\Pi,\mathsf{Env}}$ is the set of compatible lazy semi-simplistic adversaries (Definition 37).

Proof. INITIAL STATE. Initially, $\mathcal{F}_p$ and $\mathcal{F}_p^\diamond$ are only storing the setup, $\mathcal{I}_i$ and $\mathcal{I}_i^\diamond$ have no active instances, and all buffers are empty. The two setups are equivalent if no action of Env or A can diverge the execution to non-equivalent states. $\mathcal{F}_{\text{io}}^\diamond$ does not have a relevant state. It always forwards the inputs that it gets and simply translates writing or reading message contents from the memory. In the following, the discussion about $b_{i,p}^+$ also applies to $b_{i,\text{e}}^-$ (and $b_{i,p}^-$ also applies to $b_{i,\text{e}}^+$), with the additional translation of messages carried out by $\mathcal{F}_{\text{io}}^\diamond$.

STATE EQUIVALENCE. Collections Π and $\Pi^\diamond$ with $\mathcal{Z}_i^\diamond$ have a natural matching between machines and their internal states. The states of the two protocol versions are equivalent if the respective components have equivalent states. The interpreters are at the same instructions and have received the same inputs. The buffer $b_{i,p}^-$ contains a message $(t_1,t_2,\mathbf{m})$ and $c_{i,p}^-$ contains OK for t_1,t_2,ℓ and $\mathfrak{s}[t_1,\delta_i,\ell_i] = m_i$ in $\mathcal{M}_i^\diamond$. Note that the $\text{DMACALL}(t_2,p,\alpha,\beta)$ specifies the locations and storage domains of the sent messages in α and of the responses in β

so the equivalence is based on the addresses in β. $c^-_{i,p}$ may keep the message for a longer time than $b^-_{i,p}$. $\mathcal{F}_p$ and $\mathcal{F}^\diamond_p$ have received the messages with the same instances, $\mathcal{F}^\diamond_p$ may not have read the message content from $\mathcal{M}^\diamond_i$ yet. There is no simple equivalence between $b^+_{i,p}$ and $c^+_{i,p}$. Instead, $\mathcal{Z}^\diamond_i$ internally maintains simulated $b^{++}_{i,p}$ for each functionality $\mathcal{F}^\diamond_p$ that is equivalent to $b^+_{i,p}$.

Overall, the behaviour of $\mathcal{Z}^\diamond_i$ is quite simple. First, it has a simulated buffer $b^{++}_{i,p}$ for each $b^+_{i,p}$ in Π. Every time A tries to write something to $b^+_{i,p}$, it is really written to $b^{++}_{i,p}$. When an adversary A clocks a message to $\mathcal{F}_p$ from a corrupted party $\mathcal{P}_i$, then $\mathcal{Z}^\diamond_i$ takes the message (t_1,t_2,m) from the simulated $b^{++}_{i,p}$. A message with the same tags is also in $c^+_{i,p}$ in the format (t_1,t_2,α,β), as the adversary is lazy semi-simplistic. Hence, $\mathcal{Z}^\diamond_i$ sets the memory location $\mathfrak{s}[t_1,\delta_i,\ell_i] = m_i$ with addresses from α corresponding to the message in $c^+_{i,p}$. Then, $\mathcal{Z}^\diamond_i$ clocks this message out from $c^+_{i,p}$.

The corrupted $\mathcal{Z}_i$ sends all inputs to A. In order to simulate this behaviour, $\mathcal{Z}^\diamond_i$ first receives the signal to clock $b^-_{i,p}$ to a corrupted party $\mathcal{P}_i$ from A and knows that the party will have an input. $\mathcal{Z}^\diamond_i$ can clock the leak on $c^i_{i,p}$ to learn the protocol instance t_2 and there can be only one outstanding DMACALL with instance t_2 to functionality $\mathcal{F}_p$. $\mathcal{Z}^\diamond_i$ reads the relevant memory location β for this DMACALL from $\mathcal{M}^\diamond_i$ and gives the received input value to A. The message is clocked out of $c^-_{i,p}$ as a next step when A orders $\mathcal{Z}_i$ to give this input value to $\mathcal{I}_i$. By definition, A orders $\mathcal{Z}_i$ to clock the value to $\mathcal{I}_i$ immediately after clocking $b^-_{i,p}$ and receiving the input from $\mathcal{Z}_i$, as A is semi-simplistic. The adversary A clocking any $b^-_{i,p}$, as well as $b^+_{i,p}$ for honest $\mathcal{P}_i$, can be directly forwarded to clock respective $c^-_{i,p}$ or $c^+_{i,p}$. Hence, $\mathcal{Z}^\diamond_i$ simply forwards these clocking events. If the adversary corrupts a party $\mathcal{P}_i$, then $\mathcal{Z}^\diamond_i$ can read the state of $\mathcal{P}_i$ from $\mathcal{M}^\diamond_i$ and give it to A the same way as $\mathcal{Z}_i$ forwards it. The adversary A can separately clock leaky buffers to receive leaks and forward messages. Leaks can be simulated by $\mathcal{Z}^\diamond_i$ either using $b^{++}_{i,p}$ for corrupted $\mathcal{P}_i$ or by using $c^+_{i,p}$ and $c^-_{i,p}$ for other cases similarly to clocking actions.

The Env gives the initial input, and property (c) of the well-formed program ensures that the interpreters do not take any actions before such input and only execute instances started by Env. A can corrupt parties, clock buffers, and write messages to b^+. It also orders $\mathcal{Z}^\diamond_i$ to forward messages from $\mathcal{F}_p$ to $\mathcal{I}_i$ but does that immediately after clocking the message from $b^-_{i,p}$ to $\mathcal{Z}_i$.

The properties (a) and (d) of well-formed programs guarantee that $\mathcal{Z}^\diamond_i$ can rewrite the memory location for messages to $\mathcal{F}_p$ without altering the behaviour of the interpreter. Furthermore, the change can be done once for each memory location. Thanks to the fact that A is lazy, the memory location has already been assigned by $\mathcal{I}^\diamond_i$ and is not written over anymore by the interpreter. Hence, $\mathcal{F}_p$ and $\mathcal{F}^\diamond_p$ always have the same input values. This means they also send the same replies to $\mathcal{I}_i$ or $\mathcal{I}^\diamond_i$. Hence, all actions of the adversary result in equivalent states for $\mathcal{I}_i$ and $\mathcal{I}^\diamond_i$ and, therefore, also the equivalent view of the protocol that either A or Env can have. □

5.4.3. Simplistic Adversary

The adversary $\phi_\diamond(\mathsf{A})$, as defined before, is simpler than the lazy semi-simplistic adversary A. Such adversaries are defined as simplistic adversaries.

Definition 39 (Simplistic adversary)**.** An adversary is simplistic if it fulfils the following conditions.

(a) The adversary clocks any outgoing buffer $c^+_{u,p}$ or $c^-_{u,\mathsf{e}}$ to party $\mathcal{P}_u$ and any incoming buffer $c^-_{j,p}$ or $c^+_{j,\mathsf{e}}$ for honest $\mathcal{P}_j$ only when all incoming buffers $c^-_{i,p}$ or $c^+_{i,\mathsf{e}}$ to corrupted parties $\mathcal{P}_i$ are empty.

(b) The adversary can modify the state of the corrupted party only in the locations α of pending $\textsc{DmaCall}(t,p,\alpha,\beta)$ and $\textsc{Send}(t,p,\alpha)$ instructions. These changes are done before the corresponding tuple is clocked to $\mathcal{F}^\diamond_p$ or $\mathcal{F}^\diamond_{\text{io}}$, and each value is modified at most once.

(c) The adversary is coherent.

The simplistic adversary is coherent, similarly to a semi-simplistic adversary. In addition, the clocking rules represent the same idea as the semi-simplistic adversary. The main difference is in condition (b), which replaces the capabilities of a semi-simplistic adversary communicating with $\mathcal{Z}_i$ with the capabilities to modify the memory for outstanding DMACALL.

Corollary 7. *For any lazy semi-simplistic adversary* A *and for any well-formed implementation of* Π*, the construction* $\phi_\diamond(\mathsf{A})$ *is a simplistic adversary.*

Proof. The clocking rules for simplistic $\phi_\diamond(\mathsf{A})$ are similar to those of the semi-simplistic A. The difference is in the concrete buffers connecting the party and $\mathcal{F}^\diamond_p$ or $\mathcal{F}_p$, respectively. As $\phi_\diamond(\mathsf{A})$ follows the same clocking as A, $\phi_\diamond(\mathsf{A})$ also preserves the condition (a) of the simplistic adversary. The way $\mathcal{Z}^\diamond_i$ in $\phi_\diamond(\mathsf{A})$ alter memory just before clocking $c^+_{i,p}$ in the proof of Theorem 13 guarantees that the property (b) of the simplistic adversary is satisfied. The corruption calls are unchanged by the transformation $\phi_\diamond$ and, therefore, the resulting adversary is coherent as the lazy semi-simplistic adversary A was coherent. □

Let $\mathbb{A}^\diamond_{\Pi^\diamond,\mathsf{Env}}$ be the class of simplistic adversaries against $\Pi^\diamond$. Based on the construction of the adversary from the hybrid execution, it is sufficient to consider only simplistic $\mathbb{A}^\diamond_{\Pi^\diamond,\mathsf{Env}}$ and to show the equivalence of the two execution models when executing with the respective adversary classes. Hence, the following also defines the semi-inverse of $\phi^\diamond$ as $\phi^*_\diamond : \mathbb{A}^\diamond_{\Pi^\diamond,\mathsf{Env}} \to \mathbb{A}^{lazy}_{\Pi,\mathsf{Env}}$ with the right properties.

It remains to also show that it is possible to move back from the shared memory model with the simplistic adversary to the basic hybrid execution with the lazy semi-simplistic adversary. Here, a semi-inverse of $\phi_\diamond$ is needed. It is constructed by reversing $\mathcal{Z}^\diamond_i$ using machines $\mathcal{Z}^*_i$ that go between Π and the simplistic adversary $\mathsf{A}^\diamond$. The simulator machines $\mathcal{Z}^*_i$ translate memory rewrites by $\mathsf{A}^\diamond$ to protocol messages and enables the reading of the state of $\mathcal{I}_i$ similarly to $\mathcal{I}^\diamond_i$. Let $\phi^*_\diamond(\mathsf{A}^\diamond)$ be the collection consisting of $\mathsf{A}^\diamond, \mathcal{Z}^*_1, \ldots \mathcal{Z}^*_n$.

Theorem 14. *Let* Π *be the protocol with a well-formed implementation and robust against malformed inputs and let* $\mathbb{E}_\Pi$ *be the set of compatible environments. Then*

$$\forall \mathsf{Env} \in \mathbb{E}_\Pi : \quad \forall \mathsf{A}^\diamond \in \mathbb{A}^\diamond_{\Pi^\diamond,\mathsf{Env}} : \qquad \mathsf{Env}\langle \Pi^\diamond, \mathsf{A}^\diamond \rangle \equiv \mathsf{Env}\langle \Pi, \phi^*_\diamond(\mathsf{A}^\diamond) \rangle \ ,$$

where $\mathbb{A}^\diamond_{\Pi^\diamond,\mathsf{Env}}$ *is the set of simplistic adversaries compatible with the protocol and the environment. The resulting adversary* $\phi^*_\diamond(\mathsf{A}^\diamond)$ *is lazy and semi-simplistic.*

Proof. The equivalence between the states of the protocol is mostly the same as used in the proof of Theorem 13. The machine $\mathcal{Z}_i^*$ has translate adversary $\mathsf{A}^\diamond$ memory accesses to interactions with $\mathcal{I}_i$ and $\mathcal{Z}_i$. In addition, $\mathcal{Z}_i^*$ has to modify adversary clocking $c^+_{i,p}$ and $c^-_{i,p}$ to clocking $b^+_{i,p}$ and $b^+_{i,p}$.

The machines $\mathcal{Z}_i^*$ contain a simulated memory $\mathcal{M}_i^*$. The buffer $c^+_{i,p}$ has a message when the interpreter $\mathcal{I}_i^\diamond$ (or $\mathcal{I}_i$ as they are in the same state) has computed it, but $b^+_{i,p}$ for corrupted $\mathcal{P}_i$ only contains a similar message if the adversary has ordered it to be written. $\mathcal{Z}_i^*$ holds a simulated copy $c^{++}_{i,p}$ of $c^+_{i,p}$ to keep track of these messages written by $\mathcal{I}_i^\diamond$ but only writes to $b^+_{i,p}$ when necessary. In addition, $\mathcal{Z}_i^*$ runs a simulated version of $\mathcal{I}_i^\diamond$ in order to keep track of the state of the simulated memory $\mathcal{M}_i^*$. This simulation can be done in polynomial time if the protocol is robust against malformed inputs.

The equivalence of the two cases is easy if no party is corrupted. In such cases, $\mathcal{Z}_i^*$ can just forward all clocking signals. When a party $\mathcal{P}_i$ is corrupted by $\mathsf{A}^\diamond$, then $\mathcal{Z}_i^*$ sends REVEAL message to $\mathcal{Z}_i$ and forwards the response to $\mathsf{A}^\diamond$ as the values in $\mathcal{M}_i^\diamond$. Then, $\mathcal{Z}_i$ becomes corrupted and, each time it receives something from $\mathcal{F}_p$, it sends this to $\mathcal{Z}_i^*$ that adds this value to the simulated memory $\mathcal{M}_i^*$ and orders $\mathcal{Z}_i$ to give the input to $\mathcal{I}_i$. $\mathcal{Z}_i$ also gives the results computed by $\mathcal{I}_i$ to $\mathcal{Z}_i^*$. $\mathcal{Z}_i^*$ stores these values in $\mathcal{M}_i^*$ based on the timing of simulated $\mathcal{I}_i^\diamond$.

By definition, the simplistic adversary $\mathsf{A}^\diamond$ only modifies the memory locations of pending DMACALL or SEND calls. If the adversary modifies the memory to contain a value m', then $\mathcal{Z}_i^*$ orders $\mathcal{Z}_i^*$ to send a message with the same tags as the pending call and content m' to the same functionality $\mathcal{F}_p$ as the original pending call. If the adversary clocks a channel $c^+_{i,p}$ then the message with the same tags is clocked to $\mathcal{F}_p$ from $b^+_{i,p}$. If there is no such message on the buffer $b^+_{i,p}$ for a corrupted party $\mathcal{P}_i$ yet, then $\mathcal{Z}_i^*$ first writes the message using the current value in $\mathcal{M}_i^*$ for the locations used in this message. The simplistic adversary modifies each value at most once. Hence, $\mathcal{Z}_i^*$ orders one message to be sent by $\mathcal{Z}_i$ for each message that is clocked.

The inputs to $\mathcal{F}_p$ and $\mathcal{F}_p^\diamond$ are clocked at the same time and with the same content. Hence, the ideal functionalities also give outputs at the same time and with the same content. Hence, $\mathcal{Z}_i^*$ can clock $b^-_{i,p}$ for honest party at the same time as $\mathsf{A}^\diamond$ clocks $c^-_{i,p}$. The same goes for corrupted parties, as long as the adversary clocks $c^-_{i,p}$ early enough. For corrupted parties, the adversary may try to read the response from $\mathcal{F}_p^\diamond$ in $\mathcal{M}_i^\diamond$ before the reply is clocked from $c^-_{i,p}$ to $\mathcal{P}_i$. In such a

case, $\mathcal{Z}_i^*$ does not have it in $\mathcal{M}_i^*$. Instead, $\mathcal{Z}_i^*$ clocks the respective message out from $b_{i,p}^-$ and receives it through $\mathcal{Z}_i$. For the new adversary to be semi-simplistic, $\mathcal{Z}_i^*$ also immediately orders $\mathcal{Z}_i$ to give the value to $\mathcal{I}_i$. If $\mathcal{I}_i$ computes any new messages, then these are given to $\mathcal{Z}_i^*$ by $\mathcal{Z}_i$. However, in this case, these are not yet stored in $\mathcal{M}_i^*$. These values will be stored when the adversary really clocks $c_{i,p}^-$ and simulated $\mathcal{I}_i^\diamond$ computes these values.

The resulting adversary $\phi(\mathsf{A}^\diamond)$ is lazy and semi-simplistic. The new adversary obeys the clocking rules of the simplistic adversary, which cover the clocking rules of the semi-simplistic adversary. In addition, the adversary fetches the state of $\mathcal{I}_i$ but otherwise does not interact with $\mathcal{I}_i$. The adversary is also lazy, as the clocking is based on the simplistic adversary. The interpreter $\mathcal{I}_i$ has always computed the messages that $\mathcal{I}_i^\diamond$ has computed. In case of early clocking of $c_{i,p}^-$ by $\mathcal{Z}_i^\diamond$ to correctly read the memory for a corrupted party $\mathcal{P}_i$, the interpreter $\mathcal{I}_i$ may be ahead of $\mathcal{I}_i^\diamond$. The simplistic adversary only clocks the message if it has been computed by $\mathcal{I}_i^\diamond$ and then it has also been computed by $\mathcal{I}_i$, obeying the rules of the lazy adversary. As the order in which messages are received by a well-formed program does not affect which messages are sent, then any changes in the clocking do not affect the next messages. □

5.4.4. Canonical MPC Protocol

The model with separate machines for memories introduces the possibility of joining the machines $\mathcal{M}_i^\diamond$ to one machine. Let the global state $\mathfrak{gs}$ be the state of such a machine where $\mathfrak{gs}[t,\delta,\ell,i] = \mathfrak{s}[t,\delta,\ell]$ for party $\mathcal{P}_i$. Let $\mathfrak{gs}[t,\delta,\ell]$ denote $(\mathfrak{gs}[t,\delta,\ell,1],\ldots,\mathfrak{gs}[t,\delta,\ell,k])$. This state is reasonable only when the same locations ℓ over $\mathfrak{s}$ of different parties correspond to the same value in the storage domain δ. Such a case is defined as memory alignment and is focused on the instances where the ideal functionalities read the memory.

Definition 40 (Memory-alignment)**.** A well-formed implementation of a protocol uses an ideal functionality $\mathcal{F}_p^\diamond$ in a memory-aligned manner if each individual input of $\mathcal{F}_p^\diamond$ is reconstructed from $\mathfrak{gs}[t,\delta,\ell]$ and each output is shared to $\mathfrak{gs}[t,\delta,\ell]$.

Again, memory alignment is more of an implementation detail of a protocol than a property of the protocol itself. If the protocol only calls $\mathcal{F}_p^\diamond$ and does not perform any local computations or if all $\mathcal{I}_i^\diamond$ in the protocol do the same local computation, then it is easy to keep the memory aligned. If some parties carry out local operations and others do not, then it is better to keep the local operation outcomes in a separate memory region to ensure alignment for the inputs to $\mathcal{F}_p^\diamond$.

So far, the protocol description in $\Pi^\diamond$ did not explicitly consider the local computations and they were internal operations to $\mathcal{I}_i^\diamond$. Here, it is reasonable to bring them to focus using the local functionality ideas in Section 3.5. By definition, a local functionality $\mathcal{G}_q^{\bowtie}$ is a collections of machines $\{\mathcal{G}_{q,j}^{\bowtie}\}_{j\in\mathcal{J}_q}$ implementing local operations for party $\mathcal{P}_j$. These are now separated from $\mathcal{I}_i^\diamond$ and $\mathcal{M}_i^\diamond$ so that these machines become $\mathcal{I}_i^{\bowtie}$ and $\mathcal{M}_i^{\bowtie}$ that communicate with $\mathcal{G}_{q,i}^{\bowtie}$ as in Figure 38.

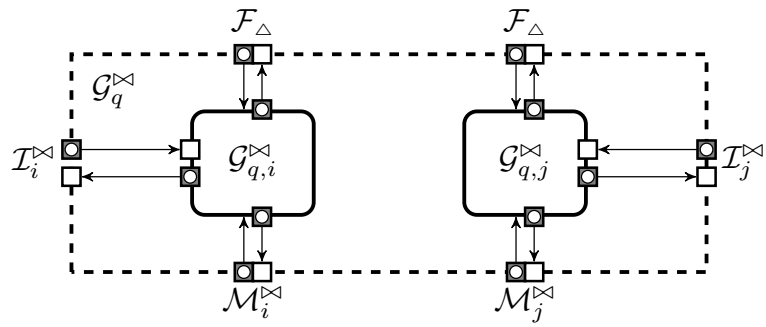


Figure 38: Encapsulation of local computations involving two parties.

Every time $\mathcal{I}_i^{\bowtie}$ has some local computation, it sends (t, α, β) to $\mathcal{G}_q^{\bowtie}$. The local operation $\mathcal{G}_{q,i}^{\bowtie}$ performs the memory lookup for input addresses α in $\mathcal{M}_i^{\bowtie}$, does the computation, writes the result to address β in $\mathcal{M}_i^{\bowtie}$ and gives the control back to $\mathcal{I}_i^{\bowtie}$. The instance t can simply be the instance of the calling protocol. The instance can also be unified by different parties like the instances used by ideal functionalities $\mathcal{F}_p$. In such case, the instance can be used to define $\mathcal{G}_q$ more explicitly as this is the version used in the following. In any case, the instance is not directly by $\mathcal{G}_{q,i}^{\bowtie}$. Local operations $\mathcal{G}_{q,i}^{\bowtie}$ may use the setup parameters of the party $\mathcal{P}_i$. Hence, there are buffers between $\mathcal{G}_q^{\bowtie}$ and $\mathcal{F}_{\triangle}$.

Definition 41 (Local operations memory-alignment)**.** A well-formed implementation of a protocol uses a local functionality $\mathcal{G}_q^{\bowtie}$ in memory-aligned manner, if each individual input of $\mathcal{G}_{q,i}^{\bowtie}$ is coming from $\mathfrak{gs}[t, \delta_1, \ell_1]$ in $\mathcal{M}_i^{\bowtie}$ and each output is written to $\mathfrak{gs}[t, \delta_2, \ell_2]$ in $\mathcal{M}_i^{\bowtie}$.

Note that the local operations can have a different set of parties that carry them out. Especially, achieving overall memory-alignment in case parties have different local functionality calls means that each party must have a local copying functionality to simply copy $\mathfrak{gs}[t, \delta_1, \ell_1]$ to $\mathfrak{gs}[t, \delta_1, \ell_2]$. When local operations are removed from the interpreter $\mathcal{I}_i^{\bowtie}$, it still has to perform conditional jumps in the code and, therefore, may need to read the respective values from $\mathcal{M}_i^{\bowtie}$. However, explicitly considering local functionalities allows us to specify further which protocols we want to consider for $\mathcal{I}_i^{\bowtie}$.

Definition 42 (Canonical protocol specification)**.** Let $\mathcal{F}_1^{\diamond}, \ldots, \mathcal{F}_k^{\diamond}, \mathcal{F}_{\text{io}}^{\diamond}, \mathcal{G}_1^{\bowtie}, \ldots, \mathcal{G}_m^{\bowtie}$ denote ideal and local functionalities used in a well-formed protocol specification. A protocol specification is in a canonical form if the following holds.

(a) All conditional jumps in $\mathcal{I}_i^{\bowtie}$ are based on values in the public storage domain or local storage domain for $\mathcal{P}_i$.

(b) All local operations except conditional jumps are implemented with $\mathcal{G}_1^{\bowtie}, \ldots, \mathcal{G}_m^{\bowtie}$.

(c) All ideal functionalities $\mathcal{F}_p$ and local functionalities $\mathcal{G}_q$ are used in a memory-aligned manner.

These conditions must hold regardless of the adversarial behaviour.

Note that any well-formed protocol can be represented in a canonical form when considering a simplistic adversary. The simplistic adversary cannot mess up memory accesses and computations of the interpreter. As discussed before, memory alignment can be achieved by separating the memory regions for inputs of ideal functionalities from the intermediate results of the local functionalities and using a local copying functionality. Overall, the program of the interpreters is public and the addressing scheme can be agreed upon and coded into the program. The separation of local functionalities does not change the capabilities of the interpreter $\mathcal{I}_i^{\bowtie}$ in any way and they are clocked instantaneously. Therefore, it also does not change the timing. Hence, condition (b) is also achievable. Condition (a) was already specified before.

Note that the canonical protocol specification is defined for the protocol as represented in $\Pi^{\diamond}$. However, the conditions can be translated to requirements on the initial protocol Π as the protocol execution has not changed. The components of a protocol have simply become more explicit. Firstly, $\mathcal{I}_i$ and $\mathcal{I}_i^{\bowtie}$ execute the same code. Hence, condition (a) has to already hold for $\mathcal{I}_i$. Secondly, condition (b) is not restrictive for a protocol in Π as any computation carried out by $\mathcal{I}_i$ can be represented as a local functionality for the party $\mathcal{P}_i$. Thirdly, the memory alignment condition (c) can be achieved in Π with the same approaches as in $\Pi^{\diamond}$.

Note that, in the canonical protocol specification, the interpreter $\mathcal{I}_i^{\bowtie}$ could be isolated from the trusted setup $\mathcal{F}_{\triangle}$. In such a case, the parameters from $\mathcal{F}_{\triangle}$ are not stored in $\mathcal{M}_i^{\bowtie}$. Hence, the reason why $\mathcal{F}_{\triangle}$ was defined (Definition 6) so that the adversary could query the corrupted parties setup from $\mathcal{F}_{\triangle}$ directly.

5.4.5. Joining the Memories of Several Parties

Canonical protocol specification defines protocols that also have aligned memory. Hence, the whole protection domain $\mathcal{F}_{\mathsf{pd}}^{\diamond}$ operates so that each call to $\mathcal{F}_p^{\diamond}$ uses the same address ℓ for all parties. This means that the individual memory machines $\mathcal{M}_1^{\diamond},\ldots,\mathcal{M}_n^{\diamond}$ can easily be combined to a single machine $\mathcal{M}^{\bowtie}$ with internal state $\mathfrak{gs}$. Instead of fetching the relevant address from all parties, a modified ideal functionality $\mathcal{F}_p^{\bowtie}$ can simply access $\mathfrak{gs}[t,\delta,\ell]$ to either read inputs or write outputs as in Figure 39. From now on, it is, therefore, easier to focus on the decomposed ideal functionalities where the machines $\mathcal{R}_u$ and $\mathcal{S}_u$, that are part of the ideal functionality, are replaced by $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ that communicate with the shared memory $\mathcal{M}^{\bowtie}$. Similarly, $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{S}}$ components of the decomposed ideal functionality are replaced by $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ that collect memory addresses instead of the messages.

The execution of $\mathcal{F}_p^{\bowtie}$ containing $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ and communicating with $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ is similar to the overall execution of the ideal functionalities. $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ collects incoming locations as (t_1,t_2,α). When all inputs for the given round of instance t_2 have arrived, then the input addresses are passed to $\mathcal{R}_u^{\bowtie}$. $\mathcal{R}_u^{\bowtie}$ reads these values from $\mathcal{M}^{\bowtie}$ as $\mathfrak{gs}[t,\delta_i,\ell_i]$ for all $(\delta_i,\ell_i)\in\alpha$, reconstructs the underlying values using $\mathcal{R}_{\delta}$ and sends the values to $\mathcal{T}_{\mathcal{R}}^{\bowtie}$. Then $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ sends the output location β to

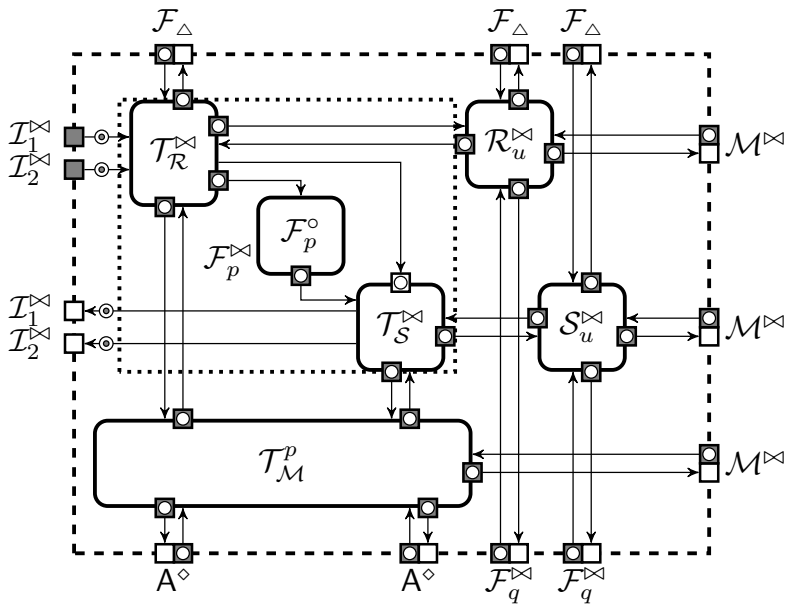


Figure 39: Decomposed canonical ideal functionality $\mathcal{F}_p^{\bowtie}$ communicating with the memory $\mathcal{M}^{\bowtie}$ through universal machines $\mathcal{S}_u^{\bowtie}$ and $\mathcal{R}_u^{\bowtie}$.

$\mathcal{T}_{\mathcal{S}}^{\bowtie}$ and gives the input value and instance to $\mathcal{F}_p^{\circ}$. The functionality $\mathcal{F}_p^{\circ}$ computes the output and gives it to $\mathcal{T}_{\mathcal{S}}^{\bowtie}$. $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ also clocks the β from $\mathcal{T}_{\mathcal{R}}^{\bowtie}$. It then uses $\mathcal{S}_u^{\bowtie}$ to get the shares of the output. For every $(\delta_i, \ell_i) \in \beta$ $\mathcal{S}_u^{\bowtie}$ gets (t, δ_i, ℓ_i, y) and writes the result of $\mathcal{S}_\delta$ with input y to $\mathfrak{gs}[t, \delta_i, \ell_i]$ in $\mathcal{M}^{\bowtie}$. The control then goes to $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ that writes output messages to $\mathcal{I}_i$.

The machines $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ see only the memory locations and the values. Hence, if the adversary expects to see any shares from them, then the already described machines cannot provide them and a special machine $\mathcal{T}_{\mathcal{M}}^{p}$ is needed. $\mathcal{T}_{\mathcal{M}}^{p}$ is essentially a simulator to turn adversaries A against $\mathcal{F}_p^{\diamond}$ to adversaries against $\mathcal{F}_p^{\bowtie}$. The main goal for $\mathcal{T}_{\mathcal{M}}^{p}$ is to enable the adversary to ask for the shares and then instead read them from $\mathcal{M}^{\bowtie}$. $\mathcal{F}_p^{\bowtie}$ is the collection $\mathcal{T}_{\mathcal{R}}^{\bowtie}, \mathcal{F}_p^{\circ}, \mathcal{T}_{\mathcal{S}}$. $\mathcal{T}_{\mathcal{M}}^{p}$ and $\mathsf{A}^{\diamond}$ against $\mathcal{F}_p^{\diamond}$ are the new equivalent adversary $\mathsf{A}^{\bowtie}$ against $\mathcal{F}_p^{\bowtie}$. $\mathcal{T}_{\mathcal{M}}^{p}$ manages cases where $\mathcal{T}_{\mathcal{R}}$ or $\mathcal{T}_{\mathcal{S}}$ give shares of corrupted parties to A, but $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ or $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ give the addresses to the corrupted parties memory. If $\mathcal{T}_{\mathcal{M}}^{p}$ receives a memory address from $\mathcal{F}_p^{\bowtie}$ then $\mathcal{T}_{\mathcal{M}}^{p}$ fetches the values from $\mathcal{M}^{\bowtie}$ to give to A. In total, both adversaries $\mathsf{A}^{\diamond}$ and $\mathsf{A}^{\bowtie}$ get the same access to $\mathcal{M}_i^{\bowtie}$ and see the same values when interacting with the canonical ideal functionality. The ideal functionality limits which values are revealed from the memory. $\mathcal{T}_{\mathcal{M}}^{p}$ is only allowed to read the memory $\mathcal{M}^{\bowtie}$ and cannot write anything to memory directly. If $\mathcal{T}_{\mathcal{M}}^{p}$ receives any other communication from $\mathsf{A}^{\diamond}$ or $\mathcal{F}_p^{\bowtie}$, such as a corruption request or abort message, it simply forwards it to $\mathcal{F}_p^{\bowtie}$. A simplistic adversary $\mathsf{A}^{\diamond}$ is still allowed to overwrite protocol inputs in $\mathcal{M}^{\bowtie}$, but this is done outside of the interactions with $\mathcal{F}_p^{\bowtie}$.

The resulting adversary $\mathsf{A}^{\bowtie}$ is still simplistic with the additional power to read

the memory $\mathcal{M}^{\bowtie}$ for corrupted parties and for the addresses received from $\mathcal{F}_p^{\bowtie}$. Canonical ideal functionalities $\mathcal{F}_p^{\bowtie}$ only give the adversary reading access to the memory of the corrupted parties. By definition, adversary $\mathsf{A}^{\bowtie}$ can read the memory locations for corrupted parties. Hence, this interaction with canonical $\mathcal{F}_p^{\bowtie}$ does not in any way invalidate the properties of the simplistic adversary. Note that it is possible and, for reasonable corruption models and canonical $\mathcal{F}_p^{\bowtie}$, meaningful to define corruption in $\Pi^{\bowtie}$ so that if the adversary accesses the memory for a party, then this party is considered corrupted.

In order to define the full protection domain $\mathcal{F}_{pd}^{\bowtie}$, the machine $\mathcal{F}_{\mathrm{io}}^{\diamond}$ is modified to $\mathcal{F}_{\mathrm{io}}^{\bowtie}$ that writes the values to $\mathcal{M}^{\bowtie}$ and sends the location information to the modified interpreter $\mathcal{I}^{\bowtie}$ that interacts with $\mathcal{M}^{\bowtie}$ instead of $\mathcal{M}_i^{\diamond}$. The machines $\mathcal{G}_i^{\bowtie}$ are also wired to $\mathcal{M}^{\bowtie}$. Define $\Pi^{\bowtie}$ as an extended collection consisting of machines $\mathcal{I}_1^{\bowtie},\ldots,\mathcal{I}_n^{\bowtie},\mathcal{M}^{\bowtie},\mathcal{F}_1^{\bowtie},\ldots,\mathcal{F}_k^{\bowtie},\mathcal{G}_{1,1}^{\bowtie},\ldots,\mathcal{G}_{g,n}^{\bowtie},\mathcal{F}_{\mathrm{io}}^{\bowtie}$ with all attached buffers. Let $\phi_{\bowtie}(\mathsf{A}^{\diamond})$ be the collection consisting of $\mathsf{A}^{\diamond}$, and $\mathcal{T}_{\mathcal{M}}^1,\ldots,\mathcal{T}_{\mathcal{M}}^k$. Let $\phi_{\bowtie}^*(\mathsf{A}^{\bowtie})$ be the semi-inverse of $\phi_{\bowtie}$ as described in the following theorem. The protocol $\Pi^{\bowtie}$ is illustrated in Figure 40.

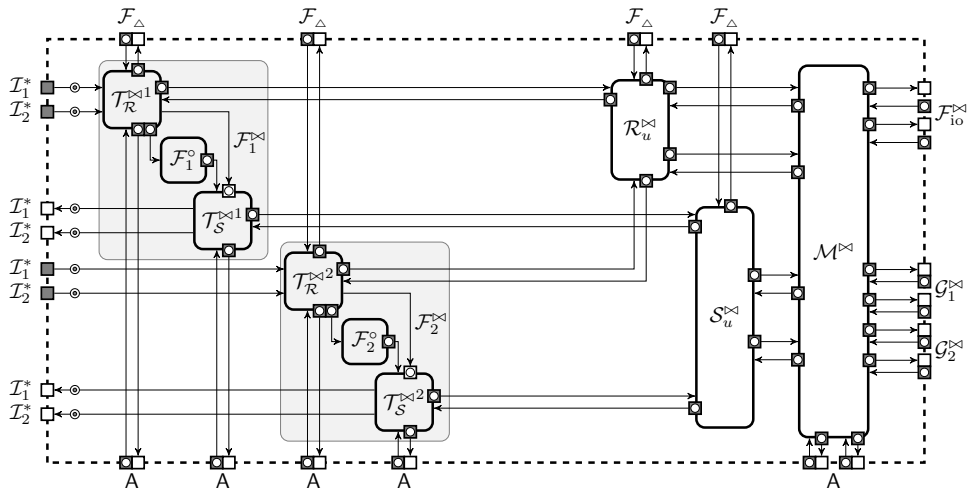


Figure 40: Protocol representation $\Pi^{\bowtie}$ with canonical ideal functionalities, $\mathfrak{F}_0$.

Theorem 15. *Let Π be a well-formed protocol specification that is in a canonical form and let $\mathbb{E}_{\Pi}$ be the set of compatible environments. Let $\Pi^{\diamond}$ and $\Pi^{\bowtie}$ be the respective modified collections of the protocol Π. Then*

$$\forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \quad \forall \mathsf{A}^{\diamond} \in \mathbb{A}^{\diamond}_{\Pi^{\diamond},\mathsf{Env}} : \qquad \mathsf{Env}\langle \Pi^{\diamond}, \mathsf{A}^{\diamond}\rangle \equiv \mathsf{Env}\langle \Pi^{\bowtie}, \phi_{\bowtie}(\mathsf{A}^{\diamond})\rangle$$
$$\forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \quad \forall \mathsf{A}^{\bowtie} \in \mathbb{A}^{\bowtie}_{\Pi^{\bowtie},\mathsf{Env}} : \qquad \mathsf{Env}\langle \Pi^{\bowtie}, \mathsf{A}^{\bowtie}\rangle \equiv \mathsf{Env}\langle \Pi^{\diamond}, \phi^*_{\bowtie}(\mathsf{A}^{\bowtie})\rangle \ ,$$

where $\mathbb{A}^{\diamond}_{\Pi^{\diamond},\mathsf{Env}}$ is the set of simplistic adversaries compatible with the protocol $\Pi^{\diamond}$ and the environment Env. *The resulting adversary $\phi_{\bowtie}(\mathsf{A}^{\diamond})$ is also simplistic. $\mathbb{A}^{\bowtie}_{\Pi^{\bowtie},\mathsf{Env}}$ is the set of simplistic adversaries compatible with $\Pi^{\bowtie}$ that only read the corrupted parties values from $\mathcal{M}^{\bowtie}$.*

Proof. The buffers between $\mathcal{I}_j^{\diamond}$ and $\mathcal{I}_j^{\bowtie}$ and the adversary are the same in both collections, and the adversary gets the same access to the state of the interpreters in either $\mathcal{M}_i^{\diamond}$ or $\mathcal{M}^{\bowtie}$. The difference is that $\mathcal{M}_i^{\diamond}$ are all joined to a single $\mathcal{M}^{\bowtie}$ that has a state $\mathcal{G}_i^{\bowtie}$. Also, the local functionalities $\mathcal{G}_i^{\bowtie}$ are separated from $\mathcal{I}_i^{\diamond}$. In addition, the interface to $\mathcal{F}_{\triangle}$ is changed to allow for $\mathcal{G}_i^{\bowtie}$ to be separated. However, this does not change the view of the adversary or the environment.

A well-formed protocol writes all memory addresses at most once. Therefore, the outputs of $\mathcal{G}_i^{\bowtie}$ in $\mathcal{M}^{\bowtie}$ are never overwritten. The buffers connecting $\mathcal{G}_{i,q}^{\bowtie}$ to $\mathcal{I}_i^{\bowtie}$ and $\mathcal{M}^{\bowtie}$ are sender-clocked, hence separating $\mathcal{G}_{i,q}^{\bowtie}$ from $\mathcal{I}_i^{\diamond}$ is not introducing new adversarial controls and is invisible to the adversary, if the states of $\mathcal{I}_i^{\diamond}$ and $\mathcal{I}_i^{\bowtie}$ remain the same. Thanks to the self-clocked channels, the functionality $\mathcal{G}_{i,q}^{\bowtie}$ is executed in the same timeframe as the same computation internal to $\mathcal{I}_i^{\diamond}$.

For $\mathcal{F}_p^{\diamond}$, the exact function was unspecified before other than inputs and outputs are read from $\mathcal{M}_i^{\diamond}$. For a canonical $\mathcal{F}_p^{\diamond}$, the decomposed look can always be considered, and $\mathcal{S}_u$ and $\mathcal{R}_u$ can be replaced by $\mathcal{S}_u^{\bowtie}$ and $\mathcal{R}_u^{\bowtie}$.

The modifications of the adversary $\mathsf{A}^{\diamond}$ to $\phi_{\diamond\to\bowtie}(\mathsf{A}^{\diamond})$ only affect the interaction with $\mathcal{F}_p^{\diamond}$ and $\mathcal{F}_p^{\bowtie}$ respectively. The clocking rules and writing of $\mathcal{M}_i^{\diamond}$ or $\mathcal{M}^{\bowtie}$ for the corrupted parties state satisfy the same rules in both cases, and hence, a simplistic adversary $\mathsf{A}^{\diamond}$ gives rise to a simplistic $\phi_{\diamond\to\bowtie}(\mathsf{A}^{\diamond})$.

The reverse equivalence from $\Pi^{\bowtie}$ to $\Pi^{\diamond}$ and $\mathsf{A}^{\bowtie}$ to equivalent $\mathsf{A}^{\diamond}$ is straightforward for $\mathsf{A}^{\bowtie}$ that only access the memory $\mathcal{M}^{\bowtie}$ for the values of the corrupted parties. The transformation requires adding $\mathcal{T}_{\mathcal{M}}^{p}$ to $\mathcal{F}_p^{\bowtie}$ in the construction of $\Pi^{\diamond}$ and pushing the local functionalities back to $\mathcal{I}_i^{\diamond}$. Note that if the adversary reads a memory location before it has clocked the output of $\mathcal{F}_p^{\bowtie}$ that is written to that location, then it has access to that memory through the ideal functionality. If it has clocked the output, then the party has also received it, and the access can be through interaction with the party. □

The collection $\Pi^{\bowtie}$ finishes the transformation of the protocol representation to the model with shared memory. The machine $\mathcal{M}^{\bowtie}$ is the shared memory used by all parties to communicate or store the values of all protection domains in the protocol.

5.5. Abstract Memory Model and Semi-Abstract Adversaries

The collection $\Pi^{\bowtie}$ using protection domain $\mathcal{F}_{\mathsf{pd}}^{\bowtie}$ contains a memory machine $\mathcal{M}^{\bowtie}$ that can be accessed by most machines in the description to read and write necessary values. This section continues with the simplistic adversary $\mathsf{A}^{\bowtie}$ that is restricted to only accessing the values belonging to corrupted parties in $\mathcal{M}^{\bowtie}$. In addition, the adversary controls the timing of the execution of $\Pi^{\bowtie}$. A simplistic adversary is restricted to only overwriting the $\mathcal{M}^{\bowtie}$ in the locations of inputs to some ideal functionality $\mathcal{F}_p^{\bowtie}$. This section simplifies the adversary further to semi-abstract adversaries that do not require read access to shares in $\mathcal{M}^{\bowtie}$ but can

still modify the values in the storage if allowed by the modification awareness and limited control properties of the storage domain. To discuss this transformation, the memory $\mathcal{M}^{\bowtie}$ is split into two parts, $\mathcal{M}_0$ for storing the values and $\mathcal{M}^*$ for storing the shares. Then, step-by-step, the need for $\mathcal{M}^*$ is reduced until the honest execution uses only $\mathcal{M}^*$ and the adversary can simulate $\mathcal{M}^*$.

Formally, this section defines a $*$-operator that acts on protocols and their components together with a universal $\phi_* : \mathbb{A}^{\bowtie} \to \mathbb{A}^*$ and its semi-inverse $\phi_*^* : \mathbb{A}^* \to \mathbb{A}^{\bowtie}$ that achieves

$$\forall \Pi \in \mathbb{P}_{\mathsf{c}} : \ \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \ \forall \mathsf{A}^{\bowtie} \in \mathbb{A}^{\bowtie} : \mathsf{Env}\langle \Pi, \mathsf{A}^{\bowtie} \rangle \equiv \mathsf{Env}\langle \Pi^{sa}, \phi_*(\mathsf{A}^{\bowtie}) \rangle \quad (5.10)$$

$$\forall \Pi \in \mathbb{P}_{\mathsf{c}} : \ \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \ \forall \mathsf{A}^{*} \in \mathbb{A}^{*} : \mathsf{Env}\langle \Pi^{sa}, \mathsf{A}^{*} \rangle \equiv \mathsf{Env}\langle \Pi, \phi_*^*(\mathsf{A}^{*}) \rangle, \quad (5.11)$$

where $\mathbb{P}_{\mathsf{c}}$ is the set of protocols in a canonical form specified in $\mathcal{F}_{\mathsf{pd}}^{\bowtie}$ and $\mathbb{E}_{\Pi}$ is the set of compatible environments. Π^{sa} is the semi-abstract protocol and $\mathbb{A}^*$ is the class of semi-abstract adversaries (Definition 46). The forward transformation is covered step-by-step in the whole section, and the reverse is summarised in Lemma 21.

5.5.1. Abstract Memory Model

This section introduces an abstract memory $\mathcal{M}_0$ that keeps the values corresponding to the storage domain representation in $\mathcal{M}^{\bowtie}$. The memory $\mathcal{M}^{\bowtie}$ is modified to $\mathcal{M}^*$ that interacts with $\mathcal{M}_0$ to keep track of any changes. The state of $\mathcal{M}_0$ is $\mathfrak{s}_0[t,\delta,\ell] = m$ containing the values corresponding to $\mathfrak{gs}[t,\delta,\ell]$ whenever the memory is accessed. The modifications of machines $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ called $\mathcal{R}_u^*$ and $\mathcal{S}_u^*$ are placed between $\mathcal{M}_0$ and $\mathcal{M}^*$ to allow for the synchronisation of the two memory components. Since $\mathcal{F}_p^{\bowtie}$ only requires the use of values and not the shares, it is connected to $\mathcal{M}_0$ and does not explicitly communicate with $\mathcal{R}_u^{\bowtie}$ and $\mathcal{S}_u^{\bowtie}$ anymore. This setup is shown in Figure 41. The local variables and state of $\mathcal{I}_i^{\bowtie}$ are also stored in $\mathcal{M}_0$. By definition, $\mathcal{M}_0$ also contains the values that are in local protection domains in $\mathcal{M}^*$. $\mathcal{I}_i^{\bowtie}$ is modified to $\mathcal{I}_i^*$ that accesses its state from $\mathcal{M}_0$.

The abstract memory $\mathcal{M}_0$ is also connected to the adversary and allows it to read the local variables of the corrupted parties. All memory modifications that the adversary can do are carried out in $\mathcal{M}^*$. The two memory machines work together to keep their states synchronised if $\mathcal{F}_p^{\bowtie}$ or $\mathcal{I}_i^*$ update $\mathcal{M}_0$ or $\mathcal{G}_{i,q}^{\bowtie}$, A or $\mathcal{F}_{\text{io}}^{\bowtie}$ update $\mathcal{M}^*$. When A, $\mathcal{G}_q^{\bowtie}$ or $\mathcal{F}_{\text{io}}^{\bowtie}$ queries a share from a location (t,δ,ℓ) and $\mathfrak{gs}[t,\delta,\ell]$ is empty or invalid $\mathcal{M}^*$ writes $\textsc{Share}(t,\delta,\ell)$ to the buffer to $\mathcal{M}_0$ and clocks it. Then, $\mathcal{M}_0$ uses $\mathcal{S}_u^*$ to share $\mathfrak{s}_0[t,\delta,\ell]$ and give control back to $\mathcal{M}^*$. $\mathcal{M}^*$ now has synchronized with $\mathfrak{s}_0$ and can answer the query for (t,δ,ℓ).

When $\mathfrak{gs}[t,\delta,\ell]$ is updated, then $\mathcal{M}^*$ writes $\textsc{Update}(t,\delta,\ell)$ to the synchronisation buffer to $\mathcal{M}_0$. After receiving this, $\mathcal{M}_0$ uses $\mathcal{R}_u^*$ to read the shares from $\mathfrak{gs}$ and update $\mathfrak{s}_0[t,\delta,\ell]$ to the new value. If $\mathcal{I}^*$ or $\mathcal{F}_p^{\bowtie}$ update $\mathfrak{s}_0$ in $\mathcal{M}_0$, then $\mathcal{M}_0$ writes $\textsc{Invalid}(t,\delta,\ell)$ to the synchronisation buffer for $\mathcal{M}^*$ so that $\mathcal{M}^*$ knows to update the storage domains values when they are queried. $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ can query $\mathcal{M}_0$ with

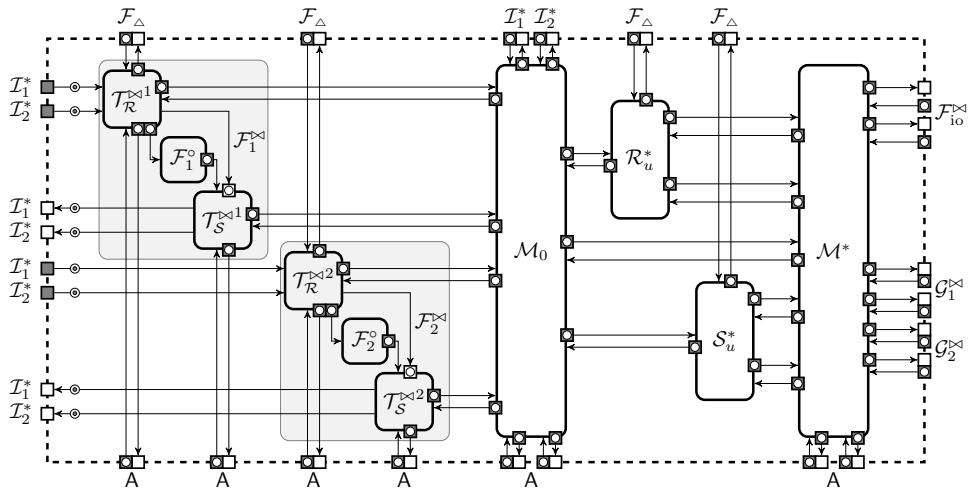


Figure 41: Separating memory $\mathcal{M}^\bowtie$ to $\mathcal{M}_0$ for values and $\mathcal{M}^*$ for shares. Collection $\mathfrak{F}_1$.

(t,δ,ℓ) to get the value $\mathfrak{s}_0[t,\delta,\ell]$ the same way it usually queries $\mathcal{R}_u^\bowtie$. Similarly, $\mathcal{T}_\mathcal{S}^\bowtie$ sends the value to $\mathcal{M}_0$ similarly as $\mathcal{T}_\mathcal{S}$ sent it to $\mathcal{S}_u^\bowtie$ before.

Note that there are some steps when $\mathfrak{gs}$ is not properly updated using the previous description. This is due to the fact that $\mathcal{G}_q^\bowtie$ and $\mathcal{F}_{\text{io}}^*$ consecutively update all shares in the location (t,δ,ℓ) depending on how the inputs arrive to $\mathcal{F}_{\text{io}}^\bowtie$ or how $\mathcal{I}_i^*$ call out $\mathcal{G}_{i,q}^\bowtie$. At a single update, the reconstruction might fail if other shares are still missing and, due to the failure, there is no value in $\mathcal{M}_0$. This incompleteness of $\mathcal{M}_0$ could be mitigated by changing the execution so that $\mathcal{M}^*$ sends the update message when all shares have been written. However, this change does not affect the rest of the execution. By definition, $\mathcal{F}_p^\bowtie$ can only query this value from $\mathcal{M}_0$ if it has collected all necessary inputs to reconstruct the value. The inputs are collected from parties $\mathcal{P}_i$ through interpreter $\mathcal{I}_i^*$ and, therefore, either the $\mathcal{G}_{i,q}^\bowtie$ for this party $\mathcal{P}_i$ has executed or $\mathcal{F}_{\text{io}}^\bowtie$ has sent the input to party $\mathcal{P}_i$. Hence, this partial update to $\mathcal{M}_0$ will go unnoticed by the execution of other machines in the collection as the value is only used when it can be reconstructed from $\mathfrak{gs}$ in $\mathcal{M}^*$. The same logic applies to the adversary view. The adversary can see the local variables of the corrupted party in $\mathcal{M}_0$ and they can always appear in $\mathcal{M}_0$ as soon as they are in $\mathcal{M}^*$.

Let $\mathfrak{F}_0$ be an extended collection corresponding to $\Pi^\bowtie$ from the previous section in Figure 40 so that is consist of $\mathcal{F}_1^\bowtie,\ldots,\mathcal{F}_p^\bowtie,\mathcal{R}_u^\bowtie,\mathcal{S}_u^\bowtie,\mathcal{M}^\bowtie,\mathcal{F}_{\text{io}}^\bowtie$. Let $\mathfrak{F}_1$ be an extended collection consisting of $\mathcal{F}_1^\bowtie,\ldots,\mathcal{F}_p^\bowtie,\mathcal{R}_u^*,\mathcal{S}_u^*,\mathcal{M}_0,\mathcal{M}^*,\mathcal{F}_{\text{io}}^\bowtie$ as shown in Figure 41. Then, for the observer consisting of A, $\mathcal{I}_i^\bowtie$, Π_{e}, $\mathcal{G}_q^\bowtie$ and $\mathcal{F}_\triangle$, these two collections are equivalent.

Theorem 16. *Collections $\mathfrak{F}_0$ and $\mathfrak{F}_1$ are indistinguishable, provided that they are executing a well-formed protocol specification in a canonical form.*

Proof. This proof first states the basic equivalences of the two collections. These hold in the beginning of the execution or refer to the structure of the collections. The focus is then on the differences that might occur in the execution of the two collections.

Communication with the machines $\mathcal{M}_0$, $\mathcal{M}^*$, $\mathcal{R}_u^*$ and $\mathcal{S}_u^*$ is sender-clocked and does not add new clocking capabilities to the adversary. Machines $\mathcal{S}_u^*$ and $\mathcal{S}_u^{\bowtie}$ ($\mathcal{R}_u^*$ and $\mathcal{R}_u^{\bowtie}$) have identical interfaces with $\mathcal{F}_{\triangle}$. Hence, communication with $\mathcal{F}_{\triangle}$ remains unchanged. Hence, it is sufficient to focus on the interaction of the memory and the ideal functionalities. It is important that all machines read the same values from the memory and that all modifications of the memory are equivalent. A significant difference between the collections is the separation of adversary connecting to $\mathcal{M}_0$ and $\mathcal{M}^*$ instead of one connection to $\mathcal{M}^{\bowtie}$ in $\mathfrak{F}_0$. This difference can be managed easily by merging or simulating the extra channels as necessary.

The state of the party, such as which DMACALL calls are outstanding and where it has progressed with its execution, is kept in memory. If $\mathcal{I}_i^{\bowtie}$ and $\mathcal{I}_i^*$ are in equivalent states, then the same state is stored in $\mathcal{M}^{\bowtie}$ and $\mathcal{M}_0$, respectively. Hence, also the adversary gets the same state when corrupting the party. However, the shares of the party must be read from $\mathcal{M}^*$ in $\mathfrak{F}_1$.

Hence, the rest of the proof needs to consider the views of the parties querying $\mathfrak{gs}[t,\delta,\ell]$ or $\mathfrak{s}_0[t,\delta,\ell]$ to verify that they get equivalent results. Initially, the memories are empty. Assume that so far, the states have been equivalent. Initially, all memory locations are unassigned. The value in $\mathcal{M}_0$ can be queried by $\mathcal{T}_{\mathcal{R}}^{\bowtie}$. By construction of the synchronisation between $\mathcal{M}^*$ and $\mathcal{M}_0$, if the value in $\mathfrak{s}_0[t,\delta,\ell]$ or $\mathfrak{gs}[t,\delta,\ell]$ is changed, then the other in invalidated and recomputed from the updated value. Hence, the reply to $\mathfrak{s}_0[t,\delta,\ell]$ contains the reconstruction of $\mathfrak{gs}[t,\delta,\ell]$ and each query $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ makes to $\mathcal{R}_u^{\bowtie}$ in $\mathfrak{F}_0$ contains the same reconstructed result by definition.

The state in $\mathfrak{gs}$ can be queried and modified by $\mathcal{G}_q^{\bowtie}$, A and $\mathcal{F}_{\text{io}}^{\bowtie}$. The reads are always consistent thanks to the synchronisation process summarised in the previous paragraph. The writing to memory still needs to be discussed. If $\mathcal{S}_u^{\bowtie}$ was the last machine to update $\mathfrak{gs}[t,\delta,\ell]$ in $\mathfrak{F}_0$, then $\mathcal{S}_u^*$ was also the last to update $\mathfrak{gs}[t,\delta,\ell]$ in $\mathfrak{F}_1$. Hence, these values have the same distribution as they are computed by $\mathcal{S}_\delta$ and can have the same value if the randomness used by $\mathcal{S}_\delta$ is aligned in the two collections. In case $\mathcal{G}_q^{\bowtie}$ was the last to update the $\mathfrak{gs}[t,\delta,\ell]$, then it used the same inputs from $\mathfrak{gs}$ to compute this value and, if any randomness is used, then it can be aligned again to get the same value in both collections. $\mathcal{F}_{\text{io}}^{\bowtie}$ only writes the values received from Π_{e} to $\mathfrak{gs}$ and if the previous state of the collections $\mathfrak{F}_1$ and $\mathfrak{F}_2$ has been the same, then, so far, everything written to Π_{e} has been the same, and we can expect the same state updates from $\mathcal{F}_{\text{io}}^{\bowtie}$ in both collections. All updates to $\mathfrak{gs}$ invalidate the value in the same location in $\mathfrak{s}_0$ and it is recomputed using $\mathcal{R}_u^*$ and always returns the same value as the one stored in $\mathfrak{gs}$. □

5.5.2. Applying Modification Awareness in MPC Protocols

The storage domains define modification awareness (Definition 10) property that specifies that adversarial modifications of shares from $\mathcal{S}_\delta$ should have predictable consequences of the underlying value. In the current model, the adversary can change the corrupted parties' values in $\mathcal{M}^*$ and this will cause the value to change in $\mathcal{M}_0$. In order to reduce the importance of $\mathcal{M}^*$, it is necessary to consider the conditions under which the adversary can update $\mathcal{M}_0$ directly instead of allowing $\mathcal{R}_u^*$ to do the update after the adversary modifies $\mathcal{M}^*$. However, inside the protection domain, the storage domain is also affected by the local operations $\mathcal{G}_q^{\bowtie}$ and, therefore, the modification awareness definition cannot be applied without extra considerations. The goal of this section is still to use the modification extractor $\mathcal{E}$ from the modification awareness property to modify the values in $\mathcal{M}_0$ in a collection $\mathfrak{F}_2$ (Figure 42). However, the success of $\mathcal{E}$ needs to be re-evaluated to account for the added local operations.

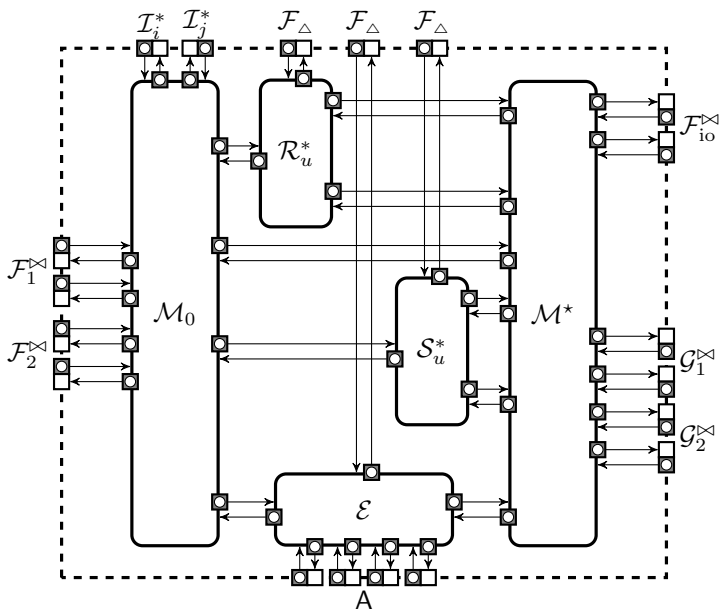


Figure 42: Adding a modification extractor $\mathcal{E}$ between $\mathcal{M}_0$ and $\mathcal{M}^\star$ to transform $\mathfrak{F}_1$ to $\mathfrak{F}_2$.

The behaviour of $\mathcal{M}^*$ is also changed in $\mathfrak{F}_2$. Hence it is replaced by $\mathcal{M}^\star$. The execution of the new memory management only affects adversarial modifications of $\mathfrak{gs}$. The extractor $\mathcal{E}$ forwards all messages from A to $\mathcal{M}^\star$ and back. If the adversary modifies values in $\mathfrak{gs}$, then $\mathcal{E}$ uses the current values in $\mathcal{M}^\star$ and the new values sent by A to extract the modification Δ. $\mathcal{E}$ sends Δ and the memory location to $\mathcal{M}_0$ that updates its value $\mathfrak{s}_0[t,\delta,\ell]$ to $\mathfrak{s}_0[t,\delta,\ell] \oplus_\delta \Delta$ using the modification function for the storage domain δ. $\mathcal{M}^\star$ differs from $\mathcal{M}^*$ only so that it does not send the location invalidation to $\mathcal{M}_0$ when $\mathcal{E}$ overwrites some values in $\mathfrak{gs}$.

$\mathcal{M}_0$ accepts these blind modifications made by $\mathcal{E}$ to values in the storage based on the modification definition of the respective storage domain δ. The following defines unexpected and, in a sense, incorrect or indirect adversarial modifications as *oblique*.

Definition 43 (Oblique modification)**.** An adversarial modification of $\mathfrak{gs}[t, \delta, \ell]$ is oblique if the extractor $\mathcal{E}$ fails to correctly update the value in $\mathfrak{s}_0[t, \delta, \ell]$.

Oblique modifications lead to inconsistencies in the states of $\mathcal{M}_0$ and $\mathcal{M}^*$. Hence, it is important to consider how likely they are in any given protocol. Oblique modifications can easily occur for verifiable storage domains, especially if the adversary can corrupt the party in Π_{e} and give inconsistent shares as input to the protocol. If the adversary also knows the valid shares from Π_{e}, then it could easily modify the shares in $\mathfrak{gs}$ to the original shares. However, in this situation, the value in $\mathcal{M}_0$ is $\perp$ and by the modification definition, no value that the extractor $\mathcal{E}$ produces can change the $\perp$ to the valid value, as long as only the allowed set of parties is corrupted. This issue is similar to the condition of modification awareness that each value is overwritten at most once. Note that the simplistic adversary, by definition, does not modify values several times. Hence, a simplistic adversary does not cause such basic oblique modifications.

Theorem 17. *Collections $\mathfrak{F}_1$ and $\mathfrak{F}_2$ are indistinguishable, provided that the probability of oblique modifications is negligible.*

Proof. $\mathcal{F}_p^{\bowtie}$, $\mathcal{F}_{\text{io}}^{\bowtie}$, $\mathcal{S}_u^*$ and $\mathcal{R}_u^*$ are the same in the two collections. The only change is the communication between $\mathcal{M}^*$ ($\mathcal{M}^\star$) and $\mathcal{M}_0$. $\mathcal{E}$ forwards the communication between A and $\mathcal{M}^\star$ without modification. The collections can become distinguishable only if the values in $\mathcal{M}_0$ in $\mathfrak{F}_2$ are inconsistent with the state in $\mathcal{M}^*$ in $\mathfrak{F}_2$. This occurs only with negligible probability, as it only happens in the case of oblique modifications. □

By definition, oblique modifications cause the extractor to fail. If the storage domain has modification awareness, then the probability of an oblique modification on a fresh output from $\mathcal{S}_\delta$ is negligible by definition. However, local operations give shares that are not necessarily with the same distribution as shares from $\mathcal{S}_\delta$ and, therefore, their modifications may be more difficult to extract. However, the most common local operations, such as copying a value and performing linear combinations, do not cause significant problems. This is because they are *transparent* in the following sense.

Definition 44. A local operation $\mathcal{G}_i^{\bowtie}$ is transparent if any modification of output shares can be effectively converted to a modification of its inputs, and oblique modification goes to oblique modification.

If the local operation has several inputs, then this definition does not specify which input is the one where the modification is propagated. In principle, the modification may be propagated to several inputs. In this case, we require that an oblique modification of the output gives at least one oblique input modification.

As mentioned, the modification could be oblique due to adversarial actions or simply the behaviour of Π_{e}. Hence, to reason about the probability of the extractor failing, both the protocol description and the execution context need to be considered. The following lemma considers the restricted case where the inputs received from Π_{e} are always generated by $\mathcal{S}_\delta$.

Lemma 13. *Assume a protocol* Π *such that all local operations are meaningful and transparent and ideal functionalities are in a canonical form. Assume that the secret sharing functionality* $\mathcal{S}_\delta$ *generates all protocol inputs for storage domain* δ *and the inputs of* $\mathcal{S}_\delta$ *are determined by the adversary. Assume that separate storage domains have independent parameters. Then the probability of extraction failures in a well-founded program is negligible for simplistic adversaries, provided that every storage domain is modification-aware.*

Proof. Consider the simplistic adversary A that, with probability ε, manages to do an oblique modification for a storage domain δ in protocol Π. The following proof defines an adversary B against the modification awareness property of the storage domain δ using A. B plays the role of Π and $\mathcal{S}_\delta$ for A. B interacts with the modification awareness game and has access to $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ in that collection (Figure 10). B can run the trusted setup $\mathcal{F}_{\triangle}$ for all other storage domains $\tau \neq \delta$ in order to simulate them in the protocol Π as the setup parameters of separate domains are independent. B receives the protocol inputs from A. Using $\mathcal{S}_\delta$ in the modification-awareness game, B can perfectly simulate the protocol inputs. Note that B knows all the protocol inputs and can therefore compute the outputs for all canonical ideal functionalities and all meaningful local functionalities. Hence, it can simulate the execution of all canonical ideal functionalities by computing the output value in plain and using the respective sharing functionalities $\mathcal{S}_\delta$ or $\mathcal{S}_\tau$ to generate the outputs. It can also simulate the adversary and $\mathcal{F}_p^{\bowtie}$ interactions. For storage domains τ, B can see all the shares of all parties. Hence, B can also perform all the local computations in storage domains τ and can even run ideal functionalities $\mathcal{F}_p^{\bowtie}$ honestly if domain δ is not used. For storage domain δ, B can compute all local operations of corrupted parties. In addition, as B knows all input values and the operations are meaningful, B can also compute the output values of all local operations.

By definition, the simplistic adversary A modifies only the inputs to some ideal functionality $\mathcal{F}_p^{\bowtie}$. If this input was generated by $\mathcal{S}_\delta$, then B can try the same modification in the modification awareness game. If this input was generated by a local operation, then B can always back-propagate the modification to some of the input of the local functionality. The backpropagation can be iterated until the modified value is some value generated by $\mathcal{S}_\delta$ (e.g. the input from Π_{e} or output of some ideal functionality).

Hence, each modification that A does in the protocol leads to some modification that B can try in the modification awareness game. A can make such modifications in all places where ideal functionality has input from storage domain δ.

Let this number of inputs be m. Adversary B may not be able to check beforehand if a modification is oblique or not. Hence in the worst case, it has to choose a random modification to use as the modification in the modification awareness game. Hence, the probability of success for B becomes ε/m. If the storage domain δ is modification-aware, then ε/m is negligible, and so is ε, as m is bounded by the size of the protocol. □

If the protection domain does not use private setup, then it is easy to execute any $\mathcal{S}_\delta$ by just running it honestly. Hence, all computations that may happen in Π_e before the protocol could be simulated by B to extend the proof of Lemma 13. If the storage domain δ uses private setup parameters, then it is less straightforward to simulate Π_e, but it could be done similarly to the simulation of the protocol by using the $\mathcal{S}_\delta$ functionality in the modification-awareness game. Simulatability of the environment Π_e (Definition 31) is discussed in Section 5.2.2. The following corollary uses the simulatability of Π_e to extend Lemma 13 to lift the restriction that all inputs come from $\mathcal{S}_\delta$. Note that this is the first occurrence where the simulatability of Π_e is needed.

By definition, the protection domain Π_e is simulatable if its execution is indistinguishable from an execution with Sim and specialised $\mathcal{S}^{\mathsf{Sim}}$ and $\mathcal{R}^{\mathsf{Sim}}$. The latter is used to give inputs and receive outputs from the protocol Π so that the parameters from $\mathcal{F}_\triangle$ are the same as used in Π. For the rest, Sim can only access the parameters of the corrupted parties.

Corollary 8. *Assume that all local operations are meaningful and transparent, ideal functionalities are in a canonical form and the environment Π_e is simulatable. Then the probability of extraction failures in a well-founded program is negligible for simplistic adversaries, provided that every storage domain is modification-aware.*

Proof. The proof of Lemma 13 can be generalised to a class of environments that can be simulated in the modification awareness game. The protocol Π in Definition 31 is the subset of the protocol where the real modification awareness game is accessed and the rest in Π_e is simulated. B can interact with the real $\mathsf{Env}_{\mathsf{pd}}$ to learn the true inputs that the given environment would give. Then B uses Sim and $\mathcal{S}^{\mathsf{Sim}}$ for storage domains τ or $\mathcal{S}_\delta$ for storage domain δ to interact with Π. Note that the $\mathcal{S}_\delta$ in the modification awareness game allows B to learn the corrupted parties' shares the same as $\mathcal{S}^{\mathsf{Sim}}$ would. Moreover, as B sill simulate Π, B can simply pass the values from Π to Sim and skip $\mathcal{R}^{\mathsf{Sim}}$ (or $\mathcal{R}^{\delta}$). This way B plays the part of Π_e for A. For Π, the adversary B can proceed as in the proof of Lemma 13 as it still knows all input and output values that Sim exchanges with $\mathcal{S}^{\mathsf{Sim}}$ and $\mathcal{R}^{\mathsf{Sim}}$ and can similarly keep track of all the values in Π. □

5.5.3. Separating Local Functionalities

In the current collection $\mathfrak{F}_2$, the local operations are using the shares from memory $\mathcal{M}^\star$ and cause $\mathcal{M}^\star$ to update the state of $\mathcal{M}_0$. Meaningful local operations (Def-

inition 15) are such that the state of the $\mathcal{M}_0$ could be updated using the function that defines the local operation rather than reconstructing it from the shares.

Recall that a value $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathcal{M}_0$ is read mostly by the ideal functionality $\mathcal{F}_p^{\bowtie}$. If it is in the local protection domain, then it can also be read by $\mathcal{I}_i^*$ and. In addition, the adversary can modify this value using the extractor $\mathcal{E}$. If a local operation updates $\mathfrak{gs}$, then this modification can occur in $\mathfrak{s}_0$ as soon as the local operation $\mathcal{G}_q^{\bowtie}$ is finished, meaning that, for all participating parties $\mathcal{P}_i$, local functionality $\mathcal{G}_{i,q}$ has finished. If the value in $\mathfrak{s}_0$ is used by the canonical ideal functionality, then it does not read this value before the preceding operations are finished. The same is true for an interpreter executing a well-formed program.

If the adversary had corrupted more parties than allowed by the hiding property of the protection domain δ, then the value $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathcal{M}_0$ can also be read by the adversary. However, in such cases, the adversary also has knowledge about where the corrupted parties have reached with their protocol execution and whether or not this value is indeed already computed in $\mathcal{M}_0$ or if some preceding operations need to be computed before it can be evaluated.

Note that the execution of the components $\mathcal{G}_{i,q}^{\bowtie}$ of a local functionality $\mathcal{G}_q^{\bowtie}$ is not synchronised. Hence, it is possible that all corrupted parties have completed this computation, but some honest party has not. This means that if the adversary modifies the output of a local computation (when it is used as an input to some ideal functionality), the extractor can compute the modification for $\mathcal{M}_0$. However, the value may not yet be computed in $\mathcal{M}_0$. As argued before, the value in $\mathcal{M}_0$ is not used before all parties complete $\mathcal{G}_{i,q}^{\bowtie}$ and, thus, the modification can be delayed until the value exist.

These observations allow us to define a new collection $\mathfrak{F}_3$ (Figure 43) where a new machine $\mathcal{G}_q^0$ executes the meaningful local operation in $\mathcal{M}_0$. The new $\mathcal{G}_q^0$ also interacts with the modified interpreters $\mathcal{I}_i^\star$. The new version of $\mathcal{M}_0$ that interacts with $\mathcal{G}_q^0$ is defined as $\mathcal{M}_0^\star$. The behaviour $\mathcal{M}^\star$ is changed to form $\mathcal{M}^{\circledast}$ that does not invalidate the location $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathcal{M}_0^\star$ when the functionality $\mathcal{G}_q^{\bowtie}$ alters the location $\mathfrak{gs}[t,\delta,\ell]$ in $\mathcal{M}^{\circledast}$.

The new interactions are defined as follows. $\mathcal{I}_i^\star$ sends each DMACALL call to $\mathcal{G}_q^{\bowtie}$ also to $\mathcal{G}_q^0$ right after it gets the control back from $\mathcal{G}_q^{\bowtie}$. $\mathcal{G}_q^0$ checks if the output is already computed in $\mathcal{M}_0^\star$. If it is not computed, but all inputs are in $\mathcal{M}_0^\star$, then it computes and stores the output according to the function g_q defining the meaning of the functionality (Definition 15). $\mathcal{G}_q^0$ gives control back to $\mathcal{I}_i^\star$. The share memory $\mathcal{M}^{\circledast}$ does not invalidate the location when $\mathcal{G}_q^{\bowtie}$ operates. For all other occasions, it behaves like $\mathcal{M}^\star$.

Theorem 18. *Collection $\mathfrak{F}_2$ with $\mathcal{I}_i^*$ and $\mathcal{G}_q^{\bowtie}$ and collection $\mathfrak{F}_3$ with $\mathcal{I}_i^\star$ are indistinguishable provided that a well-formed protocol specification is in a canonical form and all local operations are meaningful and implement some deterministic functionality.*

Proof. Modifications to create $\mathfrak{F}_3$ from collection $\mathfrak{F}_2$ only eliminate the interac-

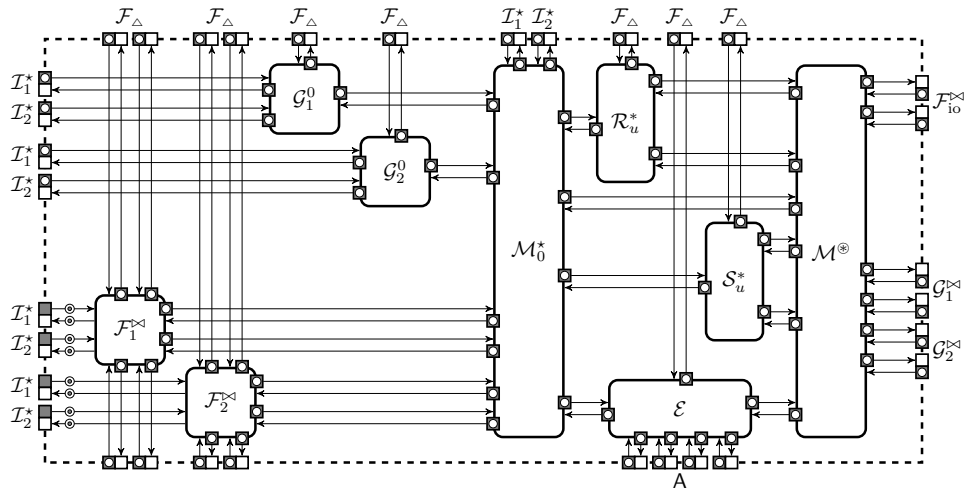


Figure 43: Protocol execution with the machines $\mathcal{G}_q^0$ added for the meaningful local functionalities. An extended look at $\mathfrak{F}_3$.

tion between $\mathcal{M}^\star$ and $\mathcal{M}_0$ and introduce $\mathcal{G}_q^0$. All modified buffers are sender-clocked and always empty when the control goes to A or Π_{e}. Hence, the collections could be distinguished based on the states of $\mathcal{M}_0$ vs $\mathcal{M}_0^\star$ and $\mathcal{M}^\star$ vs $\mathcal{M}^{\circledast}$. The modifications do not change the timing or updates of $\mathfrak{gs}$ in $\mathcal{M}^\star$ or $\mathcal{M}^{\circledast}$. Hence, it is important to only consider $\mathcal{M}_0$ vs $\mathcal{M}_0^\star$ in the places where the outputs of local computations are written. These can be read by $\mathcal{F}_p^{\bowtie}$, $\mathcal{I}_i^\star$ or A.

Overall, it is important to consider when $\mathcal{G}_q^{\bowtie}$ updates $\mathfrak{gs}[t,\delta,\ell]$ as, otherwise, both collections behave identically. In $\mathfrak{F}_2$, the value is updated in $\mathcal{M}_0$ when $\mathcal{R}_u^*$ can reconstruct the value from $\mathcal{M}^\star$. In $\mathfrak{F}_3$, the value may be written earlier if the input values are already in $\mathcal{M}_0^\star$. This early appearance in $\mathfrak{s}_0$ does not change the behaviour of canonical $\mathcal{F}_p^{\bowtie}$ that is used by a canonical program. $\mathcal{F}_p^{\bowtie}$ only fetches a value $\mathfrak{s}_0[t,\delta,\ell]$ if it has collected inputs from all interpreters needed for the reconstruction of this value. In a well-formed program(Definition 38), the interpreter does not send the command to $\mathcal{F}_p^{\bowtie}$ before it has computed the necessary inputs$\mathfrak{s}_0[t,\delta,\ell]$. More specifically, by definition, a well-formed program does not send out this memory location(t,δ,ℓ) before the input has been assigned. Hence, all interpreters must have finished the execution of the local functionality by the time the value is read and, by definition of $\mathcal{G}_q^0$, the value of the local operation output is then in $\mathfrak{s}_0$.

For the adversary, it cannot read the values in $\mathcal{M}_0^\star$ unless it has broken the hiding property for a given storage domain. In such a case, the early appearance of a local functionality output can be hidden from the adversary by simulating the appearance based on the $\mathfrak{gs}$. Note that the adversary then would also have access to sufficient shares in $\mathfrak{gs}$ to compute the value anyway.

The interpreter $\mathcal{I}_i^\star$ running a well-formed program also cannot read this mem-

ory location before it is assigned. Therefore, it does not try to read it before it knows that the local operation has been finished. Since the interpreter can only access values in local or public storage domains, then these values are always accessible once it has finished its own local computations. □

The previous lemma could be extended to some meaningful cases of non-deterministic local functionalities. However, the exact details depend on the issues discussed in Section 3.5 with regard to the meaning of non-deterministic local functionalities. However, commonly, the meaningful local functionalities are deterministic and the given result is not overly limiting.

5.5.4. Isolating Protocol Outputs

By definition, $\mathcal{F}_{\text{io}}^{\bowtie}$ interacts with $\mathcal{M}^{\circledast}$ to get the outputs of the protocol independently of the storage domain that is used. Some of these values could be in local or public domains, others generated by local functionalities or generated by $\mathcal{S}_u^*$ if they were computed by ideal functionalities. This section modifies the collection so that all outputs are always generated by $\mathcal{S}_\delta$ for storage domain δ. Note that $\mathcal{S}_\delta$ may be executing a deterministic algorithm if the storage domain is public or local. The goal of this modification is to gradually remove the need for $\mathcal{M}^{\circledast}$.

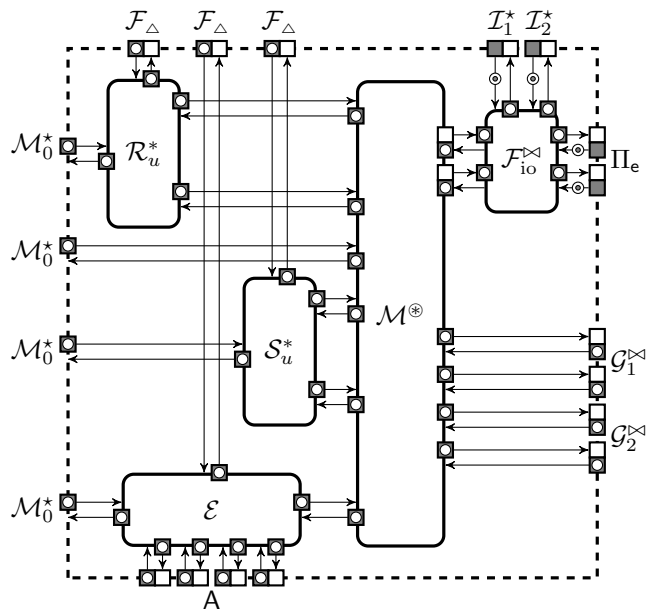


Figure 44: Memory model before output isolation, $\mathfrak{F}_3$.

The initial collection is shown in Figure 44. $\mathcal{F}_{\text{io}}^{\bowtie}$ is replaced by $\mathcal{R}^+$ and $\mathcal{S}^+$ to manage protocol inputs and outputs, respectively. Machines $\mathcal{I}_i^{\star}$, $\mathcal{M}^{\circledast}$ and $\mathcal{M}_0^{\star}$ are replaced by analogous $\mathcal{I}_i^+$, $\mathcal{M}^+$ and $\mathcal{M}_0^+$ to reflect the changes required to replace $\mathcal{F}_{\text{io}}^{\bowtie}$. The resulting collection $\mathfrak{F}_4$ is shown in Figure 45. The focus is on the output isolation. $\mathcal{R}^+$ behaves quite like the input processing so far. If $\mathcal{R}^+$ gets

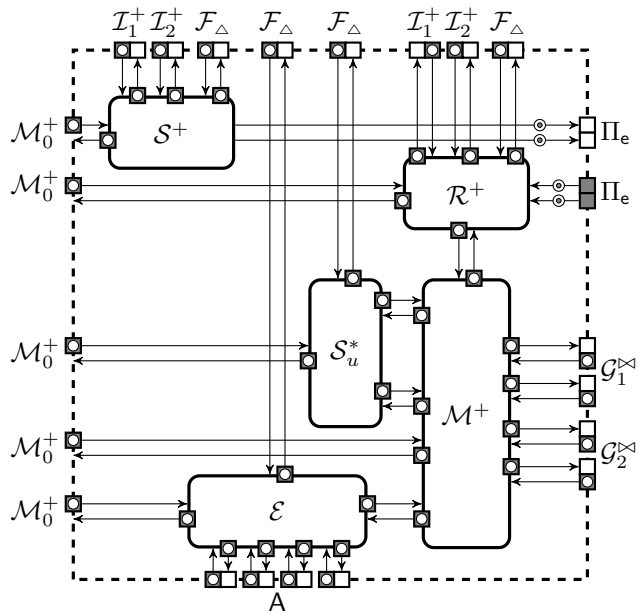


Figure 45: Memory model and behaviour of $\mathcal{F}_{\text{io}}^{\bowtie}$, $\mathcal{S}_u^*$, $\mathcal{R}_u^*$, $\mathcal{R}^+$ and $\mathcal{S}^+$ after memory isolation, $\mathfrak{F}_4$.

an input share, then it writes it to $\mathcal{M}^+$ and forwards the notification to $\mathcal{I}_i^+$. If $\mathcal{R}^+$ has enough shares for one value to reliably reconstruct[1] it, then it runs $\mathcal{R}_\delta$ and writes the value to $\mathcal{M}_0^+$. Hence, $\mathcal{R}^+$ replaces the input behaviour of $\mathcal{F}_{\text{io}}$ and the reconstruction previously done by $\mathcal{R}_u^*$. $\mathcal{R}^+$ writes the initial inputs to designated memory regions and responses to DMACALL calls to the designated response addresses. The new interpreter $\mathcal{I}_i^+$ must send the DMACALL to both $\mathcal{R}^+$ and $\mathcal{S}^+$, instead of sending one call to $\mathcal{F}_{\text{io}}^{\bowtie}$ as $\mathcal{I}_i^\star$ did. It always first sends the command to $\mathcal{S}^+$. $\mathcal{S}^+$ always shares the necessary value that it fetches from $\mathcal{M}_0^+$, writes it to buffers for Π_e and gives control back to $\mathcal{I}_i^+$ that called it. The machine $\mathcal{S}^+$ keeps a cache of the already computed shares. If $\mathcal{I}_i^+$ sends a location that has not yet been shared, then it computes fresh shares for this value. If the value already has shares, then it writes the previously generated share of party $\mathcal{P}_i$ to Π_e. The $\mathcal{R}^+$ needs to DMACALL call as it specifies the location and storage domain of the expected reply from Π_e. $\mathcal{R}^+$ stores this information and gives control back to $\mathcal{I}_i^+$. Note that for many protocols, this is irrelevant as the inputs expected from Π_e are fixed and so their memory locations can also be predetermined and $\mathcal{I}_i^+$ can simply send SEND messages to Π_e.

Note that $\mathcal{S}^+$ can only give an output if the value that needs to be shared is in $\mathcal{M}_0^+$. This is always the case if it is computed by some ideal functionality as the value is in $\mathcal{M}_0^+$ before any $\mathcal{I}_i^+$ can use it. The case is similar if the value is

[1]Often, it requires all shares, but for some sharing schemes, less than everything might be sufficient to know what the value is or to know that the reconstruction has to fail.

output by local functionalities and the inputs of the local computations came from ideal functionalities. However, this means that we cannot consider protocols that consist completely of local computations. If we have a protocol with only local operations, the computation in some $\mathcal{I}_i^+$ may finish before some other $\mathcal{I}_j^+$ even gets its inputs. Hence, it is not possible to carry out the computations in $\mathcal{M}_0^+$ to derive the value, as $\mathcal{R}^+$ has not even produced the input values yet. This is not overly limiting, as protocols that consist purely of local operations are local themselves. Therefore, they should be considered as local functionalities $\mathcal{G}_i$ and it is not necessary to try to prove that they are as secure as some ideal functionality.

On the other hand, similar issues to those with only local functionalities issue can be caused if the protocol uses ideal functionalities as well but, for example, the last operation that produces the output is a local operation using the result from the protocol so far and some input that has not been used yet. However, often it is more reasonable to assume that the interpreter only starts to execute its code if all inputs are available. Then the issue is resolved, as the value used in local operations is also in memory, as other parties have also started to execute the ideal functionalities and, therefore, also have all inputs. In more general protocols, output isolation is only achievable for protocols where no output is produced before all parties have received their inputs needed to compute this output. This restriction could be partially lifted by extending $\mathcal{S}^+$ so that it uses a share simulator for the cases where the value is not known yet. In this case, it needs to simulate shares until the value becomes known and then it needs to adjust the shares of the rest of the parties so that the shares can be reconstructed to the specified value. Note that for an adaptive adversary, this is what the share simulator $\mathcal{S}_\delta^{\mathsf{sim}}$ does in the definition of hiding storage domain in Definition 7. Note that, in many reasonable cases, it is not necessary to extend $\mathcal{S}^+$ to consider cases where the value it needs to share is not yet in $\mathcal{M}_0^+$.

Definition 45. A protocol environment pair $\mathsf{Env}\langle\Pi\rangle$ is in an output-isolated configuration for a class of adversaries $\mathbb{A}$ and resource consumption constraints if there is a construction ϕ_{oi}, such that $\mathsf{Env}\langle\mathfrak{F}_3, \mathsf{A}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_4, \phi_{\mathsf{oi}}(\mathsf{A})\rangle$ for $\mathsf{A} \in \mathbb{A}$, and the construction ϕ_{oi} satisfies resource constraints. Protocol Π is described as the protocol transformations represented by collections $\mathfrak{F}_3$ or $\mathfrak{F}_4$, respectively. A protocol Π is output-isolated if this property holds for any environment Env and is output-isolated for a class of environments if the property holds for any environment from that class.

The machine $\mathcal{S}^+$ in $\mathfrak{F}_4$ performs an explicit resharing of the protocol outputs. A particular protocol for resharing, as well as the desired ideal functionality, were discussed in Section 4.8.1. Adding this step to the end of the protocol can cause easy distinction between the collections $\mathfrak{F}_4$ and $\mathfrak{F}_3$ as the shares in $\mathcal{M}^{\circledast}$ do not match with the shares received by Π_{e}. If all outputs are returned by a deterministic $\mathcal{S}_\delta$, then this is not an issue. If the outputs of the protocol are computed by the ideal functionality, then the outputs given by the two collections are guaranteed

to have the same distribution as long as the value in $\mathcal{M}_0^\star$ and $\mathcal{M}_0^+$ is the same. However, if $\mathcal{F}_p^{\bowtie}$ requires the adversary to read the output shares from $\mathcal{M}^+$ or $\mathcal{M}^{\circledast}$ or the adversary just reads and modifies the outputs (which is allowed, as outputs are a DMACALL or SEND to Π_{e}), then it could easily distinguish the collections if it has access to the shares in Π_{e}. Further, there is no time for the adversary to modify the output in $\mathfrak{F}_4$ as the output is sender-clocked by $\mathcal{I}_i^+$. However, for a coherent adversary, it is sufficient to assume that it can do the same modification inside Π_{e}.

Despite these differences, any protocol description can be modified so that it becomes output-isolated by adding the explicit reshare functionality to the end of the protocol. At least as long as the protection domain contains a functionality $\mathcal{F}_{\mathsf{reshare}}$. Explicit resharing can definitely be avoided if the outputs are generated by some ideal functionality $\mathcal{F}_p^{\bowtie}$ that does not reveal anything about the shares to the adversary. In this case, the outputs of the two collections are generated either by $\mathcal{S}_u^*$ or $\mathcal{S}^+$. The following lemma captures these simple cases, which also, in practice, cover the most common protocols.

Lemma 14. *A canonical well-formed protocol specification is output-isolated for coherent adversaries, if all outputs are computed by some canonical ideal functionalities with standard corruption mode that do not show corrupted parties shares to the adversary, output shares are not used further in computations, and they are immediately returned as outputs.*

Proof. The protocol inputs are processed identically by $\mathcal{F}_{\text{io}}^{\bowtie}$ and $\mathcal{R}^+$ in $\mathfrak{F}_3$ and $\mathfrak{F}_4$ respectively. Therefore, the proof must focus on the changes in the outputs given by $\mathcal{F}_{\text{io}}$ or $\mathcal{S}^+$. In $\mathfrak{F}_3$, output generation states when $\mathcal{F}_p^{\bowtie}$ writes the output to $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathcal{M}_0^\star$. Then at some point, the adversary clocks the output of $\mathcal{F}_p^{\bowtie}$ to some $\mathcal{I}_i^\star$ that writes the DMACALL or SEND to $\mathcal{F}_{\text{io}}^{\bowtie}$. When such an input is clocked to $\mathcal{F}_{\text{io}}^{\bowtie}$, then $\mathcal{F}_{\text{io}}^{\bowtie}$ fetches the right share from $\mathfrak{gs}[t,\delta,\ell]$ in $\mathcal{M}^{\circledast}$ and writes it to Π_{e}. The values in $\mathfrak{gs}[t,\delta,\ell]$ are generated by $\mathcal{S}_u^*$ using the value in $\mathfrak{s}_0[t,\delta,\ell]$. The adversary A against $\mathfrak{F}_3$ can modify the output share before it clocks the input to $\mathcal{F}_{\text{io}}^{\bowtie}$.

In $\mathfrak{F}_4$, the output phase also starts with $\mathcal{F}_p^{\bowtie}$ writing the output to $\mathfrak{s}_0[t,\delta,\ell]$ in $\mathcal{M}_0^+$ and the adversary clocking the response of $\mathcal{F}_p^{\bowtie}$ to some $\mathcal{I}_i^+$. The interpreter $\mathcal{I}_i^+$ then sends DMACALL or SEND message to $\mathcal{S}^+$ and clocks it. $\mathcal{S}^+$ either uses the cached value if the location $\mathfrak{s}_0[t,\delta,\ell]$ has already been used by $\mathcal{S}^+$ or reads $\mathfrak{s}_0[t,\delta,\ell]$ from $\mathcal{M}^+$ and generates fresh shares using $\mathcal{S}_\delta$. $\mathcal{S}^+$ writes the output of the $\mathcal{P}_i$ to the buffer for Π_{e}.

The honest outputs reaching Π_{e} have the same distribution as they are, in both cases, generated from $\mathfrak{s}_0[t,\delta,\ell]$ using $\mathcal{S}_\delta$. By assumption, honest users do not access the output shares inside Π, hence adding $\mathcal{S}^+$ change does not affect honest parties. However, the adversary has different capabilities in the two collections and could notice the difference. However, we can restrict the class of adversaries such that the new class is as capable as the generic class of simplistic adversaries

but do not need to use the features separating $\mathfrak{F}_3$ and $\mathfrak{F}_4$. A coherent adversary A always corrupts the party $\mathcal{P}_i$ in the protocol together with the party $\mathcal{P}_i^*$ in Π_e and, if it wishes, it can read the outputs of the protocol in Π_e and rewrite the memory $\mathcal{M}^+$ with the outputs as they were given by $\mathcal{S}^+$. Since these memory locations are not used in the further execution of this protocol, then this change does not affect the protocol execution. However, this modification is only meaningful if it can be done in a timely manner before the adversary against $\mathfrak{F}_3$ wants to read the shares when communicating with $\mathfrak{F}_4$ instead of $\mathfrak{F}_3$. This can be achieved by adding an early clocking to simulate the view. By the assumption that the ideal functionalities do not give shares to the adversary, the adversary against $\mathfrak{F}_3$ does not read the shares before it considers them to be in the state of $\mathcal{I}_i^\star$. Hence, the adversary has clocked them to $\mathcal{I}_i^\star$ from the respective ideal functionality. In this case, $\mathcal{I}_i^+$ has already received these as well and interacted with $\mathcal{S}^+$ so that the output is written to the buffer for Π_e. Hence, the simulator can clock the input early to Π_e and read it from $\mathcal{P}_i^*$ but postpone any messages sent by $\mathcal{P}_i^*$ in Π_e until the adversary actually clocks this message to Π_e. □

There are some protocols where we need to add explicit resharing to achieve output isolation. For example, if the output is computed by a local functionality, where the distribution also differs from the output of $\mathcal{S}_\delta$. Such protocols could not be considered to be as secure as some canonical ideal functionality and may fall under the input privacy discussion in Chapter 4 that fit well with explicit resharing to achieve security. Such resharing also can serve as a copying functionality that allows giving fresh outputs and using the same value with different shares in the following steps of the protocol. However, there could be protocols that do not satisfy all the conditions of Lemma 14, but are still output-isolated. For example, if a protocol finished with a local resharing protocol that first uses some ideal functionality to generate shares of zero and then uses local functionality to add the shares of zero to the output. This is output-isolated but ends with a local computation and, therefore, does not satisfy Lemma 14.

If an output is computed by an ideal functionality, but not returned immediately, then a protocol can be transformed to satisfy the conditions of Lemma 14 if it is known during the ideal functionality execution that this value will be output. In such a case, the return statement can be moved right after the ideal functionality execution.

However, if the fact that something is an output becomes clear later, then there is no straightforward translation. Overall, output isolation is easily achievable in a context where either Π_e does not use the output of the protocol or the adversaries do not access the shares in the protocol. These results are shown in [96]. This thesis does not consider more explicit conditions that ensure output isolation in detail. Hence, there remain cases where output isolation must be explicitly proven. The following Theorem 19 ensures that requiring output isolation is a natural condition for any protocol that is as secure as a canonical ideal functionality.

Theorem 19. *Let $\mathcal{F}$ be a canonical ideal functionality that does not interact with the adversary, and let Π be a protocol that is as secure as $\mathcal{F}$ for a class of environments $\mathbb{E}$ and adversaries $\mathbb{A}$. Then Π is also output-isolated for the same classes of adversaries and environments.*

Proof. Let $\mathsf{A} \in \mathbb{A}$ be an adversary against the original protocol and environment pair $\mathsf{Env}\langle\Pi\rangle$ and let ϕ be the construction that proves the security of the protocol. Then $\phi(\mathsf{A})$ is the adversary against $\mathcal{F}$ and, by definition, it defines the protocol inputs for corrupted parties and receives protocol outputs for that party. According to the assumption, $\mathcal{F}$ does not directly communicate with $\phi(\mathsf{A})$ and, therefore, $\phi(\mathsf{A})$ does not interact with the protocol during execution. A may interact with the protocol, but ϕ is the construction that turns it into an adversary that achieves the same effect without interaction during the protocol execution.

Note that an adversary that does not interfere with Π can be directly used as an adversary against $\mathcal{F}$. A special adversary B against Π is such that it does not corrupt any parties and, by definition, $\phi(\mathsf{B})$ also does not corrupt any parties. Hence, honest execution of Π is equivalent to honest execution of $\mathcal{F}$. Next, consider the case when the ideal adversary $\phi(\mathsf{A})$ is interacting with Π instead of $\mathcal{F}$. In general, $\phi(\mathsf{A})$ can corrupt parties but, by definition, $\phi(\mathsf{A})$ can affect the protocol inputs but does not interfere with the protocol execution or read the internal state of Π. The adversary $\phi(\mathsf{A})$ only interacts with the parties before and after the protocol execution. Some more discussion relating to considering parties with $\mathcal{F}$ or direct communication between the environment and $\mathcal{F}$ is given in Section 3.4.3. In total, $\mathsf{Env}\langle\Pi, \mathsf{A}\rangle \equiv \mathsf{Env}\langle\Pi, \phi(\mathsf{A})\rangle$. The same construction ϕ can be used to show output isolation as it does not read the state of the corrupted parties.

By definition, if Π is as secure as canonical $\mathcal{F}$, then it has the same output distribution. The identical distribution ensures output isolation if the protocol's internal state is not observed. The only way to distinguish the collections in the output isolation definition is by noticing the disconnect between the outputs returned to Π_{e} and the outputs in the memory $\mathcal{M}^{+}$. The adversary $\phi(\mathsf{A})$ is not observing the protocol state. Hence, Π is output-isolated. □

The restriction on the interaction between the adversary and ideal functionalities also limits the adversary from aborting the protocol. Lemma 14 and Theorem 19 do not directly apply to functionalities with abort. For functionalities with outputs in some hiding protection domain, the possibility to abort is the only reasonable communication the adversary can have with the ideal functionality. Lemma 14 and Theorem 19 would remain unchanged for many cases, where the communication to blindly abort the protocol is allowed, as only hiding the internal state is critical for output isolation. The critical step in proving these results is that the abort-or-not decision must be made before the shares of the output are fixed in the protocol. The shares are fixed if they are later computed only using public values and local operations. However, it remains an open issue to fully formalise and prove this case. An example, where output isolation is difficult to prove because of

the adversarial interactions with the protocol, is the multiplication protocol using Beaver triples discussed in Section 5.9.

Overall it is not possible to prove output isolation for protocols where the output shares are computed deterministically from the inputs. Most such functionalities would be local functionalities and it would not be necessary to consider the property. However, a protocol that publishes some values and then does local computations would contain the canonical ideal functionality to publish the value but this functionality does not introduce randomness to the protocol outputs. This type of a functionality is neither ideal nor local and cannot be considered in this framework as one functionality. A protocol like this has to be considered as a composition.

Another interesting side effect of the output isolation definition is that all adversaries against $\mathfrak{F}_4$ can be turned into equivalent adversaries that do not read or modify the outputs in $\mathcal{M}^+$. Overall, it is not clear how the adversary might benefit from reading the output memory location, especially if the value is not used further in the protocol. Lemma 15 shows that, indeed, it is not restricting to assume that the adversary does read the outputs. The adversary is still allowed to read and modify these values in Π_e and can modify any copies of these values used as inputs to further computations in the protocol. The following sections assume such an adversary.

Lemma 15. *Any adversary against a well-formed protocol in output isolation configuration $\mathfrak{F}_4$ can be converted into an equivalent adversary that does not read or modify the output locations in $\mathcal{M}^+$ for hiding storage domains.*

Proof. By the definition of a well-formed program, the memory location used in the output is not further used in the protocol execution as each memory location can be used in one DMACALL or SEND instruction. Either the value is computed purely to be an output or the same value can be used in the protocol, but it is copied to another memory location. The value could also be either in a hiding storage domain or a value available to the adversary. Note that, by definition of $\mathfrak{F}_4$, the shares appearing in Π_e are generated by $\mathcal{S}^+$. Hence, the shares in $\mathcal{M}^+$ are not expected to be the same as the ones seen in Π_e.

If the same value is used further in the protocol, then the adversary has a copy of it in another memory location and, therefore, can read it from there instead. If it is computed and only used as an output, then the following cases hold.

If the value is in a public storage domain or a local domain of a corrupted party, then it can be read from $\mathcal{M}_0^+$. If the value is in a storage domain, where the adversary has broken the hiding property, then it can also read the underlying value from $\mathcal{M}_0^+$. However, the value adversary may also wish to read the share from $\mathcal{M}^+$. In this case, the adversary knows the underlying value and can either use the share simulator $\mathcal{S}_\delta^{\mathsf{sim}}$ from the hiding game (Definition 8) with the real value or can use the actual sharing functionality $\mathcal{S}_\delta$ if there are no private parameters. If the value is in a hiding storage domain where the hiding property still holds, then

the share can be simulated using $\mathcal{S}_\delta^{\mathsf{sim}}$ without specifying the value. □

Output Isolation and Simulatable Environments. Note that the definition of output isolation seems like a dual definition of the simulatable environment in Definition 31. The simulatable environment specifies that any shared value from Π would be reconstructed anyway. However, in fact, the machines $\mathcal{R}_{\mathsf{e}}^{\mathsf{Sim}}$ and $\mathcal{S}_{\mathsf{e}}^{\mathsf{Sim}}$ are specially defined to also propagate the shares of the corrupted parties between Π and Sim. Hence, the simulatability of Π_{e} does not straightforwardly imply that any protocol Π is running in a configuration where output isolation is ensured.

Output Isolation and Input Privacy. Chapter 4 did not directly consider output isolation. However, several concepts in that chapter are related to output isolation. Firstly, the idea of output predictability in Definition 27 also specifies that the distribution of the output shares of the functionality $\mathcal{F}_2$ is irrelevant in the composition. If canonical $\mathcal{F}_1$ or Sys_1 in output predictability definition is output-isolated and outputs are in a hiding protection domain, then its composition with a canonical ideal functionality $\mathcal{F}_2$ is always jointly predictable. The predictor can simply run $\mathcal{F}_1$ and $\mathcal{F}_2$ functionalities without generating the shares. However, if the outputs of $\mathcal{F}_1$ are public to the adversary and they are random, so that the predictor may not compute the same randomness, then the composition can be predictable only if $\mathcal{F}_2$ has hidden outputs or public outputs that do not reveal the output of $\mathcal{F}_1$. Note that predictability restricts the functionalities somewhat similarly to the definition of a canonical ideal functionality, as they are not allowed to reveal too much about their state to the adversary.

Ideas similar to output isolation are also captured by the composed ideal functionality in Definition 26 that completely hides the values that the composed ideal functionalities exchange with each other. These values are outputs of the independent functionality but not the outputs of the composed functionality. In this chapter, we require output isolation since we want to consider the protocol Π as corresponding to a specific ideal functionality $\mathcal{F}$ rather than considering the composed functionality for Π_{e} and Π. However, output isolation enables to create an effect where the internal state of Π does not affect Π_{e}.

Finally, the environment of the privacy configuration that contains two parts Env' and $\mathsf{Env}_\perp$ is such that a protocol is always output-private with respect to that class of environments that essentially ignore the outputs. All outputs of a protocol are given to $\mathsf{Env}_\perp$ and nothing about them is revealed to the adversary or to Env'. By definition, only Env' is used to distinguish protocols in a privacy configuration.

Overall, the whole idea of input privacy is that the outputs can be simulated using only the corrupted inputs. In this chapter, the output isolation is a step towards the possibility to simulate the shares rather than allowing the adversary access to all corrupted shares.

5.5.5. Limiting Adversarial Access to Share Memory

The collection $\mathfrak{F}_4$ (see the full setup in Figure 46) with the adversary that does not read the outputs from the memory $\mathcal{M}^+$ means that the outputs from $\mathcal{M}^+$ cannot affect Π_e. The actions of the adversary can still depend on $\mathfrak{gs}$ and it is used by $\mathcal{G}_q^{\bowtie}$. However, the values that reach Π_e are computed only by $\mathcal{F}_p^{\bowtie}$ and $\mathcal{G}_q^0$ using values in $\mathcal{M}_0^+$. Hence, in honest execution, $\mathcal{M}^+$ and its state $\mathfrak{gs}$ have become obsolete. This section removes the need for storing local computation outputs in $\mathcal{M}^+$ altogether by showing how the adversary can simulate these values.

The memory $\mathcal{M}^+$ contains inputs from Π_e (written by $\mathcal{R}^+$), outputs of $\mathcal{F}_p^{\bowtie}$ (written by $\mathcal{S}_u^*$), and outputs of $\mathcal{G}_q^{\bowtie}$. A coherent adversary can read the first from Π_e for all corrupted parties. Also, if an adversary can read the inputs to $\mathcal{G}_q^{\bowtie}$, then it can simulate the outputs of $\mathcal{G}_q^{\bowtie}$ and does not need to read these locations. Let $\mathbb{A}_o$ be the class of adversaries that only read outputs of $\mathcal{F}_p^{\bowtie}$ from $\mathcal{M}^+$.

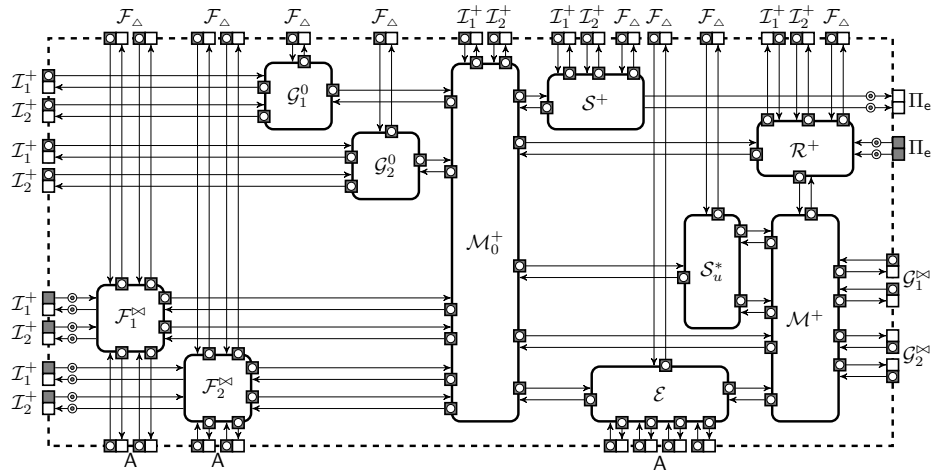


Figure 46: Full setup of protocol execution in collection $\mathfrak{F}_4$.

Lemma 16. *Let* A *be a coherent simplistic adversary against the output isolation configuration that does not read protocol outputs from* $\mathcal{M}^+$*. For any* A*, let* A_o *be a new adversary* $\mathsf{A}_o \in \mathbb{A}_o$ *that runs* A *with simulated* $\mathcal{I}_1^+, \ldots, \mathcal{I}_n^+$ *to answer the adversary's queries about* $\mathfrak{gs}$ *for local computations. Assume that we can rerun all computations for any corrupted party. Then, for any protocol* Π *with well-formed program,* $\mathsf{Env}\langle \mathfrak{F}_4, \mathsf{A} \rangle \equiv \mathsf{Env}\langle \mathfrak{F}_4, \mathsf{A}_o \rangle$ *for any coherent simplistic adversary* A *that does not read protocol outputs from* $\mathcal{M}^+$*.*

Proof. In more detail, A_o works as follows. A_o follows the clocking rules of A and also corrupts the same parties. If a party becomes corrupted, then A_o clones $\mathcal{I}_i^+$ and runs the clone in parallel with the real protocol to simulate the slice of $\mathcal{M}^+$ representing the shares of $\mathcal{P}_i$. Coherent A_o gets the inputs of $\mathcal{P}_i$ from $\mathcal{P}_i^*$ in Π_e. If the interpreter expects an output from $\mathcal{F}_p^{\bowtie}$, then A_o reads it from $\mathfrak{gs}$ and gives it.

Note that, by definition of a well-formed program, the protocol memory is not overwritten and, therefore, corrupting a party $\mathcal{P}_i$ gives enough information to read all DMACALL call input and outputs and redo all local computations until the current moment in the protocol. The local operations can be recomputed as all local operations are computed from the protocol inputs, outputs of $\mathcal{F}_p^{\bowtie}$ or other local computation outputs. Hence, all local computations can be recomputed from the protocol inputs or outputs of $\mathcal{F}_p^{\bowtie}$ according to their execution in $\mathcal{I}_i^+$. □

When $\mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}_o\rangle$ is executed, then the view of the adversary and Env depends on $\mathcal{M}_0^+$, and $\mathcal{M}^+$ is only relevant for locations that represent outputs of $\mathcal{F}_p^{\bowtie}$. All these locations are written by $\mathcal{S}_u^*$ based on the value in $\mathcal{M}_0^+$. Hence, it is safe to remove all other connections between $\mathcal{M}^+$ and $\mathcal{M}_0^+$. This gives us a new collection $\mathfrak{F}_5$ (Figure 47) with $\mathcal{M}^\times, \mathcal{R}^\times, \mathcal{I}^\times$ derived from $\mathcal{M}^+, \mathcal{R}^+, \mathcal{I}^+$. $\mathcal{R}^\times$ does not update $\mathcal{M}^\times$ and simply writes the reconstructed input to $\mathcal{M}_0^+$. Interpreter $\mathcal{I}_i^\times$ only activates $\mathcal{G}_q^0$ and $\mathcal{G}_q^{\bowtie}$ are removed. The adversary can still do modifications in $\mathcal{M}_0^+$ based on the limited control property (Definition 12). For that, $\mathcal{E}$ is rewired to not to read shares from $\mathcal{M}^\times$ but rather use the shares provided by A. In fact, A_o and $\mathcal{E}$ are combined to form a new adversary $\mathsf{A}_o^{\mathcal{E}} \in \mathbb{A}_o^{\mathcal{E}}$. $\mathbb{A}_o^{\mathcal{E}}$ is a class of adversaries that submit coherent modifications to $\mathcal{M}_0^+$ and $\mathcal{M}^\times$.

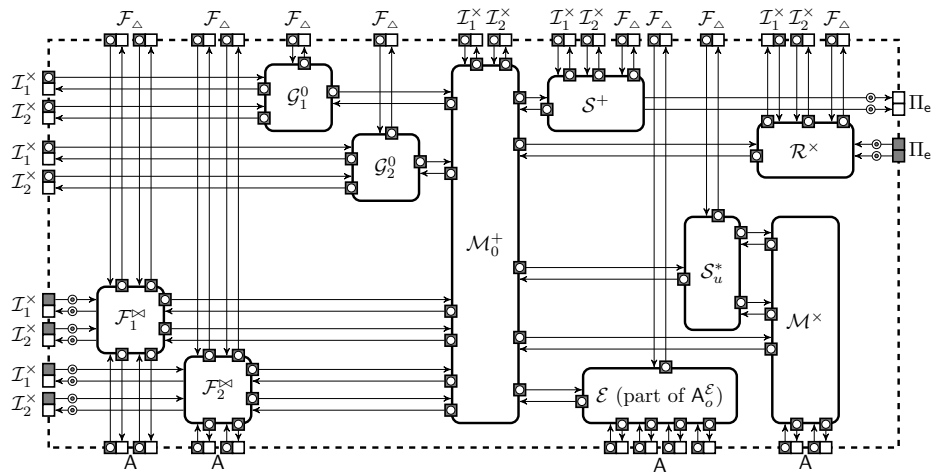


Figure 47: Full setup of protocol execution in collection $\mathfrak{F}_5$.

Lemma 17. *For any* $\mathsf{A}_o \in \mathbb{A}_o$ *and a protocol* Π *using storage domains with limited control, we have* $\mathsf{Env}\langle\mathfrak{F}_4, \mathsf{A}_o\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle$.

Proof. The transformation from $\mathfrak{F}_4$ to $\mathfrak{F}_5$ mainly removes $\mathcal{G}_p^{\bowtie}$ and disconnects $\mathcal{M}^\times$ from $\mathcal{R}^\times$. Recall that $\mathbb{A}_o$ is the class of adversaries that only read the outputs of $\mathcal{F}_p^{\bowtie}$ from $\mathcal{M}^\times$ and do not read protocol outputs at all. Outputs of $\mathcal{F}_p^{\bowtie}$ are still written to $\mathcal{M}^\times$ the same way in $\mathfrak{F}_5$ as they were written to $\mathcal{M}^+$ in $\mathfrak{F}_4$. The removed buffers and machines were all sender-clocked. Hence, removing them does not change the capabilities of the adversary A_o. The removed machines only

updated memory regions that A_o does not read from $\mathcal{M}^+$. Note that the memory locations where $\mathcal{F}_p^{\bowtie}$ writes its outputs are specified by $\mathcal{I}_i^{\times}$ in the DMACALL. Hence, independently of the removed local operations, the outputs of $\mathcal{F}_p^{\bowtie}$ are in the same locations in the two collections. The extractor $\mathcal{E}$ is now connected to the simulated memory in A_o and, as argued before in the proof of Lemma 16, this memory has the same values for all inputs and local computation outputs as were in $\mathcal{M}^+$. $\mathcal{E}$ works the same way in the two collections. □

5.5.6. Complete Memory Isolation and Semi-Abstract Adversaries

The adversary $\mathsf{A}_o^{\mathcal{E}}$ has very limited access to $\mathcal{M}^{\times}$ and the goal of this section is to define a semi-abstract adversary that does not require this access at all. Since all values currently read from $\mathcal{M}^{\times}$ are generated by $\mathcal{S}_u^*$, then they could be replaced by values simulated by $\mathcal{S}_\delta^{\mathsf{sim}}$ from the hiding property definition (Definition 8). Let A_{sa} be the adversary that does not access $\mathcal{M}^{\times}$ and internally runs $\mathsf{A}_o^{\mathcal{E}}$ with the simulated shares in $\mathcal{M}^{\times}$. Let $\mathbb{A}_{sa}$ be the class of semi-abstract adversaries that do not interact with $\mathcal{M}^{\times}$. A more detailed description of this class of adversaries is given in Definition 46 later.

Lemma 18. *Consider a protection domain $\mathcal{F}_{\mathsf{pd}}$ that uses only hiding storage domains δ without private parameters. Then $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{sa}\rangle$, for any adversary $\mathsf{A}_o^{\mathcal{E}} \in \mathbb{A}_o^{\mathcal{E}}$ and A_{sa}.*

Proof. As there are no secret parameters that might be shared between storage domains, then it suffices to consider only the case with one storage domain δ. Otherwise, the proof can be done step-by-step to each time only start simulating values in a specific storage domain until all storage domain values are simulated instead of real values in $\mathcal{M}^{\times}$. This is possible as the storage domains are independent. Recall that A_{sa} is internally using $\mathsf{A}_o^{\mathcal{E}}$ but using the simulated values for all memory accesses that $\mathsf{A}_o^{\mathcal{E}}$ does to $\mathcal{M}^{\times}$.

Consider an adversary B against the hiding property of the storage domain δ (Definition 8). The following proof defines it so that the games in the definition are equivalent to $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle$ and $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{sa}\rangle$ respectively. First, B needs to internally simulate all computations with the storage domain in Π_{e} that is part of Env. As there are no private parameters, then this is easy as B can run $\mathcal{S}_\tau$ and $\mathcal{R}_\tau$ for any storage domain τ.

In $\mathfrak{F}_5$, all communication that the protocol execution has with $\mathcal{M}^{\times}$ goes through $\mathcal{S}_u^*$. Let this collection $\mathcal{I}_1^{\times}, \ldots, \mathcal{I}_n^{\times}$, $\mathcal{F}_1^{\bowtie}, \ldots, \mathcal{F}_k^{\bowtie}$, $\mathcal{G}_1^0, \ldots, \mathcal{G}_m^0$, $\mathcal{M}_0^+, \mathcal{S}^{\times}, \mathcal{R}^{\times}$ be $\mathcal{K}$. The adversary can clock some buffers inside $\mathcal{K}$ and may interact with $\mathcal{F}_p^{\bowtie}$. If B simulates Π_{e}, then it produces the inputs to the protocol Π and can continue with simulating all interactions in $\mathcal{K}$.

B interacts with $\mathsf{A}_o^{\mathcal{E}}$ and corrupts parties the same as $\mathsf{A}_o^{\mathcal{E}}$. For all ideal functionality outputs in Π, the adversary B interacts with the hiding game. As B knows all the input values inside the protocol Π, then it can always compute the output value for each ideal functionality. Hence, if in the protocol $\mathcal{F}_p^{\bowtie}$ writes some value

x to $\mathfrak{s}_0$ and the shares are generated to $\mathfrak{gs}[t,\delta,\ell]$, then B inputs x to the hiding game and sets $\mathfrak{s}_0[\kappa] = x$ (in $\mathcal{L}$ in the hiding game) and $\mathfrak{b}[\kappa] = 1$. In the first game, the value x is always simply shared but setting $\mathfrak{b}[\kappa] = 1$ means that if B is in the second game, then it is simulated using $\mathcal{S}_\delta^{\mathsf{sim}}$. When $\mathsf{A}_o^{\mathcal{E}}$ queries corrupted shares of $\mathfrak{gs}[t,\delta,\ell]$, the adversary B fetches corresponding shares from $\mathcal{L}^*$ in the hiding game.

Hence, if B is playing against the first hiding game, then the situation for $\mathsf{A}_o^{\mathcal{E}}$ is equivalent to that of $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_o^{\mathcal{E}}\rangle$ and all shares in $\mathcal{M}^\times$ are generated using correct values and $\mathcal{S}_\delta$. If B is in the second hiding game, then, by definition, B puts $\mathsf{A}_o^{\mathcal{E}}$ in the same situation as A_{sa} and hence, this is equivalent to $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{sa}\rangle$. Hence, either the storage domains are not hiding or the collections running with the two adversaries are equivalent. □

Lemma 18 can be generalised to the case with storage domains with private setup parameters assuming that the environment is simulatable inside the hiding games. Such simulatability of the environment, especially Π_{e}, is discussed in Section 5.2.2. The simulation is similar to that used in Corollary 8.

For the semi-abstract adversary, the memory $\mathcal{M}^\times$ is useless. Hence, it is possible to define the semi-abstract execution model $\mathfrak{F}_6$ (Figure 48) by removing $\mathcal{M}^\times$ and $\mathcal{S}_u^*$ from $\mathfrak{F}_5$. This also modifies $\mathcal{M}_0^+$ to $\mathcal{M}_0^\times$ as the connections are removed.

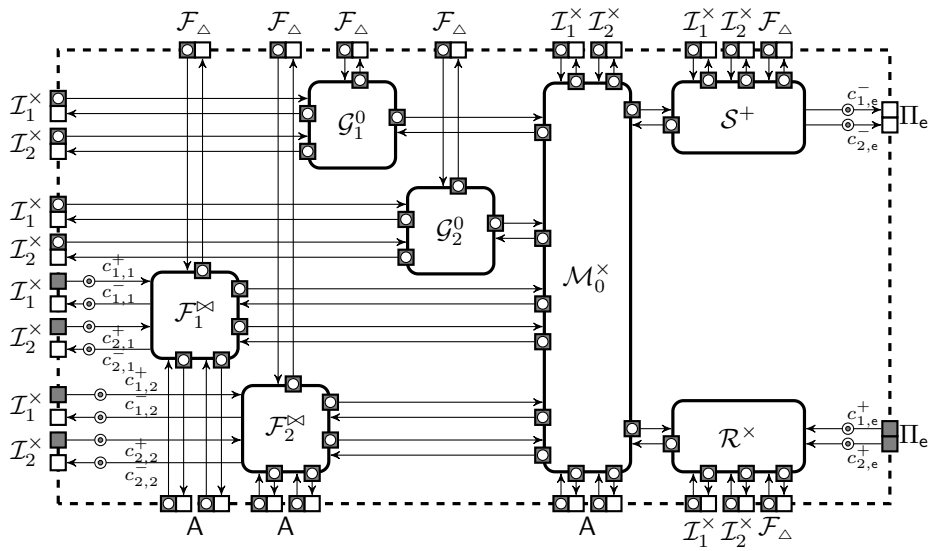


Figure 48: Semi-abstract execution model, $\mathfrak{F}_6$.

Lemma 19. *For any* $\mathsf{A}_{sa} \in \mathbb{A}_{sa}$, $\mathsf{Env}\langle\mathfrak{F}_5, \mathsf{A}_{sa}\rangle \equiv \mathsf{Env}\langle\mathfrak{F}_6, \mathsf{A}_{sa}\rangle$

Proof. By definition of $\mathbb{A}_{sa}$, this is trivial as A_{sa} does not communicate with the memory $\mathcal{M}^\times$ or $\mathcal{S}_u^*$ and the protocol execution does not require $\mathcal{M}^\times$. □

Note that, in the semi-abstract execution, the intuition that the adversary does coherent modifications to the memories is not relevant anymore, as there is only

$\mathcal{M}_0^\times$. From now on, it is more meaningful to consider the adversary effect in terms of the limited control property of the storage domain (Definition 12). Hence, the adversary A_{sa} can send $\mathcal{M}_0^\times$ modifications that are allowed by the limited control property of the storage domain where these values belong to. For example, if there is integrity protection, then A_{sa} can invalidate the value, but if there is no protection, then it can modify it. For a value x in $\mathcal{M}_0^\times$, the adversary can send a modification Δ. The memory verifies if the modification is allowed by the storage domain δ and if it is, it replaces x with $x \oplus_\delta \Delta$, where $\oplus_\delta$ is the modification function for storage domain δ.

In addition, it is important to keep in mind that $\mathbb{A}_{sa}$ is derived from the class of adversaries that do not read the protocol outputs in $\mathcal{M}^\times$. Previously, all modifications were done in $\mathcal{M}^\times$ and propagated to $\mathcal{M}_0^\times$. Hence, by construction, $\mathbb{A}_{sa}$ also do not modify output locations in $\mathcal{M}_0^\times$. As before, any modifications to the outputs can be done directly in Π_e.

However, there is no restriction to reading $\mathcal{M}_0^\times$ in locations where the hiding property has been invalidated, such as the local protection domain of the corrupted parties. Hence, the following is a valid description of the semi-abstract adversary interacting with $\mathfrak{F}_6$. All adversaries $\mathbb{A}_{sa}$ derived from simplistic adversaries satisfy the conditions of this definition.

Definition 46 (Semi-abstract adversary)**.** A simplistic adversary is semi-abstract if it fulfils the following conditions.

(a) The adversary clocks any outgoing buffer $c_{j,p}^+$ and incoming buffers $c_{j,p}^-$ to an honest party only when all incoming buffers $c_{i,p}^-$ and $c_{i,\mathsf{e}}^+$ to corrupted parties are empty.

(b) The adversary can modify the state of $\mathcal{M}_0^\times$ only in locations α of pending $\textsc{DmaCall}(t,p,\alpha,\beta)$ and $\textsc{Send}(t,p,\alpha)$ instructions to sub-protocols $\mathcal{F}_p^{\bowtie}$ (but not the ones sent to $\mathcal{S}^+$). Only modifications in accordance with the limited control property of the storage domain are allowed by $\mathcal{M}_0^\times$. These changes are done before the corresponding tuple is clocked to $\mathcal{F}_p^{\bowtie}$ and each value is modified at most once.

(c) The adversary can read the memory locations of $\mathcal{M}_0^\times$ for storage domains δ where it has broken the hiding property of the storage domain.

(d) The adversary is coherent.

Lemma 20. *For any simplistic adversary* A*, the construction of* A_{sa} *is a semi-abstract adversary.*

Proof. By definition, A_{sa} is constructed by simulating the memory $\mathcal{M}^\times$ with simulated shares or recomputed shares and satisfying the clocking rules of the underlying A. Hence, A_{sa} follows the simplistic clocking rules and also those of the semi-abstract adversary. Also, the simplistic adversary is coherent and the adversary A_{sa} corrupts the same parties. Thus, it is also coherent.

By construction, the modifications to $\mathcal{M}_0^\times$ result from the modifications computed by $\mathcal{E}$ and, therefore, there were equivalent modifications to shares in $\mathcal{M}^\times$. If $\mathcal{E}$ is a strong modification extractor, then it could also compute these. Hence, the rules about modifying $\mathcal{M}_0^\times$ are satisfied. The rules about reading $\mathcal{M}_0^\times$ stem from the initial definition of $\mathcal{M}_0$ and have not changed in the transformations to $\mathcal{M}_0^\times$. Hence, this hiding behaviour could either be considered the property of the value memory or the adversary. □

5.5.7. Equivalence of the Semi-Abstract Execution Model and Hybrid Execution Model

Most of Section 5.5 has transformed the shared memory model into the abstract memory model. The following corollary summarises these transformations and adds the previous results regarding the shared memory model. In total, it shows that the hybrid execution model with $\mathcal{F}_p$ can be transformed to the semi-abstract execution in $\mathfrak{F}_6$. Secondly, this section shows the transformation from the semi-abstract execution to the hybrid model. Hence, under the conditions of Corollary 10 and Corollary 9, the semi-abstract and hybrid execution models with respective adversaries are equivalent to each other.

Corollary 9. *Any adversary against the hybrid model can be transformed to an equivalent coherent semi-abstract adversary against the same protocol in $\mathfrak{F}_6$ for hiding and modification-aware storage domains with negligible probability of oblique modifications and for well-formed protocols that are robust against rushing, in a canonical form, use meaningful transparent local operations and use canonical ideal functionalities with tight scheduling.*

Proof. The proof results from a series of results shown in this chapter so far.

- It is sufficient to only consider a coherent adversary against the hybrid protocol (Lemma 1).
- If the protocol is robust against malformed inputs (Definition 34), then it is sufficient to consider only semi-simplistic adversary (Lemma 11 and Corollary 6).
- If all ideal functionalities have tight scheduling (Definition 32) and the protocol is secure against rushing (Definition 36), then it is sufficient to consider only lazy semi-simplistic adversaries (Lemma 12).
- For well-formed programs (Definition 38), it is sufficient to only consider simplistic adversaries (Theorem 13 and Corollary 7).
- Value and share memories can be separated for well-formed protocols (Definition 38) in a canonical form (Definition 42) with meaningful (Definition 15) and transparent ((Definition 44)) deterministic local functionalities (Theorem 15, 16).
- The interaction between value and share memories can be removed for hiding storage domains (Definition 8) and output-isolated protocols (Defini-

tion 45) if the probability of oblique modifications (Definition 43) is negligible (Theorem 17, Theorem 18, Lemma 16, Lemma 17, Lemma 18).
- The share memory can be completely removed for a semi-abstract adversary to achieve semi-abstract execution (Lemma 19, Lemma 20).

□

The previous corollary specifies that the hybrid execution model with a generic adversary can be turned into an equivalent semi-abstract execution with the semi-abstract adversary. The following lemma also shows that the semi-abstract execution can be transformed into the hybrid execution model under similar assumptions. Note that while Corollary 9 requires modification awareness, the following lemma requires its reverse notion called limited control.

Lemma 21. *Let all storage domains in a protection domain be hiding and have limited control and negligible probability of oblique modifications. Let protocol* Π *running in the protection domain be in a canonical form, have well-formed specifications and use meaningful local operations implementing a deterministic functionality. Then any coherent semi-abstract adversary against this protocol* Π *in* $\mathfrak{F}_6$ *running this protection domain can be transformed to an equivalent coherent simplistic adversary against the same protocol* Π *in* $\mathfrak{F}_0$.

Proof. First, Theorem 15 shows the equivalence of $\mathfrak{F}_0$ and $\mathfrak{F}_1$ for any well-formed protocol in a canonical form. Second, Theorem 17 shows the equivalence of $\mathfrak{F}_1$ and $\mathfrak{F}_2$ for storage domains with negligible chance of oblique modifications. Thirdly, Theorem 18 shows equivalence of $\mathfrak{F}_2$ and $\mathfrak{F}_3$ for meaningful deterministic local functionalities. Hence, it remains to consider transforming $\mathfrak{F}_6$ to $\mathfrak{F}_3$ and any semi-abstract adversary against $\mathfrak{F}_6$ to simplistic adversary (Definition 39). The setup in $\mathfrak{F}_3$ is shown in Figure 43. The adversary in $\mathfrak{F}_3$ can interact with the extractor $\mathcal{E}$ to do modifications to shares in $\mathcal{M}^{\circledast}$. The adversary clocking capabilities are the same as for the A_{sa}.

Consider the simulator Sim that connects the semi-abstract adversary to $\mathfrak{F}_3$. The adversary A_{sa} only communicates with $\mathcal{M}_0^{\times}$, $\mathcal{F}_p^{\bowtie}$ and clocks the leaky buffers. By assumption, the storage domains in $\mathcal{F}_{\mathsf{pd}}$ have limited control. Hence, all modifications of $\mathcal{M}_0^{\times}$ can be translated to modifications of shares in $\mathcal{M}^{\circledast}$ in $\mathfrak{F}_3$. The new adversary against $\mathfrak{F}_3$ combining Sim and A_{sa} only reads and modifies the memory locations from $\mathcal{M}^{\circledast}$, if A_{sa} modifies the same address in $\mathcal{M}_0^{\times}$. Internally, Sim runs the strong extractor for the storage domain defined by the limited control in Definition 12. If the storage domain has limited control, then, by definition, the strong extractor successfully computes the required modification and, hence, Sim can successfully compute the share modification for the shares in $\mathcal{M}^{\circledast}$ to match the value change A_{sa} writes to $\mathcal{M}_0^{\times}$. Hence, the values in $\mathcal{M}_0^{\circledast}$ and $\mathcal{M}_0^{\times}$ are the same in the two collections after one adversarial change. A similar argument can be extended to cover all adversarial modifications done by A_{sa}, which is the only difference in the adversarial behaviour in $\mathfrak{F}_3$ and 6. □

Corollary 10. *For all semi-abstract adversaries* A_{sa} *against the protocol* Π^{sa} *described in the semi-abstract model* $\mathfrak{F}_6$ *(in Figure 48), there exists an equivalent generic adversary* $\phi^*(\mathsf{A}_{sa})$ *against the hybrid protocol* Π *such that* $\mathsf{Env}\langle\Pi^{sa}, \mathsf{A}_{sa}\rangle \equiv \mathsf{Env}\langle\Pi, \phi^*(\mathsf{A}_{sa})\rangle$ *if the protocols are secure against rushing and malformed inputs, have a well-formed specification, are in a canonical form, use meaningful local operations for deterministic functionalities, use storage domains with limited access and negligible extraction failure.*

Proof. This corollary is the result of a series of previously proven results.

- The semi-abstract model can be transformed into the joint memory model (Lemma 21).
- The memory machines in the joint memory model can be split into individual machines for each interpreter so that $\mathcal{M}_i$ are memory aligned. These machines and local operations can be merged into the interpreters (Theorem 15).
- Any simplistic adversary against the joined memory model can be turned into the lazy semi-simplistic adversary against the hybrid model (Theorem 14).
- Lazy semi-simplistic adversary is a valid generic adversary (Theorem 11, Corollary 6).

□

5.6. Abstract Execution Model and Abstract Adversaries

The previous section defined the semi-abstract execution depicted in Figure 49 together with the environment where it is executed. This section further simplifies the execution setting to the abstract execution working with an abstract adversary and abstract environment. The abstract environment removes the protection domain component Π_{e} and considers the execution of the protocol in isolation from the rest of the computations that can happen in the protection domain. The abstraction removes the need for the secret data representation and purely uses values in memory with access rights defined by the security properties of the storage domains. In a way, the abstraction also modifies the protocol as the $\mathcal{S}^+$ and $\mathcal{R}^\times$ machines are removed.

So far, this section has defined all equivalences with respect to Env. As highlighted in the protection domain and its environments description in Section 3.6.2, the environment against the protocol Π is actually $\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$. Here, $\mathsf{Env}_{\mathsf{pd}}$ is the generic environment that gives plain inputs to parties inside Π_{e} and receives plain outputs. The protection domain is represented by Π_{pd} for the basic case and $\Pi_{\mathsf{e}}\langle\Pi\rangle$ for the case where the protocol Π is separated from the rest of the computations Π_{e}.

Formally, this section defines a *abs*-operator that acts on protocols and their components together with $\phi_{abs} : \mathbb{A}_{\mathsf{sa}} \to \mathbb{A}_{\mathsf{abs}}$ and its semi-inverse $\phi^*_{abs} : \mathbb{A}_{\mathsf{abs}} \to \mathbb{A}_{\mathsf{sa}}$

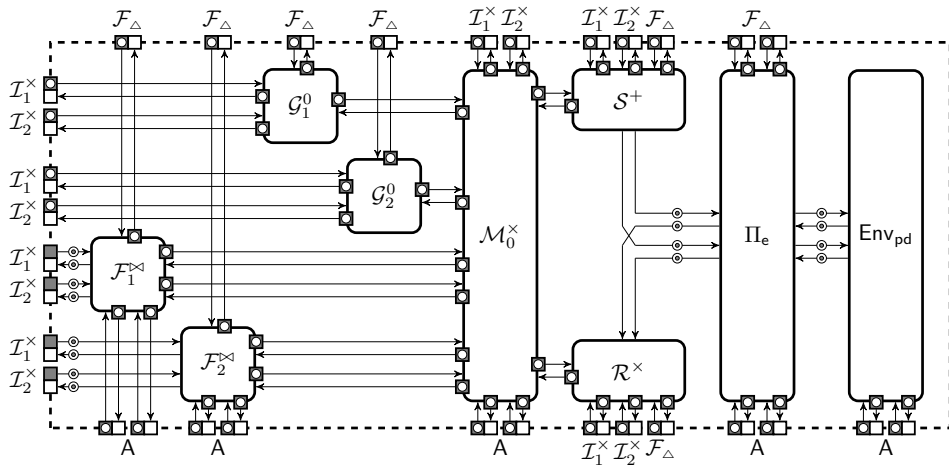


Figure 49: Semi-abstract protocol and the environment, $\mathfrak{F}_6$ with $\mathsf{Env}_\mathsf{pd}\langle\Pi_\mathsf{e}\rangle$.

and $\psi_{abs} : \mathbb{E} \to \mathbb{E}_\mathsf{abs}$ and its semi-inverse $\psi^*_{abs} : \mathbb{E}_\mathsf{abs} \to \mathbb{E}$ that achieve

$$\begin{aligned}\forall\Pi \in \mathbb{P}_\mathsf{sa} : \forall\mathsf{Env} \in \mathbb{E}_\Pi : \ \forall\mathsf{A}_{sa} \in \mathbb{A}_\mathsf{sa} : \\ \mathsf{Env}\langle\Pi, \mathsf{A}_{sa}\rangle \equiv \psi_{abs}(\mathsf{Env})\langle\Pi^{abs}, \phi_{abs}(\mathsf{A}_{sa})\rangle\end{aligned} \tag{5.12}$$

$$\begin{aligned}\forall\Pi \in \mathbb{P}_\mathsf{sa} : \forall\mathsf{Env}_\mathsf{abs} \in \mathbb{E}_{\mathsf{abs},\Pi} : \ \forall\mathsf{A}_{abs} \in \mathbb{A}_\mathsf{abs} : \\ \mathsf{Env}_\mathsf{abs}\langle\Pi^{abs}, \mathsf{A}_{abs}\rangle \equiv \psi^*_{abs}(\mathsf{Env}_\mathsf{abs})\langle\Pi, \phi^*_{abs}(\mathsf{A}_{abs})\rangle\end{aligned} \tag{5.13}$$

$$\begin{aligned}\forall\Pi \in \mathbb{P}_\mathsf{sa} : \forall\mathsf{Env} \in \mathbb{E}_\Pi : \ \forall\mathsf{A}_{sa} \in \mathbb{A}_\mathsf{sa} : \\ \mathsf{Env}\langle\Pi, \mathsf{A}_{sa}\rangle \equiv \psi^*_{abs}(\psi_{abs}(\mathsf{Env}))\langle\Pi, \mathsf{A}_{sa}\rangle\end{aligned} \tag{5.14}$$

$$\begin{aligned}\forall\Pi \in \mathbb{P}_\mathsf{abs} : \forall\mathsf{Env}_\mathsf{abs} \in \mathbb{E}_{\mathsf{abs},\Pi} : \ \forall\mathsf{A}_{abs} \in \mathbb{A}_\mathsf{abs} : \\ \mathsf{Env}_\mathsf{abs}\langle\Pi^{abs}, \mathsf{A}_{abs}\rangle \equiv \psi_{abs}(\psi^*_{abs}(\mathsf{Env}_\mathsf{abs}))\langle\Pi^{abs}, \mathsf{A}_{abs}\rangle \ ,\end{aligned} \tag{5.15}$$

where $\mathbb{P}_\mathsf{sa}$ is the set of protocols in a canonical form and semi-abstract specification, $\mathbb{A}_\mathsf{sa}$ is the set of semi-abstract adversaries and $\mathbb{E}_\Pi$ is the set of compatible environments. Protocol Π^{abs} is running in the abstract execution model (Definition 48), $\mathbb{E}_{\mathsf{abs},\Pi}$ is the set of compatible abstract environments (Definition 47) and $\mathbb{A}_\mathsf{abs}$ is the set of abstract adversaries (Definition 49).

Lemma 22 proves the first two equivalences for a special case. The full required result holds for embeddable environments (Definition 50), which defines one of the equivalences. The rest are proven in Lemma 23 and Lemma 24. The summary of all the equivalences between the hybrid and abstract world is given in Theorem 20.

5.6.1. Abstract Execution Model

Before describing the abstract model, consider a restricted class of environments $\mathbb{E}_\mathsf{r}$ against the semi-abstract protocol. Environment $\mathsf{Env}_\mathsf{r} \in \mathbb{E}_\mathsf{r}$ consists of Π_e and Env_pd as in Figure 49. Environment Env_r is restricted in a sense that the inner

environment Π_{e} is trivial – parties just forward inputs and outputs between $\mathsf{Env}_{\mathsf{pd}}$ and Π^{sa} and convert them between storage domains and public values as needed. More than that, each instance of Π_{e} runs exactly one instance of Π^{sa}, and the only communication inside Π_{e} occurs as Π_{io} protocol for handling inputs and outputs. The ideal implementation $\mathcal{F}_{\mathrm{io}}$ of Π_{io} consists of $\mathcal{R}_\delta$ and $\mathcal{S}_\delta$ for all storage domains δ supported by Π_{e}. Hence, the restricted environment $\mathsf{Env}_{\mathsf{r}}$ is depicted in Figure 50 with the machines $\mathcal{S}$ and $\mathcal{R}$ representing the collections of $\mathcal{S}_\delta$ and $\mathcal{R}_\delta$ respectively. All inputs of Π^{sa} are given by $\mathsf{Env}_{\mathsf{pd}}$ and $\mathcal{S}$ is used to transform the public values to the suitable storage domain for Π^{sa}. For outputs, the machine $\mathcal{R}$ collects all outputs in the output storage domain and returns the public value to $\mathsf{Env}_{\mathsf{pd}}$.

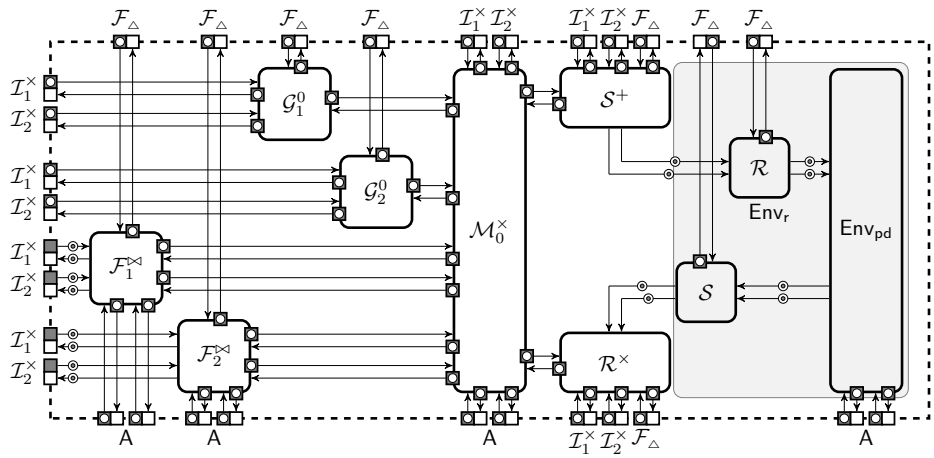


Figure 50: Semi-abstract protocol Π^{sa} with a restricted environment $\mathsf{Env}_{\mathsf{r}}$, $\mathfrak{F}_6$ with $\mathsf{Env}_{\mathsf{r}}$.

Considering the execution of the semi-abstract protocol Π^{sa} with the restricted environment $\mathsf{Env}_{\mathsf{r}}$ reveals that the input-output behaviour in Π^{sa} and this restricted Π_{e} kind of cancel each other out. The output given by $\mathcal{S}^+$ from Π^{sa} is reconstructed by $\mathcal{R}$ (a machine collecting all the $\mathcal{R}_\delta$). Similarly, all inputs arriving to Π^{sa} come from $\mathcal{S}$ and are reconstructed by $\mathcal{R}^{\times}$. So far, the details of the communication between $\mathsf{Env}_{\mathsf{pd}}$ and Π_{e} have not been considered. It is reasonable to expect that there are protocol instances used by Π_{e} to separate the protocols and the roles of the results given to $\mathsf{Env}_{\mathsf{pd}}$. It is also natural to require tight scheduling between $\mathsf{Env}_{\mathsf{pd}}$ and Π_{e} as well as between $\mathsf{Env}_{\mathsf{r}}$ and Π^{sa}.

For restricted adversaries that do not read the state of the corrupted parties in Π_{e}, it is easy to simplify the general environment in Figure 49 to the restricted environment in Figure 50. As the adversary does not use Π_{e}, then it is sufficient to assume that $\mathsf{Env}_{\mathsf{pd}}$ can run the operations computed by Π_{e} internally to derive the real values. For more general adversaries, it is then necessary to consider the cases when the operations in Π_{e} can be simulated for the adversary. The following defined this case as the abstract execution model as the Π_{e} is abstracted away

from the protection domain description. The restricted environment Env_r and Π^{sa} running with Env_r can be simplified to the abstract model with the abstract protocol description Π^{abs} (Figure 51). The equivalence is proven in Lemma 22.

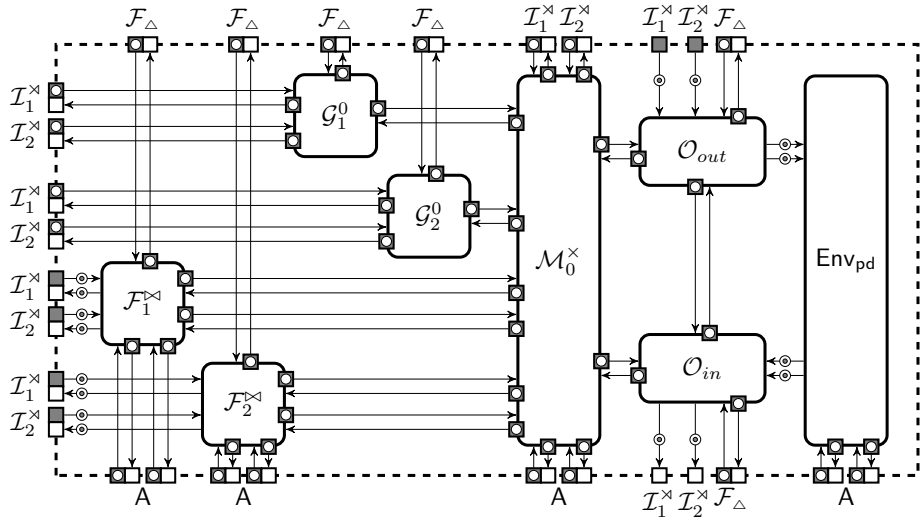


Figure 51: Abstract execution model for two-party protocols, $\mathfrak{F}_7$.

The abstract model explicitly cancels out the input-output machines in the restricted Π_e and Π^{sa}. Instead, it considers machines $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ that write inputs and read outputs from $\mathcal{M}_0^{\times}$. This also slightly changes the interpreter from $\mathcal{I}_i^{\times}$ to $\mathcal{I}_i^{\bowtie}$. The sender-clocked buffer pair between $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ allows instant sharing of their states to manage correct memory locations. Both machines also connect to $\mathcal{I}_i^{\bowtie}$, $\mathcal{M}_0^{\times}$, $\mathcal{F}_\triangle$ and Env_pd. Most buffer pairs for both of these machines are sender-clocked. The exceptions are the buffers with $\mathcal{I}_i^{\bowtie}$ and Env_pd that are leaky and clocked by the adversary. $\mathcal{I}_i^{\bowtie}$ behaves like $\mathcal{I}_i^{\times}$ with the difference in the clocking of the outputs. The sender-clocked buffer from $\mathcal{I}_i^{\times}$ to $\mathcal{S}^+$ is replaced by the leaky buffer to $\mathcal{O}_{out}$. Previously, $\mathcal{I}_i^{\times}$ expected to get the control back from $\mathcal{S}^+$. Instead of that, $\mathcal{I}_i^{\bowtie}$ can just do all the actions that $\mathcal{I}_i^{\times}$ would do after getting the signal back.

In Π^{sa} (running in $\mathfrak{F}_6$), $\mathcal{I}_i^{\times}$ sends anything to $\mathcal{S}^+$ only for DMACALL and SEND instructions that go to Π_e. In the restricted environment Env_r, there is only one instance of Π^{sa}. Hence, the instance information can be simplified. The interpreter $\mathcal{I}_i^{\bowtie}$ omits the instance of Π^{sa} from the messages and sends $(t_\mathsf{e}, \bullet, \ell)$ to $\mathcal{O}_{out}$ where t_e is the instance of Π_e calling it and ℓ is the message location. $\mathcal{I}_i^{\times}$ expects an input from $\mathcal{R}^{\times}$ to either launch a new instance or get a reply to DMACALL previously sent to Π_e. Both of these inputs are handled by $\mathcal{O}_{in}$ for $\mathcal{I}_i^{\bowtie}$. Upon receiving the input m from Env_o, the machine $\mathcal{O}_{in}$ writes m to location ℓ in $\mathcal{M}_0^{\times}$ and sends $\textsc{Init}(t_\mathsf{e}, \bullet, \delta, \ell)$ or $(t_\mathsf{e}, \bullet, \varepsilon)$ simultaneously to all buffers to all $\mathcal{I}_i^{\bowtie}$.

If a DMACALL call is sent to $\mathcal{O}_{out}$, then $\mathcal{O}_{out}$ gives the result location to $\mathcal{O}_{in}$ so that $\mathcal{O}_{in}$ knows which location to use with the respective input from Env_pd.

The control goes back to $\mathcal{O}_{out}$. After getting the control back, $\mathcal{O}_{out}$ simulates the behaviour of $\mathcal{S}^+$ and $\mathcal{R}$. If $\mathcal{R}$ has enough inputs from interpreters to reliably reconstruct the value for $\mathsf{Env}_{\mathsf{pd}}$, then $\mathcal{O}_{out}$ fetches the value from $\mathcal{M}_0^\times$ and writes it to $\mathsf{Env}_{\mathsf{pd}}$.

In a way, $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ simulate the execution of Π_{e} with instant clocking. The timing can be adjusted to fit that of the actual Π_{e} by clocking the buffers with $\mathcal{I}_i^{\bowtie}$ or $\mathsf{Env}_{\mathsf{pd}}$ at a suitable time. The adversary is in control of the timing of these buffers. Therefore it can still control the execution timing as it did in $\mathfrak{F}_6$. Let the collection with $\mathcal{O}_{in}$, $\mathcal{O}_{out}$ and $\mathcal{I}_i^{\bowtie}$ be $\mathfrak{F}_7$.

Lemma 22. *Let $\mathbb{A}_{sa}^r$ be the class of restricted semi-abstract adversaries against the collection $\mathfrak{F}_6$ running with the restricted environment $\mathsf{Env}_{\mathsf{r}}$. Let Π^{sa} be the protocol in $\mathfrak{F}_6$ and Π^{abs} be the same protocol in $\mathfrak{F}_7$. Let $\mathbb{A}_{\mathsf{abs}}$ be the class of adversaries against $\mathfrak{F}_7$. Then, there exist transformations ϕ_{abs} and ϕ_{abs}^* such that*

$$\forall \mathsf{Env}_{\mathsf{r}} \in \mathbb{E}_{\mathsf{r}} : \quad \forall \mathsf{A} \in \mathbb{A}_{sa}^r : \qquad \mathsf{Env}_{\mathsf{r}}\langle \Pi^{sa}, \mathsf{A}\rangle \equiv \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \phi_{abs}(\mathsf{A})\rangle$$
$$\forall \mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs}} : \quad \forall \mathsf{A}_{abs} \in \mathbb{A}_{\mathsf{abs}} : \ \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \mathsf{A}_{abs}\rangle \equiv \mathsf{Env}_{\mathsf{r}}\langle \Pi^{sa}, \phi_{abs}^*(\mathsf{A}_{abs})\rangle,$$

where $\mathsf{Env}_{\mathsf{r}}$ is the restricted environment as in Figure 50 and $\mathsf{Env}_{\mathsf{abs}}$ is the corresponding environment in $\mathfrak{F}_7$ containing the same $\mathsf{Env}_{\mathsf{pd}}$ and $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ instead of $\mathcal{S}$, $\mathcal{R}$, $\mathcal{S}^\times$ and $\mathcal{R}^\times$.

Proof. First, consider converting a restricted semi-abstract adversary A against $\mathfrak{F}_6$ to adversary B against $\mathfrak{F}_7$. B clocks buffers from $\mathcal{I}_i^{\bowtie}$ to $\mathcal{O}_{out}$ instead of buffers from $\mathcal{S}^+$ to Π_{e} and buffers form $\mathcal{O}_{in}$ to $\mathcal{I}_i^{\bowtie}$ instead of buffers from Π_{e} to $\mathcal{R}^+$. The timing of the buffers connected to $\mathsf{Env}_{\mathsf{pd}}$ remains the same for the two adversaries. Hence, when A clocks the input to $\mathcal{R}^\times$, then it has already clocked this input to $\mathcal{S}$, and, therefore, B has clocked into $\mathcal{O}_{in}$ and $\mathcal{O}_{in}$ already has written it to $\mathcal{M}_0^\times$ and to buffers to all parties. Hence, B can clock the input to respective $\mathcal{I}_i^{\bowtie}$. Note that values appear in $\mathcal{M}_0^\times$ earlier in $\mathfrak{F}_7$ than in $\mathfrak{F}_6$, but the protocol is not started earlier and, therefore, they are not used earlier than in $\mathfrak{F}_6$. Since the appearance of the values is controlled by how the inputs are clocked, then it is straightforward for the adversary to know when the value would be in the memory in $\mathfrak{F}_6$.

With outputs, A expects to clock the output shares to $\mathcal{R}$ from $\mathcal{S}^+$. $\mathcal{S}^+$ gets them from the interpreters with sender-clocked buffers. $\mathcal{R}$ would collect enough shares and then produce an output for $\mathsf{Env}_{\mathsf{pd}}$. The same effect is achieved by B in $\mathfrak{F}_7$ by clocking the $\mathcal{I}_i^{\bowtie}$ input to $\mathcal{O}_{out}$ at the same time as A clocks the input to $\mathcal{R}$. By definition, $\mathcal{O}_{out}$ needs the same set of inputs to reconstruct and write the value to $\mathsf{Env}_{\mathsf{pd}}$ as $\mathcal{R}$. Hence, $\mathcal{R}$ and $\mathcal{O}_{out}$ produce outputs for $\mathsf{Env}_{\mathsf{pd}}$ at the same time. This also implies that A and B can clock the outputs to $\mathsf{Env}_{\mathsf{pd}}$ with the same timing. note that $\mathcal{I}_i^{\bowtie}$ can write many outputs during one activation and so could $\mathcal{I}_i^\times$ as it got control back from $\mathcal{S}^+$.

Note that a semi-abstract A does not read the protocol inputs and outputs in $\mathcal{M}_0^\times$. In this case, it is easy to reverse the transformation to also get an equivalent A from any B. A generic adversary B against $\mathfrak{F}_7$ may still read the inputs in $\mathcal{M}_0^\times$

earlier than possible in $\mathfrak{F}_6$. If this is the case, then the inputs can be made to appear in memory early by adding some fast clocking to $\mathfrak{F}_6$ and postponing the clocking of the interpreter commands. For such clocking, when an input is clocked to $\mathcal{O}_{in}$ in $\mathfrak{F}_7$, it can be clocked first to $\mathcal{S}$ and then to $\mathcal{R}^{\times}$ in $\mathfrak{F}_6$. Later, the clocking of the inputs of $\mathcal{R}^{\times}$ must be simulated, as well as the state of the interpreters. However, without lessening of generality, we can assume that the adversary does not need to read the input early. Note that if the storage domain is hiding, then the adversaries cannot read the values in $\mathcal{M}_0^{\times}$ at all. Honest execution uses them to evaluate ideal functionalities only when all interpreters have given the command and the adversary has clocked the inputs to the ideal functionality. Hence, this timing would remain the same in the two collections. The local functionalities that are evaluated on inputs would be computed earlier together with the early clocking, but they can be simulated as part of the state of the interpreter to look like they are not yet evaluated. □

Corollary 11. *Let $\mathbb{A}_{sa}^{r}$ be the class of restricted semi-abstract adversaries against the collection $\mathfrak{F}_6$ that do not communicate with Π_{e}. Let Π^{sa} be the protocol in $\mathfrak{F}_6$ and Π^{abs} be the same protocol in $\mathfrak{F}_7$. Let $\mathbb{A}_{\mathsf{abs}}$ be the class of adversaries against $\mathfrak{F}_7$. Then, there exist transformations ϕ_{abs} and ϕ_{abs}^{*} such that*

$$\forall \mathsf{Env} \in \mathbb{E}_{\mathsf{r}} : \quad \forall \mathsf{A} \in \mathbb{A}_{sa}^{r} : \qquad \mathsf{Env}\langle \Pi^{sa}, \mathsf{A} \rangle \equiv \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \phi_{abs}(\mathsf{A}) \rangle$$

$$\forall \mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs}} : \quad \forall \mathsf{A}_{abs} \in \mathbb{A}_{\mathsf{abs}} : \qquad \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \mathsf{A}_{abs} \rangle \equiv \mathsf{Env}\langle \Pi^{sa}, \phi_{abs}^{*}(\mathsf{A}_{abs}) \rangle,$$

where Env *is the environment* $\mathsf{Env}_{\mathsf{pd}}\langle \Pi_{\mathsf{e}} \rangle$ *and* $\mathsf{Env}_{\mathsf{abs}}$ *is the corresponding abstract environment in $\mathfrak{F}_7$ containing the same* $\mathsf{Env}_{\mathsf{pd}}$ *and $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ instead of* Π_{e}.

Proof. For the adversaries that do not interact with Π_{e}, any environment $\mathsf{Env}_{\mathsf{pd}}\langle \Pi_{\mathsf{e}} \rangle$ is equivalent to the respective restricted environment $\mathsf{Env}_{\mathsf{r}}$. Hence, this is a direct result of Lemma 22. □

For most practical protocols, it is reasonable to also expect that some secure computation application is such that it only runs this protocol. For example, this is the case of comparison for the traditional millionaires' problem. Hence, it is natural that the set of all possible environments contains the environment with the restricted Π_{e}. This also means from Lemma 22, that for most protocols, the abstract environment as defined in $\mathfrak{F}_7$ is at least one environment among all that need to be considered as part of the security definition for all or a class of environments.

Definition 47. An abstract execution model for a protocol is defined as the collection $\mathfrak{F}_7$. The abstract protocol Π^{abs} is described in terms of $\mathcal{M}_0^{\times}$, $\mathcal{G}_q^0$, $\mathcal{F}_p^{\bowtie}$ and $\mathcal{I}_i^{\rtimes}$. The set of all abstract environments $\mathsf{Env}_{\mathsf{abs}}$ containing $\mathsf{Env}_{\mathsf{pd}}^{abs} \in \mathbb{E}_{\mathsf{pd}}$, $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$ with the respective buffers is called $\mathbb{E}_{\mathsf{abs}}$. The notion can also be considered for restricted environment classes, such as all polynomial time environments.

Note that the previous definition specifies that the environment component inside the abstract environment is from the class $\mathbb{E}_{\mathsf{pd}}$. The added notation $\mathsf{Env}_{\mathsf{pd}}^{abs}$

is used to distinguish it from the hybrid environment $\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$. The following specifies the transformation $\psi : \mathbb{E} \to \mathbb{E}_{\mathsf{abs}}$, where $\psi(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$ would introduce $\mathsf{Env}_{\mathsf{pd}}^{abs}$ that is different from $\mathsf{Env}_{\mathsf{pd}}$.

Definition 48 (Abstract execution model details)**.** At the beginning of the execution, the setup is distributed by $\mathcal{F}_{\triangle}$. $\mathcal{F}_{\triangle}$ sends the public setup information to $\mathcal{F}_p^{\bowtie}$, $\mathcal{G}_q^0$, $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$. Note that no private setup information is required. The adversary A can learn the setup information of corrupted parties, including the private parameters, directly from $\mathcal{F}_{\triangle}$. Note that private parameters are not used in the honest execution.

The protocol execution starts with the INIT message from $\mathsf{Env}_{\mathsf{pd}}$. The adversary A is in control of all leaky buffers in $\mathfrak{F}_7$. The INIT message is also sent through a leaky buffer. At some point, INIT is clocked to $\mathcal{O}_{in}$ that writes input values to $\mathcal{M}_0^{\times}$, gets control back and writes the INIT message to buffers for all $\mathcal{I}_i^{\rtimes}$. When A clocks the input to $\mathcal{I}_i^{\rtimes}$, the interpreter executes the next program instructions and writes the respective DMACALL or SEND instructions or writes a message to $\mathcal{G}_q^0$ and clocks it. If a local operation is used, then $\mathcal{G}_q^0$ performs the operation in $\mathcal{M}_0^{\times}$ if all inputs are present and output is not yet computed. The local functionality $\mathcal{G}_q^0$ always gives control back to $\mathcal{I}_i^{\rtimes}$ that called it. $\mathcal{I}_i^{\rtimes}$ continues execution. During each activation, $\mathcal{I}_i^{\rtimes}$ can have one input from either $\mathcal{F}_p^{\bowtie}$, $\mathcal{G}_q^0$ or $\mathcal{O}_{in}$. Each time it reads the input and executes the next relevant part in its program.

If the adversary clocks the input to $\mathcal{F}_p^{\bowtie}$ and $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ has collected enough input messages, then $\mathcal{F}_p^{\bowtie}$ fetches the value from $\mathcal{M}_0^{\times}$ and computes the output. The output is written to $\mathcal{M}_0^{\times}$, and the output message is written to buffers for all $\mathcal{I}_i^{\rtimes}$ in the output signature. If the ideal functionality has any interactions with the adversary before giving the output, then a message is sent to A before sending the output notifications. In such case, A will send a notification to proceed with giving the outputs. The machine $\mathcal{O}_{out}$ behaves similarly for the outputs of the protocol Π^{abs}. If $\mathcal{O}_{out}$ has enough inputs to reconstruct the value, then it means that enough $\mathcal{I}_i^{\rtimes}$ have sent the output command. In such case, $\mathcal{O}_{out}$ reads the output from $\mathcal{M}_0^{\times}$ and writes the value to the buffer for $\mathsf{Env}_{\mathsf{pd}}$.

The adversary against this execution can access the corrupted parties state in $\mathcal{M}_0^{+}$. It can also read all values where the hiding property of the respective storage domain is broken. It can also modify all values based on the limited control property. The adversary is still simplistic. Hence, only inputs to $\mathcal{F}_p^{\bowtie}$ are modified. The abstract adversary is properly defined in Definition 49.

Definition 49 (Abstract adversary)**.** An adversary is abstract if it fulfils the following conditions.

(a) The adversary clocks any outgoing buffer $c_{u,p}^{+}$ or $c_{u,\mathsf{Env}_{\mathsf{abs}}}^{-}$ and any incoming buffer $c_{j,p}^{-}$ or $c_{j,\mathsf{Env}_{\mathsf{abs}}}^{+}$ for honest $\mathcal{P}_j$ only when all incoming buffers $c_{i,p}^{-}$ or $c_{i,\mathsf{Env}_{\mathsf{abs}}}^{+}$ to corrupted parties $\mathcal{P}_i$ are empty.

(b) The adversary can modify the state of $\mathcal{M}_0^{\times}$ only in the locations α of pending DMACALL(t, p, α, β) and SEND(t, p, α) instructions to sub-protocols

$\mathcal{F}_p^{\bowtie}$. The modifications are allowed if A has some control over the value using the limited control property of the protection domain. These changes are done before the corresponding tuple is clocked to $\mathcal{F}_p^{\bowtie}$ and each value is modified at most once.

(c) The adversary can read the memory locations of $\mathcal{M}_0^{\times}$ for storage domains δ where it has broken the hiding property of the storage domain.

The abstract execution model abstracts away the part of the protection domain that is not the protocol at hand. In order to do so for the generic Π_{e}, it must be possible to instead do similar actions inside the new environment. The following definition specifies which generic environments can be turned into abstract environments. This property is needed to be able to apply the abstract model to prove the security of the protocol as detailed in Section 5.6.3.

Definition 50. An environment class $\mathbb{E}$ is embeddable into the abstract model for the class of adversaries $\mathbb{A}_{\mathsf{sa}}$, if there exist transformations $\phi_{abs} : \mathbb{A}_{\mathsf{sa}} \to \mathbb{A}_{\mathsf{abs}}$ and $\psi_{abs} : \mathbb{E} \to \mathbb{E}_{\mathsf{abs}}$, such that

$$\forall \mathsf{Env} \in \mathbb{E} : \quad \forall \mathsf{A} \in \mathbb{A}_{\mathsf{sa}} : \qquad \mathsf{Env}\langle \Pi^{sa}, \mathsf{A} \rangle \equiv \psi_{abs}(\mathsf{Env})\langle \Pi^{abs}, \phi_{abs}(\mathsf{A}) \rangle \ ,$$

for all protocols Π that can be transformed to their representation $\Pi^{abs} \in \mathbb{P}^{abs}$ in the abstract model and $\Pi^{sa} \in \mathbb{P}^{sa}$ in the semi-abstract model.

Note that we have denoted $\mathsf{Env}_{\mathsf{abs}} = \mathsf{Env}_{\mathsf{pd}}^{abs}\langle \mathcal{O}_{in}, \mathcal{O}_{out} \rangle$ and $\mathsf{Env} = \mathsf{Env}_{\mathsf{pd}}\langle \Pi_{\mathsf{e}} \rangle$. Especially, $\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle \Pi_{\mathsf{e}} \rangle) = \mathsf{Env}_{\mathsf{pd}}^{abs}\langle \mathcal{O}_{in}, \mathcal{O}_{out} \rangle$ where $\mathsf{Env}_{\mathsf{pd}} \in \mathbb{E}_{\mathsf{pd}}$ and $\mathsf{Env}_{\mathsf{pd}}^{abs} \in \mathbb{E}_{\mathsf{pd}}$ but $\mathsf{Env}_{\mathsf{pd}}$ and $\mathsf{Env}_{\mathsf{pd}}^{abs}$ are not the same environment.

Further, note that embeddability is closely related to the simulatability (Definition 31) of the environment that was required in the hiding and modification games in Section 5.5 and is discussed in Section 5.2.2. For embedding, all interactions between A and Π_{e} must be part of $\mathsf{Env}_{\mathsf{pd}}$ and adversary interactions. If all parameters by $\mathcal{F}_{\triangle}$ are public, then it is straightforward for $\mathsf{Env}_{\mathsf{pd}}$ to run anything that may happen in Π_{e}. In such a case, $\mathsf{Env}_{\mathsf{pd}}$ can simply learn the public parameters and honestly run all operations inside Π_{e} instead of interacting with the separate Π_{e} functionality. It can also reconstruct all values that $\mathsf{Env}_{\mathsf{pd}}$ would give to Π^{abs}. The adversary must be rewired to interact with the copy of Π_{e} inside $\mathsf{Env}_{\mathsf{pd}}$ instead of the external functionality.

With a private setup, the embeddability is similar to the simulatability with the simplification that the desired environment always gives out plain values for the abstract protocol. Hence, the emulated protocol does not need to be able to generate the right shares with the right private parameters as was done by $\mathcal{S}_{\mathsf{e}}^{\mathsf{Sim}}$ in the simulatability definition. Hence, the view modification between the simulation and the protocol run is simpler. The reverse is also true, as the abstract protocol gives out plain values and the embedded version can generate shares internally.

In more detail, if the adversary A_{sa} running with an environment $\mathsf{Env}_{\mathsf{pd}}$ wishes to communicate with Π_{e} then in the embedded version $\phi_{abs}(\mathsf{A})$ in fact communicates with $\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle \Pi_{\mathsf{e}} \rangle)$. Here $\phi_{abs} : \mathbb{A}_{\mathsf{sa}} \to \mathbb{A}_{\mathsf{abs}}$ (as used already in

Lemma 22) and $\psi_{abs} : \mathbb{E} \to \mathbb{E}_{\mathsf{abs}}$. The abstract environment $\psi_{abs}(\mathsf{Env})$ internally runs the part of Π_{e} without communicating with $\mathcal{F}_{\triangle}$, gives the desired result to $\phi_{abs}(\mathsf{A})$ that can internally deliver it to A. The semi-inverse transformation $\phi^*_{abs} : \mathbb{A}_{\mathsf{abs}} \to \mathbb{A}_{\mathsf{sa}}$ is defined analogously by instead wiring the adversary to the separate copy of Π_{e}.

There is also a semi-inverse $\psi^*_{abs} : \mathbb{E}_{\mathsf{abs}} \to \mathbb{E}$ that turns the abstract environment into the general outer environment and reintroduces Π_{e}. Note that for any abstract environment and adversary pair, it is very simple to create the respective restricted environment that can give inputs to Π^{sa}, essentially $\mathbb{E}_{\mathsf{r}} \cong \mathbb{E}_{\mathsf{abs}}$. The abstract environment itself can be used as the $\mathsf{Env}_{\mathsf{pd}}$ in the restricted environment definition. Such a restricted environment is a valid environment against any version of the protocol from the hybrid to the semi-abstract version. Hence, simply introducing $\mathcal{S}$ and $\mathcal{R}$ like in the restricted environment is a possible definition of ψ^*_{abs} for generic $\mathsf{Env}_{\mathsf{abs}}$. However, without lessening of generality, for $\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$ assume that the resulting abstract environment specifies the distinction between Π_{e} and $\mathsf{Env}_{\mathsf{pd}}$ internally. In such case, $\psi^*_{abs}(\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle))$ can use this distinction and separate Π_{e} again.

The following lemma specifies that the transformations to and from the abstract environment satisfy the equalities 5.15 and 5.14 as required.

Lemma 23. *For ψ_{abs} and ψ^*_{abs} described above,*

$$\forall \Pi^{abs} \in \mathbb{P}_{\mathsf{abs}} : \forall \mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs},\Pi} : \ \forall \mathsf{A}_{abs} \in \mathbb{A}_{\mathsf{abs}} : \\ \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \mathsf{A}_{abs}\rangle \equiv \psi_{abs}(\psi^*_{abs}(\mathsf{Env}_{\mathsf{abs}}))\langle \Pi^{abs}, \mathsf{A}_{abs}\rangle$$

and

$$\forall \Pi^{sa} \in \mathbb{P}_{\mathsf{sa}} : \forall \mathsf{Env} \in \mathbb{E}_{\Pi} : \ \forall \mathsf{A}_{sa} \in \mathbb{A}_{\mathsf{sa}} : \\ \mathsf{Env}\langle \Pi^{sa}, \mathsf{A}_{sa}\rangle \equiv \psi^*_{abs}(\psi_{abs}(\mathsf{Env}))\langle \Pi^{sa}, \mathsf{A}_{sa}\rangle \ .$$

Proof. By definition, $\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$ specifies the internal border between $\mathsf{Env}_{\mathsf{pd}}$ and Π_{e}. Hence, $\psi^*_{abs}(\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)) = \mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$ which satisfies the second equivalence.

First, $\psi^*_{abs}(\mathsf{Env}_{\mathsf{abs}})$ for general $\mathsf{Env}_{\mathsf{abs}} = \mathsf{Env}^{abs}_{\mathsf{pd}}\langle\mathcal{O}_{in}, \mathcal{O}_{out}\rangle$ gives the restricted environment with $\mathcal{S}$ and $\mathcal{R}$ and keeping $\mathsf{Env}^{abs}_{\mathsf{pd}}$ in the role of the $\mathsf{Env}_{\mathsf{pd}}$ inside $\mathsf{Env}_{\mathsf{r}}$. The transformation $\psi_{abs}(\psi^*_{abs}(\mathsf{Env}_{\mathsf{abs}})$ would give the same abstract environment $\mathsf{Env}_{\mathsf{abs}}$. For the case where $\mathsf{Env}_{\mathsf{abs}} = \psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$, the analysis of the previous equality gives $\psi^*_{abs}(\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)) = \mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle$ and, hence $\psi_{abs}(\psi^*_{abs}(\psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle))) = \psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$. Hence, the first equivalence is satisfied for both cases. □

Lemma 24. *For ψ^*_{abs} defined above, there exists $\phi^*_{abs} : \mathbb{A}_{\mathsf{abs}} \to \mathbb{A}_{\mathsf{sa}}$ such that*

$$\forall \Pi \in \mathbb{P}_{\mathsf{sa}} : \forall \mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs},\Pi} : \ \forall \mathsf{A}_{abs} \in \mathbb{A}_{\mathsf{abs}} : \\ \mathsf{Env}_{\mathsf{abs}}\langle \Pi^{abs}, \mathsf{A}_{abs}\rangle \equiv \psi^*_{abs}(\mathsf{Env}_{\mathsf{abs}})\langle \Pi, \phi^*_{abs}(\mathsf{A}_{abs})\rangle \ ,$$

where Π^{abs} is the abstraction of the semi-abstract protocol Π.

Proof. For the case where $\mathsf{Env}_{\mathsf{abs}} = \psi_{abs}(\mathsf{Env}_{\mathsf{pd}}\langle\Pi_{\mathsf{e}}\rangle)$, the $\phi^*_{abs}(\mathsf{A}_{abs})$ has to be such that again communicates with Π_{e} that is separated. Note that A_{abs} communicates with the emulated Π_{e}, hence, the only transformation in ϕ^*_{abs} is the rewiring from the emulated to real Π_{e}.

For the case where Π_{e} cannot be separated, $\mathsf{Env}_{\mathsf{abs}} = \mathsf{Env}^{abs}_{\mathsf{pd}}\langle\mathcal{O}_{in},\mathcal{O}_{out}\rangle$ and $\psi^*_{abs}(\mathsf{Env}_{\mathsf{abs}}) = \mathsf{Env}_{\mathsf{pd}}\langle\mathcal{S},\mathcal{R}\rangle$ as in the restricted environment. For such a case, Corollary 11 applies as, by definition, Π_{e} does not exist, and therefore the adversary does not communicate with it. □

5.6.2. Equivalence of the Two Ideal Functionality Descriptions

The following discussion in Section 5.6.3 about the meaning of security in the abstract model requires another look at the functionality $\mathcal{F}$ that is implemented by the protocol Π^{abs}. The goal is to show that a protocol is as secure as the ideal functionality. This subsection works in the hybrid model, where shares are passed between parties and the functionality.

Section 3.4.3 introduced two ways for considering an ideal functionality. One, where the functionality is used directly by the environment, and the other, where it is used by a party $\mathcal{P}_i$ that only executes this functionality. The latter is better suited for the considerations of this chapter since, if this functionality $\mathcal{F}$ is canonical, then this setup could be transformed to the abstract execution mode,l if the rest of the conditions are met. However, the following lemma specifies that, for coherent adversaries, the two approaches are always equivalent.

Lemma 25. *For coherent adversaries and canonical ideal functionalities, the collections $\mathfrak{D}$ and $\mathfrak{I}$ in Figure 12 are equivalent.*

Proof. Let A_1 be the adversary against $\mathfrak{D}$ where the ideal functionality $\mathcal{F}$ is used directly and let A_2 be the adversary against $\mathfrak{I}$. The collection $\mathfrak{I}$ is such that the ideal functionality $\mathcal{F}$ is used by parties $\mathcal{P}_i$ executing only the command to call $\mathcal{F}$ and return the result. In both collections, $\mathcal{F}$ gets exactly the same inputs, including tags, and the adversary controls when $\mathcal{F}$ receives its inputs. However, the adversary has two leaky buffers to control in $\mathfrak{I}$ for each party and only one in $\mathfrak{D}$.

Adapting A_1 to work against $\mathfrak{I}$ simply means that the new adversary $\mathsf{A}_1^{\mathfrak{I}}$ clocks the buffers between $\mathcal{P}_i$ and $\mathcal{F}$, as soon as there are messages and clocks the buffers between $\mathcal{P}_i$ and Env at the same time as A_1 clocks the buffer under its control.

The extra channel must be simulated when turning A_2 to work against $\mathfrak{D}$. The more difficult part is the fact that, in $\mathfrak{D}$, there is no $\mathcal{P}_i$ that could be corrupted by $\mathsf{A}_2^{\mathfrak{D}}$. For a coherent adversary A_2, corrupting a party $\mathcal{P}_i$ means that also $\mathcal{P}_i^*$ in Π_{e} that is part of the Env is corrupted. Hence, to construct $\mathsf{A}_2^{\mathfrak{D}}$ all modifications that A_2 does in $\mathcal{P}_i$ must be done in $\mathcal{P}_i^*$ for $\mathfrak{D}$. Hence, the clocking of inputs of Env to $\mathcal{P}_i$ is simulated, as are the interactions with $\mathcal{P}_i$. If the adversary A_2 modifies the value that $\mathcal{P}_i$ gives to $\mathcal{F}$, then $\mathsf{A}_2^{\mathfrak{D}}$ does this modification in Env, writes the new

message and clocks this to $\mathcal{F}$ when A_2 clocks the input to $\mathcal{F}$. For the outputs, $\mathsf{A}_2^{\mathfrak{D}}$ clocks the outputs to Env when A_2 clocks the outputs of $\mathcal{F}$ to $\mathcal{P}_2$. The $\mathsf{A}_2^{\mathfrak{D}}$ then stops $\mathcal{P}_i^*$ from proceeding and first simulates the interactions between A_2 and $\mathcal{P}_i$. If A_2 modifies the value sent to Env then $\mathsf{A}_2^{\mathfrak{D}}$ modifies this value in $\mathcal{P}_i^*$. It allows $\mathcal{P}_i^*$ to proceed with execution according to commands from A_2 once A_2 clocks the message from $\mathcal{P}_i$ to $\mathcal{P}_i^*$. □

5.6.3. Security in the Abstract Model

The abstract execution model is the final execution model considered in this thesis. It remains to argue that it is meaningful to consider this model. Firstly, it is necessary to consider what it means for a protocol to be as secure as another in the abstract model. Secondly, security in the abstract model is only reasonable to use if it is correlated with the security in the hybrid model. The first open issue is solved by the following definition, which is a variation of the basic security definition (Definition 2).

Definition 51 (Abstract security)**.** Let Π_1 and Π_2 be protocols and Π_1^{abs} and Π_2^{abs} their abstractions. Let $\mathbb{A}_1^*$ and $\mathbb{A}_2^*$ be the abstractions of classes of adversaries $\mathbb{A}_1$ and $\mathbb{A}_2$ against the original protocols. Then Π_1, is as secure as Π_2 in the abstract model if there exists a construction $\rho_* : \mathbb{A}_1^* \to \mathbb{A}_2^*$ such that $\mathsf{Env}_{\mathsf{abs}}\langle\Pi_1^{abs}, \mathsf{A}_1^*\rangle \equiv \mathsf{Env}_{\mathsf{abs}}\langle\Pi_2^{abs}, \rho_*(\mathsf{A}_1^*)\rangle$, for all $\mathsf{A} \in \mathbb{A}_1^*$ and $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs}}$ that are compatible with the protocols.

In the common case, the proof goal is to show that some protocol is as secure as an ideal functionality. As considered in Lemma 25, the ideal functionality can be executed by explicit parties. Such ideal functionality usage is a special case of general protocol execution considered for the protocol Π throughout this chapter. The parties apply a protocol Π_0 that only calls the ideal functionality $\mathcal{F}_0$ once and then returns the result to Π_{e}. Hence, the usage of an ideal functionality $\Pi_0\langle\mathcal{F}_0\rangle$ can be simplified from the hybrid model to the abstract execution model as long as the conditions for the transformation are satisfied. The result of the simplifications is the abstract protocol Π_0^{abs} where all interpreters call $\mathcal{F}_0^{\bowtie}$ that uses the values in $\mathcal{M}_0^{\times}$.

The definition of abstract security is the final link to complete the steps highlighted in Section 5.1.1. Figure 31a defined the relationships needed to achieve security. The following theorem lists all the requirements for the transformations between the hybrid and abstract model that are required to satisfy the equivalences needed for Theorem 9 to hold in this case.

Theorem 20 (Soundness of abstract security, Theorem 8 formally)**.** *Let $\mathcal{F}_0$ be an ideal functionality and let Π be a protocol constructed on top of a hybrid protection domain containing functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_k$ with canonical (Definition 14) $\mathcal{F}_p$ with tight scheduling (Definition 32), meaningful (Definition 15) and transparent (Definition 44) local functionalities. The storage domains in the protection domain are hiding (Definition 8), modification-aware (Definition 10) and have*

limited control (Definition 12). If the protocol Π *satisfies all abstraction assumptions:*

- *it is well-formed (Definition 38) and in a canonical form (Definition 42),*
- *it is robust against malformed inputs (Definition 34),*
- *it is secure against rushing (Definition 36) ,*
- *it is output-isolated (Definition 45),*

and the environment class $\mathbb{E}$ *is embeddable (Definition 50) into* $\mathbb{E}_{\mathsf{abs}}$ *for protocols* Π *and* $\mathcal{F}_0$ *and the class of adversaries* $\mathbb{A}$ *and* $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}$ *then security (*Π *is as secure as* $\mathcal{F}_0$*) in the abstract model (Definition 51) is necessary and sufficient for security in the hybrid model.*

Proof. IDEAL FUNCTIONALITY. First of all, note that $\Pi_0\langle\mathcal{F}_0\rangle$ is equivalent to $\mathcal{F}_0$ for the trivial protocol Π_0 that only calls $\mathcal{F}_0$, as shown in Lemma 25. The trivial protocol can be made so that it easily satisfies all the abstraction assumptions for the protocols. Thus, $\Pi \geq \mathcal{F}_0$ if and only if $\Pi \geq \Pi_0\langle\mathcal{F}_0\rangle$.

Note that the trivial protocol Π_0 running just $\mathcal{F}_0$ is output-isolated since it always returns the output of the ideal functionality right after it is computed (Lemma 14). It can also be easily written so that it is well-formed and in a canonical form. Especially, it is easy to achieve memory alignment as the only values are the ones that are inputs or outputs of $\mathcal{F}_0$. Since the interpreter does not do any local computations and can easily be written to limit the size of the inputs, then it is also robust against malformed inputs. Similarly, the protocol is secure against rushing since the protocols of all interpreters are symmetric and throughout the simplifications, we assume that Π_{e} is such that it does not help to rush the protocol it is using.

From the previous results, Corollary 9 and Corollary 10 specified that under the conditions of this theorem, the hybrid model and the semi-abstract execution are equivalent. For the adversaries that do not interact with Π_{e}, Corollary 11 specifies that the abstract execution is equivalent to the semi-abstract execution. Hence, for such adversaries, abstract security is sound for security in the hybrid model.

By definition of embeddability in Definition 50, any environment-adversary pair can be turned to an abstract environment and adversary that achieve the same effect against the abstract version of the protocol. Lemmas 23 and Lemma 24 specify that the abstract model and the semi-abstract model are equivalent. Hence, the protocol and the respective ideal functionality can both be transformed into their respective abstract versions, and it is possible to consider their abstract security. Hence, this theorem is a special case of Theorem 9 for the transformations $\phi_1 = \phi_2$ as outlined throughout this chapter and ψ as defined in this section.

The hybrid (or real) execution is defined by the general environments $\mathbb{E}$ and classes of adversaries $\mathbb{A}_1$ and $\mathbb{A}_2$ for the hybrid and ideal world, respectively. The work so far has shown the definitions of ϕ and ψ, whereas $\phi_1 = \phi_2$, as well as their semi-inverses. Note that as shown in Lemma 25, the ideal functionality can be considered as a very special protection domain. Hence, the transformations

to turn it to the abstract model are the same as done for the general protocol and the transformations ϕ_2 and ϕ^* are the same as the transformations done for the adversary throughout this chapter. The abstract environments are $\mathbb{E}^*$ in the context of Theorem 9 and abstract adversaries are in classes $\mathbb{A}_1^*$ and $\mathbb{A}_2^*$. If abstract security is proven, then this proof provides ρ^*. Hence, this implies that there exists ρ that proves security in the hybrid model. The transformations so far have shown that they satisfy a series of equivalences highlighted in Section 5.1.1. Hence, the fact that abstract security implies security in the hybrid model is a direct consequence of Theorem 9. □

Note that it is also intuitive that abstract security is at least a necessary property for the security of the protocol. For the necessity of the security in the abstract model, it is important to observe that $\mathbb{E}_{\mathsf{abs}}$ is commonly a valid class of environments against any protocol. $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}$ since a trivial Π_{e} that just forwards the inputs and outputs has to be a valid protocol. The security definition specifies that the security has to be maintained for all compatible environments. This means that the protocol must preserve security when running with $\mathsf{Env}_{\mathsf{abs}} \in \mathbb{E}_{\mathsf{abs}} \subseteq \mathbb{E}$. Hence, security in the abstract environment is a necessity for all protocols that consider the trivial Π_{e} as a valid environment.

SUFFICIENCY. The security in the abstract model is sufficient for security in the hybrid model if any attack that is possible in the hybrid model has an equivalent attack in the abstract model. In this case, the lack of any such attacks, which is implied by the security in the abstract model also means that there cannot be any valid attacks in the hybrid model. The fact that all adversaries in the hybrid model have equivalent adversaries in the abstract model is shown by Lemma 22 for the semi-abstract and abstract model equivalence and Corollary 9 for the transformation from hybrid to semi-abstract model.

5.6.4. Simplified Clocking and the Combined Interpreter

The abstract execution model leaves the adversary the power to clock each individual input to ideal functionalities. However, it is well-defined that the ideal functionalities only operate when they have sufficient inputs for the given round. Similarly, all outputs of ideal functionalities are written together. Hence, in many cases, it would be sufficient for the adversary to have just one clocking channel, where it can tell the ideal functionality to execute the next round. This is especially reasonable if the interpreters execute sequential code and there are no parallel DMACALL calls to several ideal functionalities. If the execution is sequential, then both input and output buffers for $\mathcal{F}_p^{\bowtie}$ can be simplified. If the execution is not sequential, then the order in which versions of $\mathcal{I}_i^{\rtimes}$ receive responses from $\mathcal{F}_p^{\bowtie}$ may affect which computation is performed next and the separate clocking channels are needed to deliver the outputs to $\mathcal{I}_i^{\rtimes}$.

All interpreters can be trivially joined to a master interpreter $\mathcal{I}$ simply by defining $\mathcal{I}$ as the collection of all interpreters. Let this be collection $\mathfrak{C}_0$ in Figure 52.

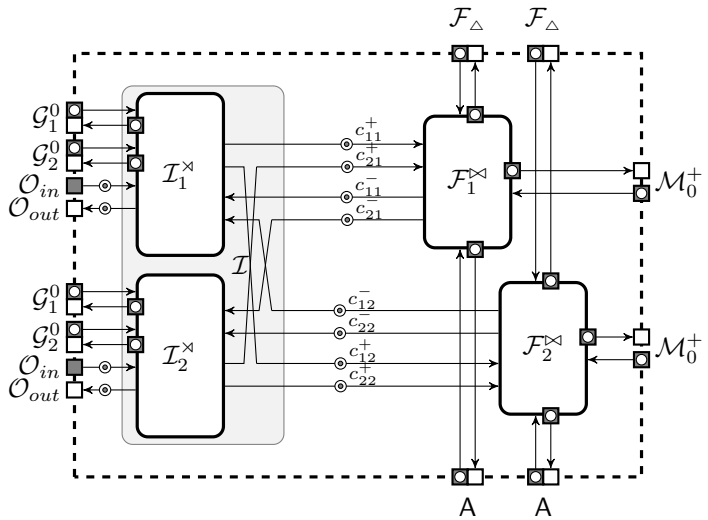


Figure 52: Trivial joining of the interpreters, $\mathfrak{C}_0$.

Let $c_{i,p}^+$ and $c_{i,p}^-$ be leaky buffers from $\mathcal{I}_i^{\bowtie}$ to $\mathcal{F}_p^{\bowtie}$ and from $\mathcal{F}_p^{\bowtie}$ to $\mathcal{I}_i^{\bowtie}$. The canonical $\mathcal{F}_p^{\bowtie}$ contains $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ machines that manage the collection of the inputs and distribution of the output signals as well as interactions with the adversary. The machines $\mathcal{C}_p$ and $\mathcal{D}_p$ are extracted from these machines to combine or distribute the messages as necessary. The interaction with the adversary still remains with $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$. Separating these machines defines $\mathfrak{C}_1$ in Figure 53 and $\mathfrak{C}_2$ in Figure 54. Note that $\mathcal{F}_p^{\bowtie}$ is extended to $\mathcal{F}_p^{\blacktriangleright\bowtie}$ and to $\mathcal{F}_p^{\blacktriangleright\blacktriangleleft}$ as its input buffers and the behaviour of $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ and $\mathcal{T}_{\mathcal{S}}^{\bowtie}$ changes. In $\mathfrak{C}_1$, there is a single leaky buffer d_p^+ replacing $c_{1,p}^+,\ldots,c_{n,p}^+$ for each functionality $\mathcal{F}_p^{\blacktriangleright\bowtie}$. In $\mathfrak{C}_2$, buffers $c_{1,p}^-,\ldots,c_{n,p}^-$ are replaced by d_p^-.

The execution of $\mathfrak{C}_1$ works as follows. When $\mathcal{I}_i^{\bowtie}$ sends a DMACALL call to $\mathcal{C}_p$ then $\mathcal{C}_p$ processes this call. If $\mathcal{C}_p$ has collected enough messages to activate the functionality, then it writes the DMACALL to d_p^+. If not, then it simply gives control back to $\mathcal{I}_i^{\bowtie}$. $\mathcal{I}_i^{\bowtie}$ can continue the execution from the DMACALL call like it did when it did not have to clock the respective buffer itself in $\mathfrak{C}_0$. Hence, the execution of $\mathcal{I}_i^{\bowtie}$ is slightly modified to account for the new clocking behaviour.

It is reasonable to expect that the interpreter code is public and that the adversary can always know which lines of code are executed next. In addition, it is also reasonable to assume that it is known which parties participate in any round of computations and the adversary can use this information. A protocol specification has a predictable subprotocol invocation pattern if an adversary can always predict which buffers with which tags any interpreter $\mathcal{I}_i^{\bowtie}$ is going to write activated. A functionality $\mathcal{F}_p^{\bowtie}$ has predictable input signatures if the adversary can always predict the input signature for the current round of computation.

Lemma 26. *If a protocol specification has a predictable subprotocol invocation pattern and all ideal functionalities have predictable input signatures, then the*

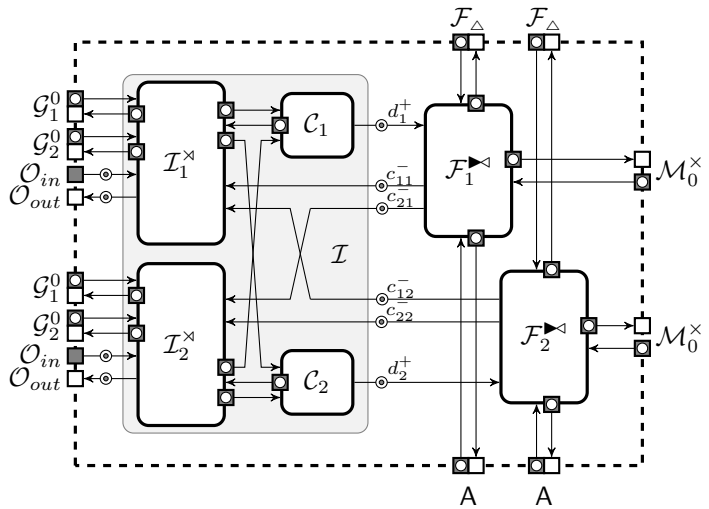


Figure 53: Combining interpreters and buffers to $\mathcal{F}_p^{\blacktriangleright\triangleleft}$, $\mathfrak{C}_1$.

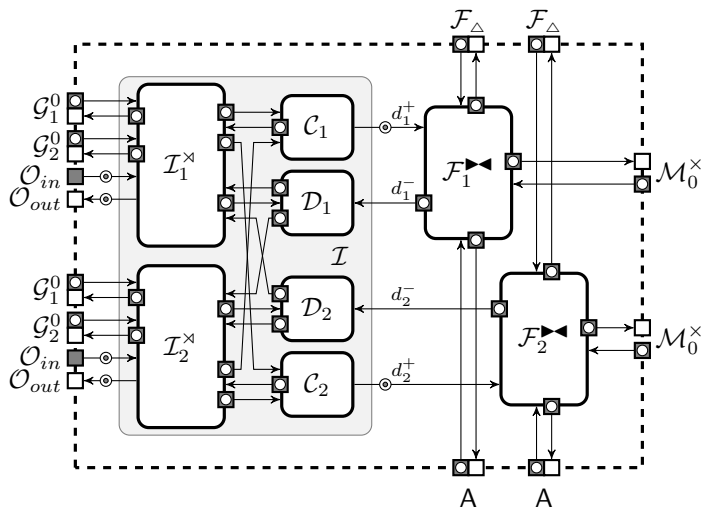


Figure 54: Combining interpreters and buffers to and from $\mathcal{F}_p^{\blacktriangleright\blacktriangleleft}$, $\mathfrak{C}_2$.

collections $\mathfrak{C}_0$ *and* $\mathfrak{C}_1$ *are equivalent.*

Proof. Let A_0 be the original adversary against $\mathfrak{C}_0$ with the trivial joined interpreter and all buffers and A_1 be the adversary against $\mathfrak{C}_1$. If adversary A_0 or A_1 clocks some input to $\mathcal{I}_i^{\bowtie}$ then, by the predictable subprotocol invocation, the adversary knows which buffers $b_{i,p}^+$ will get new messages from $\mathcal{I}_i^{\bowtie}$ and what is the tag that is leaked by the buffer.

In order to show equivalence, both adversaries A_0 and A_1 must be able to execute all their actions against the other version of the protocol. When using A_0 with $\mathfrak{C}_1$ instead of $\mathfrak{C}_0$, the missing buffers to $\mathcal{F}_p^{\bowtie}$ can be perfectly simulated. If A_0

clocks $b_{i,p}^{+}$ then, by the predictable input signature assumption, it is known if this clocking causes $\mathcal{F}_p^{\bowtie}$ to execute or $\mathcal{T}_{\mathcal{R}}^{\bowtie}$ is still collecting inputs and not executing yet. Equivalent A_1 would clock the buffer d_p^{+} to $\mathcal{F}_p^{\blacktriangleright\bowtie}$ when $\mathcal{F}_p^{\bowtie}$ would be ready to execute based on the clocking of A_0. All other actions of A_0 can be carried out against $\mathfrak{C}_1$ without modifications.

The clocking of d_p^{+} by A_1 can be simulated in $\mathfrak{C}_0$ by clocking the respective buffers $b_{i,p}^{+}$ for all parties. If A_1 clocks this buffer, then the equivalent adversary against $\mathfrak{o}$ would clock all $b_{i,p}^{+}$ in the fixed order. After each clocking, it gets the control back as the master scheduler and, therefore, can finish clocking all independent buffers the same way as $\mathcal{C}_p$ clocks them. The leakage on any of the separate buffers can be simulated using the predictability of the execution pattern. □

In $\mathfrak{C}_2$ in Figure 54, whenever $\mathcal{I}_i^{\rtimes}$ receives a message from $\mathcal{D}_p$ it runs as usual. When it finishes its usual execution, it gives control back to $\mathcal{D}_p$. $\mathcal{D}_p$ can then send the message to the next $\mathcal{I}_j^{\rtimes}$ until all interpreters have received the message corresponding to the input of $\mathcal{D}_p$. The equivalence to $\mathfrak{C}_2$ can only be achieved for sequential scheduling where the order of the messages arriving does not affect which line of code is executed next. Concretely, a protocol run by an interpreter has sequential scheduling if at most one sub-protocol is active at any time and new inputs from Π_{e} are also not processed before the sub-protocol is complete.

Lemma 27. *If a protocol specification with sequential scheduling has a predictable subprotocol invocation pattern and all ideal functionalities have predictable input and output signatures, then the protocol the collections* $\mathfrak{C}_0$ *and* $\mathfrak{C}_2$ *are equivalent.*

Proof. Equivalence of $\mathfrak{C}_0$ and $\mathfrak{C}_1$ was shown in Lemma 26. Let A_1 be the adversary against $\mathfrak{C}_1$ and A_2 be the adversary against $\mathfrak{C}_2$. The following proof considers the equivalence of $\mathfrak{C}_1$ and $\mathfrak{C}_2$.

Thanks to sequential scheduling, there is at most one buffer d_p^{+} and d_p^{-} per protocol instance t that has any message in $\mathfrak{C}_2$. If A_1 gives input to $\mathcal{F}_p^{\bowtie}$ in $\mathfrak{C}_1$, then $\mathcal{F}_p^{\blacktriangleright\bowtie}$ writes messages to $c_{i,p}^{-}$ for parties in the output signature. In $\mathfrak{C}_2$, the output of $\mathcal{F}_p^{\blacktriangleright\blacktriangleleft}$ is immediately delivered to all $\mathcal{I}_i^{\rtimes}$ in the output signature and a new message is computed to d_p^{+} or the protocol returns an output. In $\mathfrak{C}_1$, the clocking of the $\mathcal{F}_p^{\blacktriangleright\bowtie}$ output is not immediate, instead A_1 can clock the buffers to parties separately and A_1 may clock inputs from $\mathcal{O}_{in}$ to $\mathcal{I}_i^{\rtimes}$ for the same protocol instance. However, due to sequential scheduling, inputs from $\mathcal{O}_{in}$ are not processed before the message from $c_{i,p}^{-}$ is received. Hence, the early clocking of the messages to $\mathcal{I}_i^{\rtimes}$ in $\mathfrak{C}_2$ does not change the steps in the execution of $\mathcal{I}_i^{\rtimes}$. In $\mathfrak{C}_1$, the interpreter $\mathcal{I}_i^{\rtimes}$ will give the same next messages and, therefore, the same value will appear in d_p^{+}.

Based on the previous description, In order to convert A_1 to work against $\mathfrak{C}_2$, the buffers $c_{i,p}^{-}$ must be simulated and d_p^{+} must be simulated as empty until A_1 has clocked all the messages out from $c_{i,p}^{-}$. To convert A_2 to work against $\mathfrak{C}_1$, the

buffers $c_{i,p}^{-}$ must be clocked right after clocking d_p^{+}.

Note that the abstract adversary only alters memory locations of outgoing DMACALL calls. The return values are written to the memory by $\mathcal{F}_p^{\blacktriangleright\!\triangleleft}$ or $\mathcal{F}_p^{\blacktriangleright\!\blacktriangleleft}$ and they appear in the memory at the same time. However, the interpreters are executing with different timing and compute their DMACALL calls at different times. However, neither adversary has a reason to want to modify the memory before all interpreters in $\mathcal{I}$ have written their output. The modification can be done when interpreters have already produced the message to d_p^{+}. □

5.7. Abstract Model and Arithmetic Black-Box

Table 2 summarises the derivation of the abstract model and the assumptions needed to reach any step from the hybrid execution to the abstract model. Note that each assumption is marked for the steps in the transformations where the property was used. However, the transformations themselves are in order and essentially, to arrive at the abstract model from the hybrid model, all assumptions have to be met.

The hybrid execution model is the initial protection domain. Each party sends its inputs to the respective ideal functionalities and gets the results. The adversary can completely control the execution timing and the actions of the corrupted parties. The model with simplified scheduling is derived in Section 5.3. This introduces an adversary that only modifies messages that the party would compute and does not send arbitrary messages. The shared memory model derived in Section 5.4 separates the memory from the parties and specifies the details of the protocol execution. The abstract memory is introduced in Section 5.5 to hold the values represented by the values in storage domains. Firstly, the memory is separated into two machines that are synchronised. Next, the memory holding the private values of the storage domains is gradually removed to arrive at the protocol execution with only abstract memory. Finally, in Section 5.6, the protocol execution model is simplified to the abstract model.

The arithmetic black-box (ABB) was introduced in Section 2.1.8 and compared with the formalisation of protection domains in Section 3.7. The abstract model is in many ways closer to the ABB formalisation than the hybrid model was. For example, ABB allows accessing secret data through handles which have the same role as the memory locations used in the abstract model. The operations inside ABB are like the canonical ideal functionalities or the local functionalities $\mathcal{G}_q^0$. They expect the command from all or a sufficient number of parties to use the secret value and then perform the computations and store the result in a handle. The results are stored in an internal memory like $\mathcal{M}_0$ in the abstract model. In both cases, the functionalities can also have direct interactions with the adversary. The interpreters are not part of the ABB description. The abstract model is derived with the intention of considering a specific protocol that is executed by the interpreter. However, a possible protocol is such that expects public commands

Table 2: Summary of the transformations to the abstract model and the assumptions made about the protocol Π and protection domain in Π_e.

	Hybrid model	Simplified scheduling	Shared memory for all parties	Separated memory $\mathcal{M}$ and $\mathcal{M}_0$	Abstract memory	Abstract execution model	Joint interpreters
Canonical ideal functionalities		+	+	+		+	+
Tight scheduling		+					
Robust against malformed inputs		+					
Secure against rushing		+					
Well-formed protocol			+	+			
Canonical protocol			+	+			
Meaningful local operations					+		
Transparent local operations					+		
Hiding storage					+		
Modification-aware storage					+		
Limited-control storage					+		
Negligible oblique modifications					+		
Output-isolated					+		
Simulatable protection domain					+		
Embeddable protection domain						+	
Predictable subprotocol invocation							+
Sequential scheduling							+

from Π_{e}. In the abstract world, such protocol gets these inputs from $\mathsf{Env}_{\mathsf{pd}}$ that represents all parties. In this case, the abstract model with such a protocol is like the ABB. The main difference is that $\mathsf{Env}_{\mathsf{pd}}$ can directly write inputs to the memory through $\mathcal{O}_{in}$ and read the outputs from $\mathcal{O}_{out}$. ABB has only the capability to receive public inputs belonging to some party and to give public outputs.

The capability to consider only a sub-protocol represented by Π and leave the rest of the computations to Π_{e} is the strength of the modular secure computation formalisation. In addition, the strength of the abstract model is that it is sufficient for security proofs of sub-protocols. Hence, the existence of the interpreters and the ability to have private inputs to Π is a crucial factor separating the abstract model from the ABB formalisation. However, even when considering them both, the whole formalisation looks like an ABB for $\mathsf{Env}_{\mathsf{pd}}$. The results that a proof in the abstract model is sufficient for security in the hybrid model gives a simple way to define new protocols Π that are as secure as some canonical ideal functionalities $\mathcal{F}$ so that the set of the protocols in $\mathcal{F}_{\mathsf{pd}}$ can be extended by $\mathcal{F}$. As a benefit, the abstract model derivation has specified several conditions that are reasonable but must be met in order to analyse the security of the protocol in this model.

5.8. Abstract Model and Formal Verification

One of the initial motivations for developing the abstract model was to develop a system to formally verify relatively generic algorithms that can be executed using various MPC frameworks. Achieving this goal requires two tasks. Firstly, to verify the derivation of the abstract execution model itself in order to further establish its correctness and soundness for security proofs. Secondly, to define an environment where the abstract security can be proven. Both of these tasks are likely complicated undertakings but would significantly increase the assurance that the proposed protocols are secure. The hope is that abstract execution simplifies the protocol description and removes the concrete storage domains and, therefore, reduces the complexity of the protocol in a manner that may be easier for the formal tools to address. For example, to rely on the definitions of hiding and modification awareness rather than the properties of the algebraic structures of the storage domains. Of course, it is possible to prove the results from algebraic assumptions alone but the resulting formal proofs would be much longer, which is a real concern given the typical length of current formal proofs. That is, one can view the abstract execution model as try to modularise and simplify proofs before formal verification. A proof in the abstract model would have to focus on the order in which the public values are created and which inputs are used and the simulation of the public values. Focusing on the values in the memory enables us to better describe the protocol as one unit and not a collection of parties also removing the complexity that comes from considering all possible orders of scheduling.

A significant difficulty of using proof assistants to derive and verify formal proofs for MPC is that they are designed for game-based proofs and well suited

for security notions that are game-based themselves, but it has been difficult to achieve simulation-based security, especially with composition. A good overview of computer-aided cryptography is available in [12]. The difficulties with simulation-based proofs would also affect any proofs in the abstract model. Note that while there is no established common method to formally prove security of secure computation, there are approaches to formally verify some concrete secure computation protocols as well as to describe composability frameworks.

Isabelle/HOL[2] was extended in [38] to allow simulation-based proofs for passive security and such proofs were demonstrated for the two-party multiplication using Beaver's triples and oblivious transfer protocols. There are also verified implementations of passively secure garbled circuits [1] and [2] that also needed to define simulation-based proof techniques for EasyCrypt [3] [14, 15]. Passive security of an information-theoretically secure three-party count retrieval protocol is addressed in [130]. They also use simulation but use it as part of the proofs rather than extending EasyCrypt. The first formal proof for an MPC protocol that has active security was done using EasyCrypt in [77]. These proofs required defining a new set of security definitions that are better suited for EasyCrypt but imply simulation-based security. The mobile adversary model was considered in [66].

A more general look at proofs of universal composability in EasyCrypt was taken in [48]. This work considers specifying the ideal and real functionalities, defining the simulators and demonstrating the validity of the simulator, allowing to combine proofs to a proof of the composed protocol. They define EasyCrypt modules to express these executions with any adversary and environment and express the statement that real and ideal executions are indistinguishable. Their formalisation does not cover the full capabilities of the universal composability framework but should be a suitable basis for the case required to formalise MPC as done in this thesis. EasyCrypt has been enhanced to also consider adversarial computational complexity [13] to include statements about adversaries' success in relation to their computation resources. This enables them to propose another formalisation of UC in EasyCrypt. Another line of work formalizes the interactive Turing machines used in UC through new interactive lambda calculus to support tools for UC proofs [97]. Constructive cryptography [105] is also a variant of defining secure protocols through composition of possibly less secure components, it has been modelled in HOL [16, 104]. The question of formalizing verification of observational equivalence is addressed in [68] with the goal of proving equivalence without explicit use of bisimulation. However, all these can be seen mostly as early work still in need of further tool development rather than solutions ready to use. It would be interesting to combine these formalisations with the formalisation of MPC from Chapter 3 and the abstract model derived in this chapter.

[2] https://isabelle.in.tum.de/

[3] `http://www.easycrypt.info/`

5.9. Application of the Abstract Model

This section gives some examples of how existing secure computation frameworks fit into the formalism of this thesis and what changes when the security of some protocols is proven in the abstract model instead of the real model.

Overall, the conditions that the framework and protocol have to satisfy in order to allow the transformations from the real to the abstract model and back are as follows.

1. All ideal functionalities are canonical (Definition 14).
2. All ideal functionalities have tight scheduling (Definition 32).
3. All used local functionalities are meaningful (Definition 15) and transparent (Definition 44).
4. Used storage domains are hiding (Definition 8), modification-aware (Definition 10) and have the limited control property (Definition 12).
5. The protocol description is well-formed (Definition 38) and in a canonical form (Definition 42).
6. The protocol is robust against malformed inputs (Definition 34) and secure against rushing (Definition 36).
7. The protocol is output-isolated (Definition 45).
8. The environment (the protection domain Π_e) has to be embeddable (Definition 50) and simulatable (Definition 31).

Items 1, 2, 3, and 4 characterise the protection domain itself. For any existing protection domain, these could be proven once and then used for different kinds of protocols. The condition in item 8 is actually also mainly the requirement for the protection domain, as the part of the environment that it applies to, is the part that is executed in the protection domain as Π_e. The protocol can be quite easily written to satisfy conditions in item 5, as well as to manage malformed inputs as required in item 6. Security against rushing that is part of item 6 can be achieved for secure multiparty computation protocols as discussed in connection with Theorem 12. The main protocol property that must be carefully considered is the output isolation in item 7. However, it is common for many protocols to return their outputs right after the outputs are computed and, therefore, according to Lemma 14, output isolation is also trivially achieved.

5.9.1. Passive Security with Secret Sharing

Many honest majority protocols that are based on secret sharing and are secure against a passive adversary can be achieved without a private setup. Such protocols are based on the linear properties of the sharing scheme. The storage domain and the allowed corruption for it are defined by the sharing scheme. Linear combinations are local functionalities where the meaning is defined by the linear combinations, and multiplication and other computations are represented

as ideal functionalities operating on the shares. The canonical ideal functionalities for the passively secure case are very simple. They do not leak anything to the adversary, and they always succeed in reconstructing the inputs and, therefore, also always give valid outputs. In addition, the adversary is not allowed to do any modifications in the case of passive corruption and therefore, the modification awareness and limited control over the storage domain are trivial for a passive adversary. Simulatability and embeddability of such protection domains are again trivial since there is no private setup information. Note that the passive adversary does not perform any modifications. Therefore the transparency of local functionalities is irrelevant since there can also be no modifications that need to be propagated through the functionalities. In addition, security against malformed inputs is trivial since the adversary cannot modify the computations and planned computations should not result in malformed inputs. For the following examples, security against rushing is achieved thanks to using symmetric programs (discussed in Section 5.3.3), where all parties need to give input to all ideal functionalities.

5.9.2. Sharemind Protection Domain

The Sharemind computation framework with a protection domain using three parties and additive secret sharing as described in Section 4.8.1 is a suitable example for the passive security case. The storage domain uses additive secret sharing over rings $\mathbb{Z}_{2^\ell}$ for three parties. Hence, $[\![x]\!] = ([\![x]\!]_1, [\![x]\!]_2, [\![x]\!]_3)$ where $x \in \mathbb{Z}_{2^\ell}$ and $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$. This is hiding as long as at least one party remains honest. The protection domain using additive secret sharing and three parties is secure as long as at most one party is corrupted due to the properties of the computation algorithms. The sharing functionality $\mathcal{S}$ generates two random values $[\![x]\!]_1, [\![x]\!]_2$ and computes $[\![x]\!]_3 = x - [\![x]\!]_1 - [\![x]\!]_2$. The reconstruction functionality $\mathcal{R}$ simply sums $[\![x]\!]_1, [\![x]\!]_2, [\![x]\!]_3$ to learn x.

The core computation functionalities are linear combinations that are local functionalities and the multiplication protocol. The linear operations defined by local functionalities are meaningful by definition. The multiplication protocol is given in Section 4.8.1 as Algorithm 12. However, many more functionalities have been proposed for the Sharemind protection domain, for example, equality checks and division in [33], sorting in [31] or floating-point arithmetic in [87]. The approach taken by the protocol design for Sharemind naturally defines the ideal protocols as canonical ideal functionalities, making it easy to consider Sharemind in the current formalisation. The functionalities can also be easily considered as having tight scheduling.

Greater Than Comparison. A simple protocol to compare signed ℓ-bit values in Sharemind from [28] is given in Algorithm 17. Note that it assumes that the most significant bit of the inputs is used to denote the sign of the integer and that there is no overflow for the subtraction. It uses a bit shift protocol to ex-

tract the bit. The most significant bit is one if the number is negative and this bit is extracted with the right shift protocol. Hence, $x \in \{2^{\ell-1}, \ldots, 2^{\ell}-1\}$ in interpreted as a negative number. This protocol returns a fresh output right after it is computed. Therefore, it is output-isolated, according to Lemma 14. It also uses only a meaningful local operation to compute the subtraction and uses canonical ideal functionalities otherwise. It can be implemented so that it is well-formed and canonical. Memory alignment can be ensured by agreeing on the memory locations beforehand. A security proof in the abstract model has to manage an adversary that does not see any values but can schedule the protocol. It has to show that it is clear when the protocol produces outputs so that the outputs of the real and ideal case can be clocked synchronously.

Algorithm 17: Sharemind `GreaterThan`($[\![x]\!], [\![y]\!]$) protocol.

Input: $[\![x]\!]$,$[\![y]\!]$ for $x, y \in \mathbb{Z}_{2^\ell}$
Output: $[\![w]\!]$ where $w = 1$ if $x > y$ and $w = 0$ otherwise
$[\![d]\!] = [\![y]\!] - [\![x]\!]$
$[\![v]\!] =$ `ShiftRight`$([\![d]\!], \ell - 1)$
$[\![w]\!] =$ `Reshare`$([\![v]\!])$
return $[\![w]\!]$

Finding the Minimum. Different comparison-based sorting algorithms can be implemented in Sharemind as proposed in [31]. A simplification of sorting is a protocol to take a minimum of two items in Algorithm 18. It uses a protocol `Shuffle` that gets two inputs and gives out fresh shares for them in a random order.

Algorithm 18: Sharemind `Min`($[\![x]\!], [\![y]\!]$) protocol.

Input: $[\![x]\!]$,$[\![y]\!]$
Output: $[\![w]\!]$ where $w = min(x, y)$
$[\![a]\!], [\![b]\!] =$ `Shuffle`$([\![x]\!], [\![y]\!])$
$[\![w]\!] =$ `GreaterThan`$([\![a]\!], [\![b]\!]))$
$w =$ `Publish`$([\![w]\!])$
if $w = 1$ **then**
    return $[\![b]\!]$
else
    return $[\![a]\!]$

The problem with this algorithm is that it is computing the values $[\![a]\!], [\![b]\!]$ before it becomes known which of them is the output. Hence, it is not falling under Lemma 14 because the output is not returned right after it is computed. In the case of Sharemind, it is very easy to make this protocol fall under the conditions of Lemma 14 by adding a `Reshare` protocol after the output is chosen and before it is returned. However, it makes the protocol more complex. The other possibility is to prove output isolation separately. In this case, the proof is

relatively simple as the passive adversary cannot modify the values in the protocol and the scheduling of a well-formed protocol like this does not change what is computed. Hence, it is possible to define some default scheduling and simply clock through the protocol to learn the value w and the output $\mathcal{S}^+$ gives in $\mathfrak{F}_4$ and simulate the protocol timing for the adversary who is interacting with $\mathfrak{F}_3$.

The final security proof of the protocol simply has to show that it can simulate the value w that is published. The exact distribution of w depends on the knowledge that there is for the distribution where x and y come from and how likely it is that they are equal. On the other hand, if you use a version of `GreaterThan` that outputs 1 with probability $1/2$ when $x = y$, then the protocol is unconditionally secure.

5.9.3. Active Security with Private Setup

Compared to the passive security model, the protocols in the active security model have to consider the actions of an active adversary in the protocol and to consider storage domains that offer some security for modifications as well as hiding property. In addition, these protocols require some setup information.

SPDZ Protection Domain. The SPDZ computation framework [59] is achieving active security with a storage domain that contains additive secret sharing in a field and an authentication code for the shared value. SPDZ operates in the precomputation model, where shares of random values and random multiplication triples are prepared before the online phase. The core of the online phase has remained the same as originally proposed in [59] with the changes in how the reconstruction verifies the shared result. The up-to-date verification is described in [54]. However, for the precomputation phase, there have been several different approaches [54, 59, 88, 89]. The following description focuses on the online phase and assumes that there is some secure precomputation phase available.

The storage domain is based on additive secret sharing in a field $\mathbb{F}$. A secret $x \in \mathbb{F}$ is shared as $[x] = \{([\![x]\!]_1, [\![\alpha x]\!]_1), \ldots, ([\![x]\!]_n, [\![\alpha x]\!]_n)\}$ where party $\mathcal{P}_i$ has $[x]_i = ([\![x]\!]_i, [\![\alpha x]\!]_i)$ and $[\![x]\!]$ denotes the additive shares of x and $[\![x]\!]_i \in \mathbb{F}$. $[\![\alpha x]\!]$ denotes the additive shares of $\alpha \cdot x$ and α is the private key of the authentication mechanism. The private key is also additively shared and party $\mathcal{P}_i$ only has $[\![\alpha]\!]_i$. It is the role of the setup $\mathcal{F}_\triangle$ to distribute $[\![\alpha]\!]$. The sharing functionality takes the input x and $[\![\alpha]\!]$ to produce authenticated shares $[x]$. Internally, it computes α and $\alpha \cdot x$ and then uses additive secret sharing to produce fresh shares for both x and αx. This storage domain is hiding as long as at least one party remains honest. The reconstruction functionality $\mathcal{R}$ reconstructs x, αx and α internally and verifies $\alpha x = x \cdot \alpha$. The security analysis of the authentication mechanism shows that the adversary can only find a suitable modification of the corrupted shares of $[\![x]\!]$ and $[\![\alpha x]\!]$ if it could also guess the key α. Hence, if the key space is big enough, then the storage domain is modification-aware and the only allowed modification is to invalidate the shared value. Any value that is public or in the local storage of the

corrupted party can be freely modified as it does not have any integrity protection. For hidden values that are authenticated, the adversary can only modify them to $\perp$ and a random non-zero modification of the corrupted party's share achieves the same effect. Therefore, the storage domain also has limited control property. If the adversary has broken the hiding property, then it also can reconstruct the key α and trivially modify all values.

The ideal functionality for the online phase of SPDZ is commonly described as an ABB and not the modular description required here. However, the basic functionalities inside the ABB can also be formalised as canonical ideal functionalities or local functionalities. The building blocks of the linear combination are local functionalities. These are either to add or subtract shared values or multiply the shared value with a public value. All these are clearly meaningful. They are also transparent, for example, if the adversary modifies output shares $(\llbracket y \rrbracket_i, \llbracket \alpha y \rrbracket_i)$ to be $(\overline{\llbracket y \rrbracket_i}, \overline{\llbracket \alpha y \rrbracket_i})$ and the operation was addition $x+z$ or subtraction $x-z$, then the modification that changes the input $(\llbracket x \rrbracket_i, \llbracket \alpha x \rrbracket_i)$ to $(\llbracket x \rrbracket_i + \overline{\llbracket y \rrbracket_i} - \llbracket y \rrbracket_i, \llbracket \alpha x \rrbracket_i + \overline{\llbracket \alpha y \rrbracket_i} - \llbracket \alpha y \rrbracket_i)$ achieves the same effect. For a multiplication $c\llbracket x \rrbracket$ with a public value c, the modification on the input is $(\llbracket x \rrbracket_i + (\overline{\llbracket y \rrbracket_i} - \llbracket y \rrbracket_i) \cdot c^{-1}, \llbracket \alpha x \rrbracket_i + (\overline{\llbracket \alpha y \rrbracket_i} - \llbracket \alpha y \rrbracket_i) \cdot c^{-1})$. In both cases, if the adversary manages an oblique modification that the extractor does not catch, then the input modification is also oblique. For SPDZ, oblique modification means that the modification was successful at changing the value so that the reconstruction succeeds and its probability is equivalent to guessing the key.

In the basic case of protocols, the multiplication protocol is the only one that has to be corresponding to a canonical ideal functionality. The corresponding canonical ideal functionality collects both inputs, reconstructs them, computes multiplication and gives fresh shares of the output to all parties. The ideal functionality knows α from the $\mathcal{F}_{\triangle}$ so that it can verify the opening and generate valid shares. The key α is only used by the reconstruction and sharing components in the modular ideal functionality. The real multiplication protocol of SPDZ is using precomputed triples using the idea from [17]. However, these triples are not considered as inputs to the multiplication functionality. Rather, the real multiplication protocol itself is in a hybrid model, where it could be thought of as first computing the triple and then performing the rest of the multiplication protocol like in Algorithm 19. Thanks to the fact that Beaver triples are random and with uniformly random shares then, so are the outputs of the multiplication protocol. If the random triple generation can also fail, then the ideal multiplication functionality has to allow the adversary to fail the protocol execution even if the inputs can be reconstructed correctly. Each party is expected to send their inputs as one message and then they expect the output of the functionality, which means that it has tight scheduling.

The SPDZ protection domain is simulatable thanks to the relatively simple setup that only produces $\llbracket \alpha \rrbracket$ where $\llbracket \alpha \rrbracket_i$ is the private parameter of party $\mathcal{P}_i$ and that no party knows α. The simulator can use the corrupted parties $\llbracket \alpha \rrbracket_i$

Algorithm 19: Multiplication $\texttt{Mult}([x],[y])$ using Beaver triples.

Input: $[x]$,$[y]$
Output: $[w]$ such that $w = xy$
$[a],[b],[c] = \texttt{ComputeBeaverTriple}()$ // where $c = ab$
$[e] = [x] - [a]$
$[d] = [y] - [b]$
$e = \texttt{Publish}([e])$
$d = \texttt{Publish}([d])$
$[w] = [c] + e \cdot [b] + d \cdot [a] + ed$
return $[w]$

and generates random simulated $[\![\alpha]\!]_j^{sim}$ to have a simulated α^{sim}. Thanks to the perfect hiding property of additive secret sharing, the shares of $[\![\alpha x]\!]$ and $[\![\alpha^{sim} x]\!]$ are uniformly random and the difference is only clear when all parties become corrupted. However, in case all parties become corrupted, then the simulator also learns the right α since it has all individual $[\![\alpha]\!]_i$ values and can adjust the view to use this value instead. In case of adaptive corruption, each time when a new party becomes corrupted, the remaining set of $[\![\alpha]\!]_j^{sim}$ for honest parties $\mathcal{P}_j$ has to be adjusted by the simulator to keep the original value of the simulated key that can be used to verify if reconstruction succeeds in any specific point in the protocol simulation. However, this is trivial for additive shares. If the last party becomes corrupted and the simulator needs to reveal all shares of this party, then these need to be recomputed to adjust the simulated key to the real key. However, this can also be done as it is the common step required in SPDZ simulation of publishing outputs and is also considered as patching in [56].

Multiplication as a Composed Protocol. Multiplication using Beaver triples is shown in Algorithm 19 and it can be seen as a functionality that is composed of precomputation, publishing and linear combinations. It can be proven secure in the abstract model. In the abstract model, the first step uses a canonical ideal precomputation functionality to compute the triple. The triple is written to memory $\mathcal{M}_0^{\times}$ and the adversary can only invalidate the triple if it wishes, but it remains private. Then, local functionalities are used to compute the values e and d to $\mathcal{M}_0^{\times}$. These values are also published to the public storage domains for all parties. Note that this differs from the local protection domain for each party. For a local value, the adversary can modify them. However, the local functionalities expect the input to be in the public protection domain - all parties should send the same e to the functionality in order for it to be meaningful. Hence, a suitable public domain is such that it ensures consistency of the value and does not allow modifications. As a final step, all the local computations are performed to compute w. Hence, in this case, the abstract world allows the adversary to control the timing of the publishing ideal functionalities and the precomputation and to see the values of e and d in $\mathcal{M}_0^{\times}$. This protocol can be implemented so that it is well-formed. There are no

conditional jumps and all local computations are done with local functionalities for the linear combination. The ideal precomputation functionality can be called so that it writes its outputs in a memory-aligned manner. Hence, this protocol can be specified in a canonical form. All inputs are expected to be in the field and therefore have a known length. The arithmetic of the protocol is secure against malformed inputs under reasonable assumptions.

Ideal Functionality for Multiplication. The desired ideal functionality corresponding to the multiplication can behave differently based on the actual behaviour of the precomputation and publishing functionalities. If these can abort, then also the multiplication functionality must have an explicit abort. However, if these functionalities only fail silently, then also the multiplication would only fail silently. For example, using the partial opening [59] instead of full publishing is a common optimisation of this protocol that could open e or d to a random value of the adversary's choosing. However, if this happens, then the result w is inconsistent and cannot be properly reconstructed, but the abort is not explicit in this protocol. Note that the adversary does not see the values of any shares in the abstract execution of the composed protocol or the corresponding abstract model with only the ideal functionality of multiplication.

Simulation of Multiplication. In order to prove security in the abstract model, the simulator has to translate all actions of the adversary in the abstract model of the multiplication protocol to actions against the ideal functionality in the abstract model. The simulator has two main goals. Firstly, the simulator has to generate the view that the adversary has in the abstract model version of the composed protocol. Secondly, the simulator has to decide if the adversary actions result in aborting the functionality (and whether it is silent or known to all parties). The view of the adversary consists of the values e and d that are published and it sees them even if it decides to force the publication to use some other value e' or d'. The simulation can use random values to simulate e and d as these values have a uniformly random distribution in the real protocol. Note that differently from the hybrid model, the simulator does not have to simulate the shares of the precomputation output since these are not present in the abstract model. In addition, the outputs of the protocol are not available to the adversary because they are in a hiding protection domain. Hence, there are no simulated outputs. Assume that the protocol execution is sequential. In this case, the simulation of the progress of the composed protocol is straightforward. The simulator internally simulates all buffers to track the protocol execution of the composed protocol. This simulation reveals when the composed protocol has computed its outputs or when e and d are computed. When the adversary clocks the output of the composed protocol, then the simulator clocks the output of the ideal functionality. Hence, in this case, the simulation description is focused on the main intuition about the security, which is that uniformly random values are suitable to simulate the published values.

Multiplication and Output Isolation. The difficulty of applying the transformation from hybrid to abstract model to the multiplication algorithm is in the

output isolation requirement. The outputs are computed using local operations as $[w] = [c] + e \cdot [b] + d \cdot [a] + ed$. Therefore, the basic result in Lemma 14 does not apply and output isolation must be proven separately. For this case, the proof has to rely on the randomness introduced by the precomputation functionality that gives $[a]$, $[b]$ and $[c]$. In both collections defined by the output isolation (Definition 45), the adversary sees the values e and d. In order to prove output isolation, the output shares of the precomputation in $\mathfrak{F}_4$ must be simulated by computing them from the outputs generated by $\mathcal{S}^+$ and reversing the local computations that lead to these outputs. Note that the shares of each party are independent. Hence, once e and d are fixed, and $[w]$ shares are available from $\mathcal{S}^+$. The simulator can fix uniformly random shares for $[a]_i$ and $[b]_i$ and compute the share $[c]_i$ for party $\mathcal{P}_i$ to simulate the precomputation results.

The previous idea holds if the adversary cannot abort the protocol during publishing e and d as is the case when all openings are verified as a batch later. If the adversary can abort the protocol during publishing, then the simulation is complicated, as the construction ϕ_{oi} has to learn the output shares and the values of e and d to simulate suitable values for the precomputation outputs in $\mathcal{M}^+$. Hence, in orderd to allow this the adversary has to be able to also abort the protocol after learning its output shares, hence, after $\mathcal{S}^+$ has generated the outputs.

The fact that the simulator needs to know the final output in order to simulate the precomputed values is also problematic if the adversary can modify the published values e and d. If the adversary chooses to modify these values then it means that the final output for $[w]_i$ would also not be the same as the one given by the ideal functionality. Hence, the multiplication protocol is such that the output-isolation requirement is truly complicated to prove.

Note that while this means that the abstract model is not convenient to prove the security of the multiplication protocol it does not invalidate the security of the multiplication. If the multiplication step is proven secure in the ABB model, then there are no output shares that need to match as there are no shares received from ABB. However, the assumption that the outputs of honest parties are uniformly random shares appears when the final outputs of the ABB are published and the shares of the honest parties are simulated so that the reconstruction can publish the same value.

Secure Integer Division with a Private Divisor. The integer division protocols from [135] are good examples of algorithms that are written so that their description is independent of the exact details of the protection domain. The algorithms expect data and operations to be in some finite field and the required ideal and local functionalities are explicit in the algorithm write-up. In principle, any protection domain supporting these could be used to implement the algorithm. In fact, the security proofs of these algorithms follow the same ideas as the abstract model using hiding of the shares and focusing on the values public to corrupted parties. The ideal functionality that they propose is a canonical ideal functionality where all parties send their inputs and it gives fresh shares of the output. The

adversary can abort the protocol. The following points to some possible issues in the security proofs from that paper. However, the protocol itself is, most likely, secure and simply needs a more refined security argument.

However, the need to consider shares is one step away from the full abstract model and complicates the proof. For example, the proof of Theorem 2 in [135] states that the view that the adversary can have of the shares is identically distributed to random shares. This is indeed true for the SPDZ protection domain, but some other protection domains could be such where some part of the shares is not fully random, for example, if it contains some part of the setup. In such cases, shares should be simulated using $\mathcal{S}_\delta^{\mathsf{sim}}$ from the hiding property. In addition, the observation that they could be simulated using random shares is slightly incorrect because some of these values are indeed derived from others using local computations and, therefore, some shares should have dependencies based on these operations[4]. When lifting these results further to the abstract model, the shares could be removed from consideration as the fact that intermediate shares in hiding storage domains can be simulated is already included in the transformation to the abstract model. Then, the focus would be fully on the values that are public to all or some parties.

The second problem with the security claims in [135] is that they prove privacy and do not discuss how the protocol can be simulated to give the same outputs as the ideal functionality. If another algorithm would want to use this protocol as a building block (to say that it is in a hybrid model with oracle access to this division functionality), then its security proof assumes that it gets output shares with the same distribution as the fresh shares, but currently, this is not proven. The framework proposed here addresses this issue when the output isolation of the protocol is proven.

5.9.4. Active Security with Honest Majority

The problem with the previous example was that the multiplication protocol did not fit our framework for MPC. This example considers the case of multiplicative linear secret sharing (introduced in Section 2.1.4) with linearly homomorphic commitments based on [53]. The aim is to show a case where protocols with active security and private setup fit the MPC framework set up in this chapter. The following first introduces the components of this scheme and then summarizes the storage domain and functionalities. This assumes that there is a broadcast channel that can be represented as an ideal functionality in our framework.

Shamir's Secret Sharing. In Shamir's (t,n)-threshold secret sharing scheme [129] a secret value $v \in \mathbb{F}$ is in some field $\mathbb{F}$ and shared as evaluations of a polynomial

$$q(x) = v + \sum_{i=1}^{t-1} a_i x^i$$

[4] A similar problem is solved in Theorem 3 in [135] that considers the second protocol.

where party $\mathcal{P}_i$ learns $q(e_i)$ for some publicly known e_i such that $e_i \neq e_j$ and $e_i \neq 0$. This defines a scheme for n parties with a threshold t. The secret can be reconstructed using at least t shares and interpolation. Note that this means that the reconstruction is a linear combination of the shares and some public constants. This sharing is linear and a share of the addition can be computed by each party summing their shares of the inputs locally. The multiplication for Shamir's scheme was defined in [71] and it is a special case of the Maurer's protocol considered in [53]. The idea is that all parties multiply their shares locally and then get shares of the multiplication result but these are shared with a polynomial with degree $2(t-1)$. If $2(t-1) < n$ then the multiplication result can be reconstructed using $2t-1 \leq n$ shares. In such case, the multiplication protocol can also be concluded with a step where each party shares its locally computed multiplication result and the parties together compute the reconstruction operation on the new shares to reduce the degree of the polynomial sharing the output back to $t-1$. Hence, for these cases, Shamir's scheme is also multiplicative.

Commitments. In general commitment schemes consists of two algorithms $\mathsf{Commit}(x) = (c,d)$ and $\mathsf{Open}(c,d) = x$. The idea is that a party can create a commitment c and release it, later it can release a decommitment string d that can be used to open the commitment. A commitment is hiding, if seeing c without d does not reveal information about x and binding, if a commitment c can only be opened to the value x that was used to compute the commitment. In the following, a series of commitments is needed. By $\lceil x \rfloor_i$ the following denotes a hiding commitment that is computed by party $\mathcal{P}_i$ that also holds the information needed to open the commitment. A commitment is linearly homomorphic if $\lceil x \rfloor_i$ and $\lceil y \rfloor_i$ can be used to compute $\lceil x+y \rfloor_i$ which is a valid commitment to $x+y$ and party $\mathcal{P}_i$ can compute the decommitment message needed to open $\lceil x+y \rfloor_i$. In the following, it is important that a commitment $\lceil x \rfloor_i$ can be transformed to a commitment $\lceil x \rfloor_j$. Moreover, the homomorphic property also has to hold for commitments by different parties, such that one can compute $\lceil x \rfloor_i + \lceil y \rfloor_j$. Note that the description of the protocol in [53] also considers how to build suitable linear commitments from the linear secret sharing scheme, but the following description uses just the abstract definition of the commitments. These commitments are binding and hiding assuming the set of parties that can reconstruct the sharing scheme remain honest.

Protection Domain. Together, the idea of the commitments and Shamir's sharing result in a storage domain where each value is shared by the Shamir's scheme and each share is committed to using the homomorphic commitment. The commitments are known to all parties. In this section, shared value x is denoted as $\langle x \rangle$ where $\langle x \rangle = \{x_1, \ldots x_n, \lceil x_1 \rfloor_1, \ldots, \lceil x_n \rfloor_n\}$ where each party $\mathcal{P}_i$ knows x_i and all commitments $\lceil x_1 \rfloor_1, \ldots, \lceil x_n \rfloor_n$ and the decommitment for $\lceil x_i \rfloor_i$. This storage is hiding for a collection of up to $t-1$ parties if the commitments are hiding. This is also modification aware if the commitments are binding. As all parties have the means to verify all shares then any modification by an adversary will be no-

ticed by the honest parties. Furthermore, as there is an honest majority there are always at least t honest shares and the sharing scheme is robust. Hence, in this case, the adversary can not modify nor invalidate the values in the storage domain. This means that the scheme is modification aware according to our definition, just that no modifications are possible. This is a verifiable secret sharing scheme that enables each party to verify each share that they see.

The operations needed to build a basic protection domain are sharing, reconstruction, addition and multiplication. Addition operation for $\langle x \rangle$ is computed by each party locally. Each party sums their shares and the needed commitments. This is clearly meaningful and results in the addition of the shared values. Reconstruction verifies the commitments and performs the interpolation. Multiplication (in Algorithm 23) and sharing algorithms need some sub-protocols to manage the commitments. These sub-protocols are the commitment transfer protocol `CTP` in Algorithm 20, commitment sharing protocol `CSP` in Algorithm 21 and commitment multiplication protocol `CMP` in Algorithm 22. For brevity, all algorithm descriptions omit the fact that if the opening of any commitments fails or is not opened to the expected value (e.g. 0) then the protocol also needs to abort as it indicates that the party opening the commitment is corrupted. In this case, all parties can exchange this information and as there is an honest majority they can always agree if the complaint is genuine or fake. A possible way to deal with any failure is to open the inputs of the corrupted party and restart the computation without them. However, for the purpose here we can assume that the protocols can simply abort.

A commitment transfer `CTP` is a functionality that takes as input a commitment $\lceil x \rfloor_i$ and outputs a fresh commitment $\lceil x \rfloor_j$ such that all parties can verify that the value in the commitment stays the same. In the protocol in Algorithm 20 each party locally computes a commitment that the party $\mathcal{P}_j$ has to be able to open to 0. If the opening succeeds then all parties can verify that the new commitment contains the same value as the input commitment.

Algorithm 20: Commitment transfer protocol $\texttt{CTP}(\lceil v \rfloor_i)$

Input: $\lceil v \rfloor_i$ know to all parties
Output: $\lceil v \rfloor_j$ known to all parties
$\mathcal{P}_i$ opens $\lceil v \rfloor_i$ to party $\mathcal{P}_j$
$\mathcal{P}_j$ computes $\lceil v \rfloor_k$ and sends it to all parties
All parties locally compute $\lceil v - v \rfloor_j = \lceil v \rfloor_i - \lceil v \rfloor_j$
$\mathcal{P}_j$ opens the commitment $\lceil v - v \rfloor_j$ to 0
Return $\lceil v \rfloor_j$

A commitment sharing protocol `CSP` is used to generate commitments to the Shamir's shares of the initial value. For input $\lceil x \rfloor_i$ it returns $\lceil x_1 \rfloor_1, \ldots, \lceil x_n \rfloor_n$. The commitments are again fresh and each party can verify the correctness of this protocol. Committing to the coefficients of the polynomial in Algorithm 21 ensures

that the resulting shares are shared with the polynomial with the right degree. Local computations and CTP ensure that the output commitments are to the right values and fresh commitments. Note that as a result each party $\mathcal{P}_j$ also knows v_j. Obtaining a secret sharing algorithm Share to get $\langle x \rangle$ is a simple extension of CSP. At first, the input party $\mathcal{P}_i$ commits to its input value v and shares $\lceil v \rfloor_i$ and then parties execute $\texttt{CSP}(\lceil v \rfloor_i)$. Note that during CTP each party $\mathcal{P}_j$ learns its output share v_j and CSP ensures that everyone knows the needed commitments $\lceil x_1 \rfloor_1, \ldots, \lceil x_n \rfloor_n$. Note that the party generating the share has control over the outputs that the other parties get as they choose the polynomial coefficients a_j. In order to ensure that the results are random shares there is a need to reshare these values. For example, we can repeat the sharing protocol so that each party shares the share that they receive and they then compute the reconstruction on the shared values.

Algorithm 21: Commitment sharing protocol $\texttt{CSP}(\lceil v \rfloor_i)$

Input: $\lceil v \rfloor_i$, party $\mathcal{P}_i$ knows v
Output: $\lceil v_1 \rfloor_1, \ldots, \lceil v_n \rfloor_n$ where $x_1, \ldots, x_n$ are shares of x
$\mathcal{P}_i$ chooses random $a_j \in \mathbb{F}$ for $q(x) = v + \sum_{j=1}^{t-1} a_j x^j$ where $v_k = q(e_k)$
$\mathcal{P}_i$ commits to a_j for $j \in \{1, \ldots, t-1\}$ as $\lceil a_j \rfloor_i$
All parties locally compute $\lceil v_k \rfloor_i = \lceil v \rfloor_i + \sum_{j=1}^{t-1} \lceil a_j \rfloor_i e_k^j$
$\mathcal{P}_i$ runs $\texttt{CTP}(\lceil v_k \rfloor_i)$ for each $\mathcal{P}_k$ to get $\lceil v_k \rfloor_k$
return $\lceil v_1 \rfloor_1, \ldots, \lceil v_n \rfloor_n$

A commitment multiplication protocol CMP is a functionality to produce a commitment to the multiplication result of the inputs and prove that the commitment is to the right value. The protocol in Algorithm 22 first computes a new commitment and then all parties verify that the commitment is right. Note that the common randomness can be chosen, for example, by each party choosing a random value and then all parties summing the values. This protocol fails if any of the opening fails or the value r_2 is not 0 which indicates that party $\mathcal{P}_i$ is cheating.

The protocol for multiplication is in Algorithm 23 and makes use of CMP and the fact that there is a public linear combination that can be used to represent the interpolation needed to reconstruct a value. A double subscript m_{ij} means that the value m_i has been shared and it is the j-th share. The linear combination is represented by the values r_j such that $w = \sum_{j=1}^{n} r_j w_j$. This functionality fails if CMP or CSP fails, after that it is local computation and all parties have means to later verify the outcome. Note that the execution of the CSP can be aligned so that first all parties commit to their polynomial and then they open to ensure that no party chooses their share based on the values of other shares. The respective ideal

Algorithm 22: Commitment multiplication protocol $\texttt{CMP}(\lceil v \rfloor_i, \lceil y \rfloor_i)$

Input: $\lceil v \rfloor_i$, $\lceil y \rfloor_i$
Output: $\lceil w \rfloor_i$ where $w = v \cdot y$
$\mathcal{P}_i$ computes $w = v \cdot y$ and commits $\lceil w \rfloor_i$
$\mathcal{P}_i$ chooses a random $\beta \in \mathbb{F}$ and publishes $\lceil \beta \rfloor_i$ and $\lceil \beta y \rfloor_i$
Other parties collectively choose random $r \neq 0$
All parties locally compute $\lceil r_1 \rfloor = r\lceil v \rfloor_i + \lceil \beta \rfloor_i$
$\mathcal{P}_i$ reveals r_1
All parties locally compute $\lceil r_2 \rfloor = r_1 \lceil y \rfloor_i - \lceil \beta y \rfloor_i - r\lceil w \rfloor_i$
$\mathcal{P}_i$ opens $\lceil r_2 \rfloor$ to 0
return $\lceil w \rfloor_i$

functionality for the multiplication can fail but if it succeeds then it outputs fresh shares of the multiplication results to all parties.

Algorithm 23: Multiplication protocol $\texttt{Mult}(\langle v \rangle, \langle y \rangle)$

Input: $\langle v \rangle$, $\langle y \rangle$, public reconstruction parameters $r_1, \ldots, r_n$
Output: $\langle w \rangle$ where $w = v \cdot y$
Each party $\mathcal{P}_i$ runs $\texttt{CMP}(v_i, y_i)$ giving $\lceil m_i \rfloor_i$
Each party $\mathcal{P}_i$ runs $\texttt{CSP}(\lceil m_i \rfloor_i)$ giving $\lceil m_{i1} \rfloor_1, \ldots, \lceil m_{in} \rfloor_n$
$\mathcal{P}_i$ computes $w_i = \sum_{j=1}^{n} r_j m_{ji}$
All parties locally compute $\lceil w_i \rfloor_i = \sum_{j=1}^{n} r_j \lceil m_{ji} \rfloor_i$ for all $i \in \{1, \ldots, n\}$
return $\langle w \rangle$

These protocols with $\langle x \rangle$ sharing with commitments give an example of an actively secure protocol that follows the pattern of secure computation outlined in Chapter 3. The addition protocol is local and sharing and multiplication give out fresh shares of the result. Hence, this can also be used as a basis to develop further algorithms using the abstract model developed in this chapter. The simulatability of this protection domain depends on the concrete instance of the commitments. For the case of the commitments built from the sharing scheme in [53] simulatability is straightforward as there are no secret parameters. Otherwise, the simulatability depends on the private setup and how it is used, especially in case of an adaptive adversary corrupting more parties during the protocol.

6. CONCLUSION

This thesis puts forth two approaches for simplifying security proofs for secure multiparty computation algorithms. Both are intended to make protocol design more efficient without sacrificing rigorous security proofs. In addition, they can enable more efficient protocols or more general descriptions of algorithms. Both approaches encourage a more modular analysis of secure multiparty computation frameworks and applications. The overall goal is the enable a standard library of secure multiparty computation algorithms and functionalities, ensuring that algorithms can be defined using their building blocks and later used in all secure computation frameworks that satisfy the preconditions of the algorithm. In order to achieve these goals, the thesis defines a new modular way to formalise the secure computation frameworks.

The first contribution is the possibility to consider input privacy as a sufficient security level for many protocols included in complex applications. Input privacy can be easier to prove than full security and can yield more efficient protocols. As shown in Chapter 4, it is often easy to extend an input-private protocol to a secure protocol, and input-private protocols can be combined to achieve composed protocols that are still input-private. An algorithm designer simply must show input privacy of their new protocol. If the protocol is deterministic and correct, then output predictability is achieved trivially. Otherwise, the security result also requires showing output predictability. Input-private components of a secure composed protocol can often give more efficient protocols as the notion is less restrictive than security. The main restriction and avenue for future work regarding input privacy is generalising the notion and composition results to the active security model.

The second contribution is the ability to use the modular description of secure computation and simplify it to the abstract execution model considered in Chapter 5. There are clear conditions that need to be satisfied by protection domains and protocols to do the security proofs in the abstract model. These conditions are separate for the protection domain and the protocol, enabling us to consider these independently when using the framework. For example, protection domains can be shown to fit the modular formalisation of MPC with canonical ideal functionalities as was done for Sharemind and SPDZ in Section 5.9. The designers of the protection domain can do this part. The protection domains are often used by others who will use them to implement their algorithms. People designing new protocols can define the necessary functionalities and show that the protocol satisfies the conditions set by the transformations. For protocols that achieve output isolation trivially, it is not required to consider which protection domain or storage domains are used to implement the protocol. Any protection domain that has the necessary functionalities can be used to execute the algorithm securely. Proving output isolation may require making some assumptions regarding the used storage domains. In addition, any such protocol can be safely considered to extend

the protection domain with the new functionality, and it is explicitly clear what the new functionality and protection domain look like. The main complication of using this approach to prove the security of algorithms lies in showing output isolation. It would be beneficial to derive more results like Lemma 14 characterising the structure of the output-isolated protocols.

The approaches formalised and analysed in this thesis are intuitively simple. However, the detailed analysis discovered some underlying conditions that, while often natural for secure multiparty computation protocols, should still be explicitly considered. For example, these are the output-isolation or output predictability properties. As an extension of this work, it would be beneficial to go over these results in some proof assistant. Further verification would help to either be confident that all crucial details were noticed in the proofs and definitions of this thesis or to find and fix any such missing details. It would be especially valuable if the formalisation of the abstract model and the protection domain was available in some proof assistant that would allow to prove the security of new algorithms in the abstract model or to prove input privacy and output predictability. In addition, it would be interesting to extend the formalisation of the modular MPC framework, for example for the case of mobile adversary and corruptible ideal functionalities, and explore if the results of this thesis can be extended to these additions.

BIBLIOGRAPHY

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. Verified Implementations for Secure and Verifiable Computation. *IACR Cryptol. ePrint Arch.*, page 456, 2014.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A Fast and Verified Software Stack for Secure Function Evaluation. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1989–2006. ACM, 2017.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. Enforcing Ideal-World Leakage Bounds in Real-World Secret Sharing MPC Frameworks. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 132–146. IEEE Computer Society, 2018.

[4] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From Keys to Databases - Real-World Applications of Secure Multi-Party Computation. *Comput. J.*, 61(12):1749–1771, 2018.

[5] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and Performance of Programmable Secure Computation. *IEEE Security & Privacy*, 14(5):48–56, 2016.

[6] Yonatan Aumann and Yehuda Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In Salil P. Vadhan, editor, *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, volume 4392 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007.

[7] Yonatan Aumann and Yehuda Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *J. Cryptol.*, 23(2):281–343, 2010.

[8] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A General Composition Theorem for Secure Reactive Systems. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354. Springer, 2004.

[9] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The Reactive Simulatability (RSIM) Framework for Asynchronous Systems. *Inf. Comput.*, 205(12):1685–1720, 2007.

[10] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure Computation Without Authentication. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 361–377. Springer, 2005.

[11] Boaz Barak and Amit Sahai. How to play almost any mental game over the net - concurrent composition via super-polynomial simulation. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 543–552, 2005.

[12] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

[13] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2541–2563. ACM, 2021.

[14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A Tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[15] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[16] David A. Basin, Andreas Lochbihler, Ueli Maurer, and S. Reza Sefidgar. Abstract Modeling of System Communication in Constructive Cryptography using CryptHOL. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.

[17] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[18] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 777–804, 2016.

[19] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *Proc. of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 478–492, Washington, DC, USA, 2013. IEEE Computer Society.

[20] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. Cryptology ePrint Archive, Report 2013/426, 2013.

[21] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of Garbled Circuits. In *Proc. of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, USA, 2012. ACM.

[22] Mihir Bellare and Phillip Rogaway. Robust Computational Secret Sharing and a Unified Account of Classical Secret-Sharing Goals. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 172–184, New York, NY, USA, 2007. Association for Computing Machinery.

[23] Sara Belluccini, Rocco De Nicola, Marlon Dumas, Pille Pullonen, Barbara Re, and Francesco Tiezzi. Verification of Privacy-Enhanced Collaborations. In Kyungmin Bae, Domenico Bianculli, Stefania Gnesi, and Nico Plat, editors, *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020*, pages 141–152. ACM, 2020.

[24] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[25] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In Oded Goldreich, editor, *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 351–371. ACM, 2019.

[26] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual*

International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[27] George Robert Blakley. Safeguarding Cryptographic Keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[28] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[29] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: A Tool for Cryptographically Secure Statistical Analysis. *IEEE Trans. Dependable Secur. Comput.*, 15(3):481–495, 2018.

[30] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From Input Private to Universally Composable Secure Multi-party Computation Primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 184–198. IEEE Computer Society, 2014.

[31] Dan Bogdanov, Sven Laur, and Riivo Talviste. A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, volume 8788 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2014.

[32] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[33] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[34] Florian Böhl and Dominique Unruh. Symbolic universal composability. *J. Comput. Secur.*, 24(1):1–38, 2016.

[35] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2005.

[36] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory*

and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367. Springer, 2015.

[37] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[38] David Butler, David Aspinall, and Adrià Gascón. How to Simulate It in Isabelle: Towards Formal Proof for Secure Multi-Party Computation. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2017.

[39] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. iUC: Flexible Universal Composability Made Simple. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 191–221. Springer, 2019.

[40] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.*, 13(1):143–202, 2000.

[41] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `https://eprint.iacr.org/2000/067`.

[42] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[43] Ran Canetti. Obtaining Universally Composable Security: Towards the Bare Bones of Trust. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 88–112. Springer, 2007.

[44] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA,*

USA, August 16-20, 2015, Proceedings, Part II, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[45] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively Secure Multi-Party Computation. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648. ACM, 1996.

[46] Ran Canetti, Rafael Pass, and Abhi Shelat. Cryptography from Sunspots: How to Use an Imperfect Reference String. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 249–259. IEEE Computer Society, 2007.

[47] Ran Canetti and Tal Rabin. Universal Composition with Joint State. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2003.

[48] Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 167–183. IEEE, 2019.

[49] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[50] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19. ACM, 1988.

[51] Ran Cohen and Yehuda Lindell. Fairness Versus Guaranteed Output Delivery in Secure Multiparty Computation. *J. Cryptol.*, 30(4):1157–1186, 2017.

[52] Geoffroy Couteau. New Protocols for Secure Equality Test and Comparison. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 303–320. Springer, 2018.

[53] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General Secure Multiparty Computation from any Linear Secret-Sharing Scheme. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques,*

Bruges, Belgium, May 14-18, 2000, Proceeding, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.

[54] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[55] Ivan Damgård and Jesper Buus Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[56] Ivan Damgård and Jesper Buus Nielsen. Adaptive versus static security in the UC model. In *International Conference on Provable Security*, pages 10–28. Springer, 2014.

[57] Ivan Damgård and Claudio Orlandi. Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

[58] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 799–829. Springer, 2018.

[59] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[60] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[61] Yevgeniy Dodis and Silvio Micali. Parallel Reducibility for Information-Theoretically Secure Computation. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2000.

[62] Marlon Dumas, Luciano García-Bañuelos, Joosep Jääger, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen-Raudvere, Aivo Toots, Reedik Tuuling, and Maksym Yerokhin. Multi-Level Privacy Analysis of Business Processes: the Pleak Toolset. *International Journal on Software Tools for Technology Transfer*, 24(2):183–203, 2022.

[63] Cynthia Dwork. Differential Privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2006.

[64] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *1st Conference on Information-Theoretic Cryptography, ITC 2020, June 17-19, 2020, Boston, MA, USA*, volume 163 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[65] Fabienne Eigner, Matteo Maffei, Ivan Pryvalov, Francesca Pampaloni, and Aniket Kate. Differentially private data aggregation with optimal utility. In Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 316–325. ACM, 2014.

[66] Karim Eldefrawy and Vitor Pereira. A High-Assurance Evaluator for Machine-Checked Secure Multiparty Computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 851–868. ACM, 2019.

[67] Daniel Escudero. An Introduction to Secret-Sharing-Based Secure Multiparty Computation. Cryptology ePrint Archive, Report 2022/062, 2022.

[68] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proc. ACM Program. Lang.*, 7(POPL):866–892, 2023.

[69] Simson L. Garfinkel. De-identification of Personal Information:, 2015-10-01 04:10:00 2015.

[70] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 495–504. ACM, 2014.

[71] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 101–111. ACM, 1998.

[72] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009. AAI3382729.

[73] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, USA, 2009. ACM.

[74] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194. ACM, 1987.

[75] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[76] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In Oded Goldreich, editor, *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. ACM, 2019.

[77] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-Aided Proofs for Multiparty Computation with Active Security. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 119–131. IEEE Computer Society, 2018.

[78] Iftach Haitner, Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. Black-Box Constructions of Protocols for Secure Computation. *SIAM J. Comput.*, 40(2):225–266, 2011.

[79] Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Actively Secure Garbled Circuits with Constant Communication Overhead

in the Plain Model. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 3–39. Springer, 2017.

[80] Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. The Price of Active Security in Cryptographic Protocols. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 184–215. Springer, 2020.

[81] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462. ACM, 2010.

[82] Martin Hirt and Ueli Maurer. Player Simulation and General Adversary Structures in Perfect Multiparty Computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[83] Martin Hirt and Ueli M. Maurer. Complete Characterization of Adversaries Tolerable in Secure Multi-Party Computation (Extended Abstract). In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 25–34. ACM, 1997.

[84] Dennis Hofheinz and Victor Shoup. GNUC: A New Universal Composability Framework. *J. Cryptology*, 28(3):423–508, 2015.

[85] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 772–783. ACM, 2012.

[86] 754-2008 - IEEE Standard for Floating-Point Arithmetic. `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`, 2008.

[87] Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec.*, 14(6):531–548, 2015.

[88] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

[89] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189. Springer, 2018.

[90] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *J. Comput. Secur.*, 21(2):283–315, 2013.

[91] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[92] Ben Kreuter, Abhi Shelat, Benjamin Mood, and Kevin R. B. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 321–336. USENIX Association, 2013.

[93] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-Theoretically Secure Protocols and Security under Composition. *SIAM J. Comput.*, 39(5):2090–2112, 2010.

[94] Ralf Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 309–320. IEEE Computer Society, 2006.

[95] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. The IITM Model: A Simple and Expressive Model for Universal Composability. *J. Cryptol.*, 33(4):1461–1584, 2020.

[96] Sven Laur and Pille Pullonen-Raudvere. Foundations of Programmable Secure Computation. *Cryptography*, 5(3):22, 2021.

[97] Kevin Liao, Matthew A. Hammer, and Andrew Miller. ILC: a calculus for composable, computational cryptography. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 640–654. ACM, 2019.

[98] Yehuda Lindell. General Composition and Universal Composability in Secure Multiparty Computation. *J. Cryptol.*, 22(3):395–428, 2009.

[99] Yehuda Lindell. How to Simulate It - A Tutorial on the Simulation Proof Technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[100] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.

[101] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[102] Yehuda Lindell and Benny Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[103] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating Efficient RAM-Model Secure Computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638. IEEE Computer Society, 2014.

[104] Andreas Lochbihler, S. Reza Sefidgar, David A. Basin, and Ueli Maurer. Formalizing Constructive Cryptography using CryptHOL. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 152–166. IEEE, 2019.

[105] Ueli Maurer. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.

[106] Ueli Maurer and Renato Renner. Abstract Cryptography. In Bernard Chazelle, editor, *Innovations in Computer Science - ICS 2011, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 1–21. Tsinghua University Press, 2011.

[107] Lúcás Críostóir Meier. Towards Modular Foundations for Protocol Security. Cryptology ePrint Archive, Paper 2023/187, 2023.

[108] Silvio Micali and Phillip Rogaway. Secure computation. In *Annual International Cryptology Conference*, pages 392–404. Springer, 1991.

[109] Payman Mohassel and Peter Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In David Lie, Mohammad Mannan, Michael

Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 35–52. ACM, 2018.

[110] Payman Mohassel and Ben Riva. Garbled Circuits Checking Garbled Circuits: More Efficient and Secure Two-Party Computation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53. Springer Berlin Heidelberg, 2013.

[111] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In Omer Reingold, editor, *Theory of Cryptography*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer Berlin Heidelberg, 2009.

[112] Yair Oren. On the Cunning Power of Cheating Verifiers: Some Observations about Zero Knowledge Proofs (Extended Abstract). In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 462–471. IEEE Computer Society, 1987.

[113] Emmanuela Orsini. Efficient, actively secure MPC with a dishonest majority: a survey. In *Arithmetic of Finite Fields: 8th International Workshop, WAIFI 2020, Rennes, France, July 6–8, 2020, Revised Selected and Invited Papers 8*, pages 42–71. Springer, 2021.

[114] Rafail Ostrovsky and Moti Yung. How to Withstand Mobile Virus Attacks (Extended Abstract). In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 51–59. ACM, 1991.

[115] Pascal Paillier. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[116] Rafael Pass. Bounded-Concurrent Secure Multi-Party Computation with a Dishonest Majority. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 232–241, New York, NY, USA, 2004. Association for Computing Machinery.

[117] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2165–2182. USENIX Association, 2021.

[118] Martin Pettai and Peeter Laud. Automatic Proofs of Privacy of Secure Multi-party Computation Protocols against Active Adversaries. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE 28th Com-*

puter Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015, pages 75–89. IEEE Computer Society, 2015.

[119] Martin Pettai and Peeter Laud. Combining Differential Privacy and Secure Multiparty Computation. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 421–430. ACM, 2015.

[120] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Cryptographic Security of Reactive Systems. In Steve A. Schneider and Peter Ryan, editors, *Workshop on Secure Architectures and Information Flow 1999, Royal Holloway, London, UK, December 1-3, 1999*, volume 32 of *Electronic Notes in Theoretical Computer Science*, pages 59–77. Elsevier, 1999.

[121] Birgit Pfitzmann and Michael Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000*, pages 245–254. ACM, 2000.

[122] Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 184–, Washington, DC, USA, 2001. IEEE Computer Society.

[123] Manoj Prabhakaran and Mike Rosulek. Cryptographic Complexity of Multi-Party Computation Problems: Classifications and Separations. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2008.

[124] Pille Pullonen, Raimundas Matulevicius, and Dan Bogdanov. PE-BPMN: Privacy-Enhanced Business Process Model and Notation. In Josep Carmona, Gregor Engels, and Akhil Kumar, editors, *Business Process Management - 15th International Conference, BPM 2017, Barcelona, Spain, September 10-15, 2017, Proceedings*, volume 10445 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2017.

[125] Pille Pullonen and Sander Siim. Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2015.

[126] Pille Pullonen, Jake Tom, Raimundas Matulevicius, and Aivo Toots. Privacy-enhanced BPMN: enabling data privacy analysis in business processes models. *Software and Systems Modeling*, 18(6):3235–3264, 2019.

[127] Tal Rabin and Michael Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85. ACM, 1989.

[128] Dragos Rotaru and Tim Wood. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology - INDOCRYPT 2019 - 20th International Conference on Cryptology in India, Hyderabad, India, December 15-18, 2019, Proceedings*, volume 11898 of *Lecture Notes in Computer Science*, pages 227–249. Springer, 2019.

[129] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

[130] Alley Stoughton and Mayank Varia. Mechanizing the Proof of Adaptive, Information-Theoretic Security of Cryptographic Protocols in the Random Oracle Model. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 83–99. IEEE Computer Society, 2017.

[131] Latanya Sweeney. k-Anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.*, 10(5):557–570, 2002.

[132] Tomas Toft. Solving Linear Programs Using Multiparty Computation. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2009.

[133] Aivo Toots, Reedik Tuuling, Maksym Yerokhin, Marlon Dumas, Luciano García-Bañuelos, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen, and Jake Tom. Business Process Privacy Analysis in Pleak. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 306–312. Springer, 2019.

[134] Aivo Toots, Reedik Tuuling, Maksym Yerokhin, Marlon Dumas, Luciano García-Bañuelos, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen, and Jake Tom. Business Process Privacy Analysis in Pleak - (Extended Abstract). *Informatik Spektrum*, 42(5):354–355, 2019.

[135] Thijs Veugen and Mark Abspoel. Secure integer division with a private divisor. *Proc. Priv. Enhancing Technol.*, 2021(4):339–349, 2021.

[136] Douglas Wikström. Simplified Universal Composability Framework. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 566–595. Springer, 2016.

[137] Andrew C. Yao. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

ACKNOWLEDGEMENT

First and foremost, many thanks to my supervisors Sven Laur and Dan Bogdanov. Sven was there to initially make me think about the boxes and formalisations and then again to make me see out of the box and look at things with a new perspective. Many thanks for the endless times we have spent in front of a whiteboard trying to understand if our ideas match. Sometimes they did and sometimes they did not. Sometimes I was right and sometimes I was not - both were equally educational while the first was probably more satisfying. I hope that I never forget that there is always some more detail, aspect or curious case to consider. Dan made sure there was enough motivation and growth in areas other than the boxes and formal definitions. Thank you for some of the diversions during this endless thesis work. Thank you for believing in me and my ability to take the lead. It sure was easier to just do the things I was told but finding myself in different roles in different projects helped me grow and understand the world where the scientific process lives in. All in all helping me to truly understand that this is what I want to do.

Many thanks to my opponents Mayank Varia and Bernardo David. I am sure it required considerable effort to work through everything. Thank you for your time and valuable feedback. Reading you found the contributions of this thesis relevant and interesting was highly encouraging. You both pointed out various inaccuracies, as well as gave ideas of new aspects that could be included to give more context and depth to the results and ideas of this thesis. Also, many thanks to Dominique Unruh, the internal reviewer from the University of Tartu. You helped iron out numerous inconsistencies and drew attention to several omitted details.

Many thanks to my friends and colleagues in Cybernetica. To Sander, Aivo, Maria, Reedik and Sebastian who survived me while I figured out what their tasks and roles should be and what my tasks as someone leading a team entail. We did survive, maybe we sometimes even thrived. To everyone in the NAPLES project. To Alisa, Peeter and Liina for the feedback on this manuscript and all the projects we have done together. To everyone working with Sharemind, ensuring my theories have real-world applications. To Jaak Ra and Ri for not telling me how bad my C++ code was. I was kind of afraid of that in the beginning. To Baldur for your enthusiasm about everything, including Pleak.

Last but not least, to my family and friends. Many thanks for sometimes asking how my studies are going but not asking too much or too often. For seeing that this journey is complicated on more levels than just science. To Uku, for gently and occasionally not so gently pushing me towards this goal. And to Paul, for making me see that this is not the most important thing.

SISUKOKKUVÕTE

Turvalise ühisarvutuse kiire ja turvalise algoritmiarenduse alused

Turvaline ühisarvutus on meetod, kuidas kasutada erinevate osapoolte privaatseid andmeid nii, et neist sisendite privaatsust säilitades saada ühiseid tulemusi. Intuitiivselt tagab turvalisus seda, et sisendite kohta ei leki muud kui planeeritud arvutuse tulemus ning arvutamise meetod on alati korrektne. Turvalise ühisarvutuse jaoks on olemas erinevaid krüptograafilisi protokolle, mis võimaldavad kas konkreetseid arvutusi teha või defineerivad meetodi kuidas arvutada kõikvõimalikke algoritme. See doktoritöö keskendub just sellele viimasele juhule, mida me võime nimetada ka programmeeritavaks turvaliseks ühisarvutuseks. Erinevad osapooled saavad anda oma privaatseid sisendeid ja ühiselt defineerida algoritmi, mida siis nende sisendite peal rakendatakse. On võimalik ka, et järgmised arvutussammud sõltuvad vahepeal saadud tulemustest.

Kui teoreetiliselt on võimalik teha kõiki arvutusi, on kriitiline õigesti mõtestada, mida turvalisus selles kontekstis tähendab. Alati on vaja läbi mõelda, kas see väljund, mis arvutustest tuleb, on ikkagi see, mida soovitakse ja mis on lubatud. Samuti on vaja tagada, et arvutuse protsessi käigus ei lekiks rohkem informatsiooni kui see väljund annab. Just teine küsimus algoritmi leketest on käesoleva töö fookuses.

Klassikaline meetod algoritmi turvalisuse tõestamisel põhineb sellel, et algoritmi tööd on võimalik simuleerida teades vaid korrumpeeritud osapooltele kättesaadavaid sisendeid ja väljundeid. Loogika on, et kui algoritmi käiku saab sedasi jäljendada, siis ei saa need osapooled saada algoritmi jooksutamise ajal rohkem informatsiooni kui neile niikuinii on arvutuse käigus ette nähtud. Formaalselt defineeritakse ideaalne funktsionaalsus, mis kujutab endast algoritmi turvadefinitsiooni. Käesolev töö defineerib turvalise ühisarvutuse jaoks kanoonilise ideaalse funktsionaalsuse, mis kogub kokku kõik sisendid, arvutab soovitud funktsiooni ning tagastab väljundi vastavatele osapooltele. Nii sisend- kui väljundandmed võivad olla kuidagi salastatud. Sellisel juhul ideaalne funktsionaalsus kõigepealt eemaldab privaatsuskaitse ning seejärel arvutab tulemuse ning rakendab väljundile ettenähtud privaatsuskaitset. Programmeeritav turvalise ühisarvutuse raamistik koosneb paljudest protokollidest ning erinevatest andmete turvaliselt hoiustamise viisidest. Näiteks võivad andmed olla ühissalastatud või krüpteeritud, neil võib, aga ei pruugi olla terviklikkuse kaitset. Käesolev töö defineerib erinevad omadused, mis turvalise arvutuse raamistikul võivad olla ning seejärel kasutab üldistatud formalisatsiooni, et arutleda algoritmide omaduste üle.

Turvalise arvutamise raamistik ise defineerib üldiselt palju olulisi alusprotokolle, näiteks liitmise ja korrutamise, millest omakorda saab ehitada keerukaid algoritme. Paljud algoritmid on lihtsad ja opereerivad ainult salastatud andmetega ning garanteerivad, et nende käigus midagi ei leki. Teisalt paljud teised algorit-

mid kas annavad avalikke väljundeid või avalikustavad töö käigus vahetulemusi ning seetõttu vajab nende turvalisus detailset analüüsi ka siis kui algoritm ise kasutab turvalisi komponente, et arvutusi teha. See töö vaatleb eraldi kahte algoritmi disainil ettetulevat juhtu. Esiteks seda, et vahel on võimalik suuremas algoritmis kasutada komponente, mis tagavad küll sisendite privaatsuse ent pole turvalised. Teiseks seda, kuidas tõestada algoritmi turvalisust nii, et saaks keskenduda huvitavamatele osadele ning palju formaalseid detaile oleks automaatselt tagatud.

Kanoonilise ideaalse funktsionaalsuse definitsiooni järgi saavad turvalised olla ainult protokollid, mis tagavad selle, et väljund on värskelt salastatud. Samas on tegelikkuses võimalik defineerida protokolle, millel seda omadust ei ole, kuid mis samas oma sisendite kohta midagi ei leki. Käesolev töö nimetab selliseid protokolle sisendi privaatsust säilitavateks protokollideks. Sisendi privaatsust säilitava protokolli väljundid on alati kas privaatsed või ei sõltu protokolli sisenditest. Kui kõik sisendi privaatsust säilitava protokolli väljundid on omakorda sisendid mõne turvalise protokolli jaoks ning komponeeritud protokolli väljundid tulevad kõik turvalisest protokollist, siis on ka komponeeritud protokoll turvaline. Lisaks on mitmest sisendi privaatsust säilitavast protokollist komponeeritud protokoll omakorda ka sisendi privaatsust säilitav. Seetõttu on võimalik sisendi privaatsust säilitavaid protokolle algoritmide arenduses kasutada. See aga on kasulik, sest sellised protokollid võivad olla nii lihtsamad disainida kui ka efektiivsemad kasutada.

Turvalisuse korrektne formaalne tõestamine on üldiselt keerukas ning nõuab paljude detailide läbimõtlemist. Teisalt on paljude algoritmide puhul üsna lihtne sõnastada intuitiivset põhjust, miks nad on turvalised. Näiteks eelnevast kirjeldusest juba läbi käinud idee, et algoritm kasutab ainult salastatud andmeid ilma midagi avalikustamata ning kõik arvutused tehakse protokollidega, mille turvalisus on juba eelnevast teada. Teine suurem osa käesolevast doktoritööst tegelebki sellega, et defineerida abstraktne mudel, milles on võimalik teha intuitsioonile hästi vastavaid tõestusi. Need tõestused keskenduvad just sellele, milliseid avalikustatud andmeid protokolli käigus näha võib olla. Abstraktne mudel ja detailne formaalne mudel protokolli jooksutamisest on samaväärsed paljude turvalise ühisarvutuse raamistike jaoks. Töö defineerib erinevaid omadusi, mis peavad kas raamistiku või protokolli jaoks kehtima, et samaväärsus kehtiks. Seeläbi kaardistab töö ka eeldusi, mis turvalisest ühisarvutusest rääkides sageli implitsiitselt tehakse.

CURRICULUM VITAE

Personal data

Name:	Pille Pullonen-Raudvere
Birth:	January 6th, 1989
Citizenship:	Estonian
Contact:	pille.pullonenraudvere@gmail.com

Education

2013–	University of Tartu, PhD candidate in Computer Science
2011–2013	University of Tartu and Aalto University, Erasmus Mundus Master's programme in Security and Mobile Computing (NordSecMob)
2008–2011	University of Tartu, BSc in Computer Science
2005–2008	Hugo Treffner Gymnasium, secondary education
1996–2005	Tartu Kesklinna school, primary education

Employment

2014–	Cybernetica AS, junior researcher
2012–2014	Cybernetica AS, programmer

Scientific work

Main fields of interest:

- Secure multiparty computation
- Privacy enhancing technologies

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Pille Pullonen-Raudvere
Sünniaeg:	6. jaanuar 1989
Kodakondsus:	Eesti
Kontaktandmed:	pille.pullonenraudvere@gmail.com

Haridus

2013–	Tartu Ülikool, informaatika doktorant
2011–2013	Tartu Ülikool ja Aalto ülikool, Erasmus Munduse magistriprogramm turvalisusest ja mobiilarvutustest (NordSecMob)
2008–2011	Tartu Ülikool, BSc informaatikas
2005–2008	Hugo Treffneri Gümnaasium, keskharidus
1996–2005	Tartu Kesklinna kool, põhiharidus

Teenistuskäik

2014–	Cybernetica AS, nooremteadur
2012–2014	Cybernetica AS, programmeerija

Teadustegevus

Peamised uurimisvaldkonnad:

- Turvaline ühisarvutus
- Privaatsuskaitse tehnoloogiad

LIST OF ORIGINAL PUBLICATIONS

Publications Covered in the Thesis

[30] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From Input Private to Universally Composable Secure Multi-party Computation Primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 184–198. IEEE Computer Society, 2014

[125] Pille Pullonen and Sander Siim. Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2015

[5] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and Performance of Programmable Secure Computation. *IEEE Security & Privacy*, 14(5):48–56, 2016

[96] Sven Laur and Pille Pullonen-Raudvere. Foundations of Programmable Secure Computation. *Cryptography*, 5(3):22, 2021

Other Publications

[124] Pille Pullonen, Raimundas Matulevicius, and Dan Bogdanov. PE-BPMN: Privacy-Enhanced Business Process Model and Notation. In Josep Carmona, Gregor Engels, and Akhil Kumar, editors, *Business Process Management - 15th International Conference, BPM 2017, Barcelona, Spain, September 10-15, 2017, Proceedings*, volume 10445 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2017

[133] Aivo Toots, Reedik Tuuling, Maksym Yerokhin, Marlon Dumas, Luciano García-Bañuelos, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen, and Jake Tom. Business Process Privacy Analysis in Pleak. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 306–312. Springer, 2019

[126] Pille Pullonen, Jake Tom, Raimundas Matulevicius, and Aivo Toots. Privacy-enhanced BPMN: enabling data privacy analysis in business

processes models. *Software and Systems Modeling*, 18(6):3235–3264, 2019

[134] Aivo Toots, Reedik Tuuling, Maksym Yerokhin, Marlon Dumas, Luciano García-Bañuelos, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen, and Jake Tom. Business Process Privacy Analysis in Pleak - (Extended Abstract). *Informatik Spektrum*, 42(5):354–355, 2019

[64] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *1st Conference on Information-Theoretic Cryptography, ITC 2020, June 17-19, 2020, Boston, MA, USA*, volume 163 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020

[23] Sara Belluccini, Rocco De Nicola, Marlon Dumas, Pille Pullonen, Barbara Re, and Francesco Tiezzi. Verification of Privacy-Enhanced Collaborations. In Kyungmin Bae, Domenico Bianculli, Stefania Gnesi, and Nico Plat, editors, *FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020*, pages 141–152. ACM, 2020

[62] Marlon Dumas, Luciano García-Bañuelos, Joosep Jääger, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen-Raudvere, Aivo Toots, Reedik Tuuling, and Maksym Yerokhin. Multi-Level Privacy Analysis of Business Processes: the Pleak Toolset. *International Journal on Software Tools for Technology Transfer*, 24(2):183–203, 2022

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.
41. **Mohammad Anagreh.** Privacy-Preserving Parallel Computations for Graph Problems. Tartu 2023, 181 p.
42. **Rahul Goel.** Mining Social Well-being Using Mobile Data. Tartu 2023, 104 p.

43. **Anti Ingel.** Algorithms using information theory: classification in brain-computer interfaces and characterising reinforcement-learning agents. Tartu 2023, 142 p.
44. **Shakshi Sharma.** Fighting Misinformation in the Digital Age: A Comprehensive Strategy for Characterizing, Identifying, and Mitigating Misinformation on Online Social Media Platforms. Tartu 2023, 158 p.
45. **Kristiina Rahkema.** Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects. Tartu 2023, 182 p.
46. **Ivan Slobozhan.** Studying Online Social Media Engagement in CIS Countries during Protests, Mass Demonstrations and War. Tartu 2023, 81 p.
47. **Nurlan Kerimov.** Building a catalogue of molecular quantitative trait loci to interpret complex trait associations. Tartu 2023, 248 p.
48. **Pavlo Tertychnyi.** Machine Learning Methods for Anti-Money Laundering Monitoring. Tartu 2023, 117 p.
49. **Abasi-amefon Obot Affia.** A Framework and Teaching Approach for IoT Security Risk Management. Tartu 2023, 180 p.
50. **Raimond-Hendrik Tunnel.** Video Game Design and Development Bachelor's Curriculum for Estonia. Tartu 2024, 137 p.
51. **Ahto Salumets.** Bioinformatics analysis of various aspects in immunology. Tartu 2024, 198 p.
52. **Mohammed Abdulhameed Shaif Ali.** Deep Learning Methods for Cell Microscopy Image Analysis. Tartu 2024, 143 p.