UNIVERSITY OF TARTU
FACULTY OF SCIENCE AND TECHNOLOGY
Institute of Computer Science
Computer Science Curriculum

Ville Sokk

An improved type system for a privacy-aware programming language and its practical applications

Master's Thesis (30 ECTS)

Supervisors: Dan Bogdanov, Phd
Jaak Randmets, MSc

Tartu 2016

An improved type system for a privacy-aware programming language and its practical applications

Abstract:
Confidential data needs to be processed in many areas, for example when making policy decisions using goverment databases or when providing cloud-based services. SHAREMIND is a framework for developing privacy-preserving applications which allows data to be analysed without revealing individual values. SHAREMIND uses a technology called secure multi-party computation. Programs using the SHAREMIND framework are written in a programming language called SECREC. SHAREMIND and SECREC are designed to support multiple secure multi-party computation methods which we call protection domain kinds. Different protection domain kinds have different security guarantees and performance characteristics and the decision about which one to use depends on the problem at hand which means SECREC should support different protection domain kinds that solve the needs of different applications. The goal of this thesis is to make it easier to add protection domain kinds to the SECREC language by allowing the programmer to define the protection domain kind data types, arithmetic operations and type conversions in the SECREC language without changing the compiler. The author developed a formal type system for the proposed language extensions, implemented them in the SECREC language compiler, described practical applications, open problems and proposed solutions.

Keywords: secure multi-party computation, domain specific languages, type systems, compiler construction

Täiustatud tüübisüsteem privaatsusteadlikule programmeerimiskeelele ja selle praktilised rakendused

Lühikokkuvõte:
Privaatseid andmeid on tarvis analüüsida või töödelda mitmes valdkonnas, näiteks tehes poliitilisi otsusi kasutades riiklikke andmekogusid või pakkudes pilvepõhiseid teenuseid. SHAREMIND on raamistik turvalisust säilitavate rakenduste arendamiseks, mis võimaldab andmeid analüüsida ilma üksikuid väärtuseid avaldamata. SHAREMIND kasutab selleks turvalise ühisarvutuse tehnoloogiat. SHAREMINDi raamistikku kasutavad programmid on kirjutatud programmeerimiskeeles nimega SECREC. SHAREMIND ja SECREC toetavad erinevaid turvalise ühisarvutuse meetodeid, mida nimetame turvaaladeks. Erinevatel turvaaladel on erinevad turvagarantiid ja efektiivsus ning turvaala valik sõltub konkreetse rakenduse vajadustest, mistõttu peaks SECREC toetama erinevate turvaalade kasutamist vastavalt rakenduse nõuetele. Töö eesmärk on võimaldada SECREC keelele turvaalade lisamist lubades programmeerijal kirjeldada turvaala andmetüübid, aritmeetilised tehted ja tüübiteisendused SECREC keeles. Töö autor lõi keele täiendustele formaalselt kirjeldatud tüübisüsteemi, teostas muudatused SECREC kompilaatoris, kirjeldas muudatuste praktilisi rakendusi, tekkivaid uusi probleeme ja nende võimalikke lahendusi.

Võtmesõnad: turvaline ühisarvutus, valdkonnaspetsiifiline keel, tüübisüsteemid, kompilaatorite ehitus

Contents

1 Introduction

1.1 Problem statement

Confidential data needs to be analysed or processed in many areas. Businesses collect and analyse data from their customers. Data from government institutions can be analysed to make decisions regarding health care, employment and education strategies. Cloud computing services necessarily learn private information about their customers to provide the service. Researchers use patients' data to evaluate and develop medical treatments or learn about the causes of diseases. However, people have a valid concern about what personal information is collected from them and how it is used. We would like to support all these use cases without compromising the privacy of the individual.

The goal of secure multi-party computation (SMC) is to securely evaluate a function on private inputs without revealing the inputs. Multiple cryptographic methods for SMC exist. Sharemind is a platform that implements SMC for programming privacy-preserving applications [Bog13]. A deployment of a Sharemind application uses distributed computers to compute on private values. The distributed programs are implemented in a programming language called SecreC [Jag10, Ris10]. Since different SMC methods have different privacy guarantees and efficiency, Sharemind and SecreC are designed to support multiple SMC methods [BLR14].

The current implementation of SecreC allows the programmer to use different SMC methods but the compiler assumes that every SMC method has private versions of the built-in data types of SecreC (such as integers, booleans and floating point numbers) and supports all arithmetic and logic operations. An implementation of some SMC method may not support all the types and operations. If an unsupported operation is used on some private value, an error will occur when the program is executed on the Sharemind platform. These errors should be excluded statically. Some SMC methods may use non-standard data types, such as big integers which do not have a matching built-in data type. Shamir's secret sharing uses a field with a prime module so private values may require an uncommon bit length (other than 8, 16, 32, 64) for their memory representation [Sha79]. Programmers should be able to extend the set of private data types beyond those built into the language. The goal of this thesis is to extend the SecreC language and compiler implementation so that support for different SMC methods and different private data types can be added dynamically by programming a module in the SecreC language as opposed to changing the compiler. This will reduce mistakes when writing SecreC programs and makes the life of a SecreC programmer easier.

1.2 Outline

This thesis is structured into the following chapters:

- Chapter 2 describes the background of the problem. The chapter defines the terms used in the thesis, describes secure multi-party computation and the SecreC programming language, explains the problem in detail and gives an overview of other programming languages designed for programming privacy-preserving applications.
- Chapter 3 describes the syntax of a subset of the SecreC programming language. This includes a few syntactic constructs added in this thesis and also existing features of the language for completeness.

- Chapter 4 gives formal static semantics to the subset of SecreC that was defined syntactically in chapter 3. The type-level distinction of private and public values is described. We give rules for valid operator definitions and arithmetic expressions as well as type conversion definitions and expressions.
- Chapter 5 describes the implementation of the language extensions proposed by the author in the SecreC language compiler.
- Chapter 6 describes how the changes can affect programming of practical applications in the SecreC language.
- Chapter 7 explains some open issues with both the language design and implementation and proposes solutions.

1.3 Author's contribution

In this section we list the author's original contributions to this thesis. The author designed the language extensions described in this thesis, defined the static semantics of the extensions based on the work of Peeter Laud, Jaak Randmets and Dan Bogdanov [BLR14] and implemented the extensions in the SecreC compiler. The main challenges of this thesis were designing a way to define operations on private values for any secure computing mechanism while requiring minimal effort from the programmer, supporting optimised definitions for special cases, supporting non-standard data types of private values and making the use of PDKs convenient for SecreC programmers by supporting arithmetic expressions with different combinations of input types, such as scalar and vector multiplication.

2 Preliminaries

2.1 Secure computation

Secure multi-party computation (SMC) is a model of computation where n parties compute a function

$$(y_1, y_2, \ldots, y_n) = f(x_1, x_2, \ldots, x_n)$$

where the i-th party provides input x_i and learns output y_i [Yao82]. The parties learn their own output but not each other's inputs or outputs. SMC protocols are designed so that they do not leak anything about the inputs besides what can be deduced from the result. Multiple mature cryptographic methods exist for implementing SMC [ABPP15].

One well-known method of SMC is Yao's garbled circuits [Yao82]. Yao's garbled circuits allow two parties P_1 and P_2 to compute any function on shared inputs. First, the function $f(\cdot, \cdot)$ is represented as a boolean circuit. Party P_1 encrypts ("garbles") the circuit of f. For each bit x of each input, two random values x_0 and x_1 are generated for the two possible values (0 or 1). To garble an OR gate with inputs x, y and output z, a random value corresponding to the output bit is encrypted using the random values corresponding to the inputs as the key. The encrypted outputs are $E_{x_0,y_0}(z_0), E_{x_0,y_1}(z_1), E_{x_1,y_0}(z_1), E_{x_1,y_1}(z_1)$. z_0 or z_1 can only be decrypted with valid inputs. The garbled circuit is sent to P_2 who uses a method called oblivious transfer to learn the random values corresponding to its inputs from P_1. Oblivious transfer does not reveal the value of P_2 to P_1. P_2 now decrypts the random output keys of gates using the learned input keys. After evaluating the circuit, P_2 learns the random values of the output bits. If the different possible output values and the result computed by P_2 are published, the output is revealed by comparing P_2's result to the possible outputs. This method has also been extended to support more than two parties [BMR90, BDNP08].

Another method of SMC is arithmetic secret sharing. To secret share a private value x in ring $\mathbb{Z}_n$, we generate random values $x_1, x_2, \ldots, x_{n-1} \in \mathbb{Z}_n$ and compute

$$x_n = x - x_1 - x_2 - \ldots - x_{n-1} \pmod{n}.$$

The original value can be reconstructed as

$$x = x_1 + x_2 + \ldots + x_n \pmod{n}.$$

Each of the values $x_i, i = 1 \ldots n$ (called shares) are then sent to a different computing party. Since the shares are random, none of the parties knows the original value x. By using network communication and local computations, the parties can compute secret shared results from the shares of the input values. For example, to add two secret shared values x and y, each party adds their shares of the two values:

$$(x_1 + y_1) + (x_2 + y_2) + (x_3 + y_3) = (x_1 + x_2 + x_3) + (y_1 + y_2 + y_3) = x + y$$

Each party now holds a share of the sum. If the parties decide that a result should be made public, they can reveal the shares and add them modulo N. A benefit of secret sharing is that the parties who provide the input shares and who receive the output shares can be independent of the parties computing the function. Extensions of Yao's garbled circuits to multiple parties also have this property.

Different SMC methods have different advantages and disadvantages. They have different performance characteristics, security guarantees and they require a different

number of computing parties. For example, most protocols based on Yao's garbled circuits and arithmetic secret sharing are secure if the computing parties observe the data but do not deviate from the protocol. Some SMC methods provide active security against computing parties who try to learn inputs by not following the protocol [GMW87]. In a practical application, the choice of SMC method depends on a balance of efficiency and security and how many independent parties can participate in the computation. More efficient methods can be used for data that has lower security requirements.

SHAREMIND is a distributed platform for developing secure computation applications [Bog13]. It provides implementations of SMC methods and developer tools for programming privacy preserving applications. The most mature SMC method in SHAREMIND is three-party arithmetic secret sharing. The deployment of a SHAREMIND application using the three-party additive secret sharing method is illustrated in Figure 1. Multiple parties can provide inputs by secret sharing them and sending the shares to the three servers. The servers can reveal outputs to multiple output parties by sending shares of the output to the output party who combines them to learn the result.

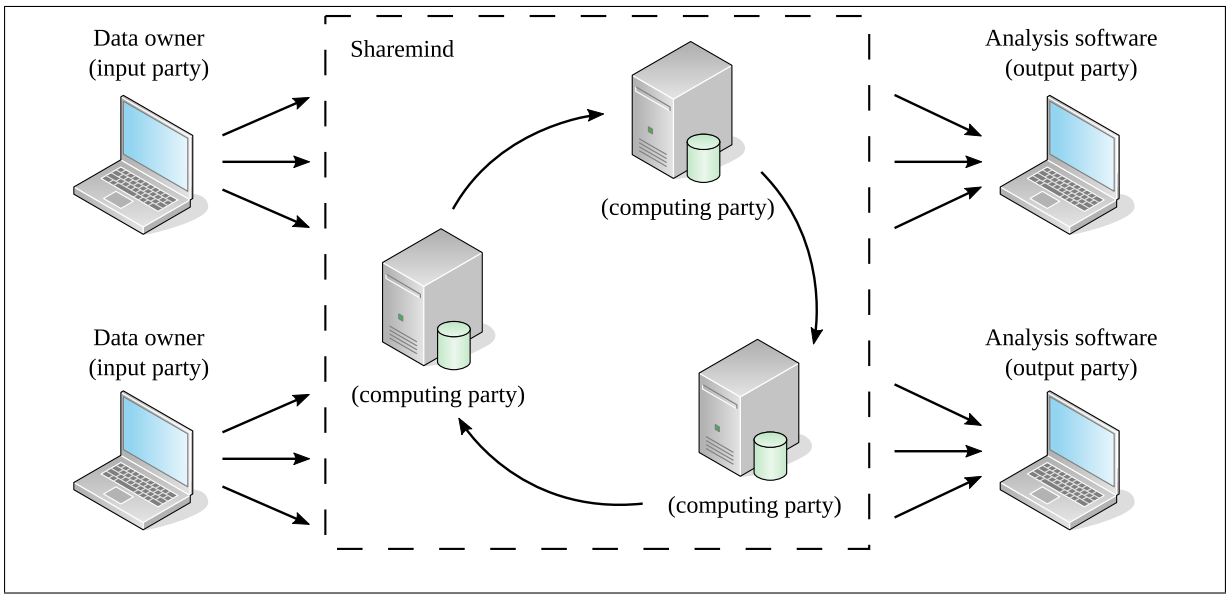

Figure 1: Deployment of a SHAREMIND application

Secure computation has been used in practical applications. SHAREMIND was used in a pilot project to analyse whether working while studying in a university affects graduation rates [BKK+16]. A statistical analysis package called Rmind was used which is implemented on the SHAREMIND platform [BKLS14]. About half of the code of Rmind is written in SECREC. The other half is a client application that runs on the statistician's computer and communicates with the SECREC program. Data from the Estonian Ministry of Education and Research and the Tax and Customs Board was secret shared and uploaded to a deployment of SHAREMIND servers. A statistician analysed the data using Rmind which guaranteed that the statistician could not see individual values in the data. To ensure privacy the servers were hosted by three independent parties (Cybernetica AS, the IT department of the Ministry of Finance and the Estonian Information System Authority). This study illustrates how secure computation, SHAREMIND and SECREC can be used to analyse confidential data without compromising privacy. A study on the full data using conventional methods would not have been possible due to data protection laws.

2.2 The SecreC programming language

To implement a privacy-preserving program using Sharemind, a developer must program two main components. A program that runs on each of the Sharemind servers is written in the SecreC programming language [Jag10, Ris10, BLR14]. The second component is a client application which starts the SecreC program, sends inputs to it and receives results.

SecreC is a procedural statically typed programming language similar to the C and C++ programming languages. Features have been added that make it more suited for writing secure computation programs. A SecreC program is compiled to bytecode which is interpreted by a virtual machine included in the Sharemind server. Two major versions of the language have existed: SecreC 1 [Jag10, Ris10] which supported only a single protection domain and SecreC 2 which has polymorphic procedures and supports multiple protection domains [BLR14].

For security, SecreC relies on an arithmetic black box [DN03]. An arithmetic black box provides protocols for creating and revealing private values and performing secure arithmetic with private values. These protocols are external to the language and SecreC is only used to implement privacy preserving algorithms and business logic using the arithmetic black box as a building block. In the Sharemind framework, protocols can be implemented using a separate domain specific language [LR15].

Since different SMC methods exist, none of which is clearly superior, both Sharemind and SecreC are designed to support multiple SMC methods. We call an implementation of an SMC method a protection domain kind. A protection domain kind (PDK) consists of supported data types, methods for creating (classifying) and revealing (declassifying) private values, arithmetic, relational and logical operations. A protection domain (PD) is an actual instance of a protection domain kind. An example of a PDK would be one using additive secret sharing. It would support classifying values by secret sharing them, declassifying values and protocols for performing arithmetic on secret-shared values. Alice, Bob and Carol can create a protection domain of the additive secret sharing PDK by hosting three Sharemind servers.

A PDK is implemented as a module that is dynamically linked by the Sharemind server. Modules are programmed using a C interface. The Sharemind platform provides a network layer that can be used by the module implementer. The PDK exposes a number of system calls which are procedures with a special interface that can be invoked from SecreC code. For example, to provide private integers, the module should expose system calls for allocating and deleting integer vectors, classifying, declassifying, assigning values, performing arithmetic, comparing values etc. A bytecode program interpreted by the virtual machine can then use these system calls to compute with private integers. The programmer can declare a PDK as follows (shared3p is used in the SecreC standard library as the name of the three party additive secret sharing PDK):

```
kind shared3p;
```

A protection domain can be instantiated as follows:

```
domain pd_shared3p shared3p;
```

In SecreC, the type of each primitive value consists of three components: protection domain, data type and dimensionality. Let us look at an example of a variable declaration:

```
pd_shared3p uint[[1]] x;
```

Here, `pd_shared3p` is the name of the protection domain. For public values there is a protection domain `public` which can be omitted. The `[[1]]` component after the data type is the dimensionality. In this case a one-dimensional vector is declared. Scalars are zero-dimensional arrays. Dimensionality can be omitted when declaring a scalar.

The syntax for operating with private values is the same as for operating with public values. For example, if `x` and `y` are private values, `x * y` is their product and the compiler uses the correct protocol according to the kind of the protection domain of the two variables. In SecreC, arrays with any dimensionality can be operands to an operator. The operator will be applied point-wise, i.e. an expression multiplying two arrays will multiply elements with the same index. A run-time error is reported if the arrays have a different shape.

SecreC uses *information flow control* to track the security class of a value [Mye99]. Protection domains are used as labels indicating the privacy level of the value. These labels form a join-semilattice which is a partially ordered set with an operator called least upper bound. The ordering relation $d_1 \sqsubseteq d_2$ states that values from protection domain d_1 can flow into domain d_2. The least upper bound $d_1 \sqcup d_2$ is the domain where values from both d_1 and d_2 can flow. The type system uses information flow control to track protection domains of values and to statically ensure that private values are not leaked. For example, a private value can not be returned by a procedure with a public return type. It has been shown that if the PDK operations are secure and composable (using multiple operations in a sequence does not reduce security), a program written in SecreC does not leak private values except values which are explicitly declassified [BLR14].

SecreC supports parametric polymorphism using templates similar to the C++ programming language. Templates support domain, data type and dimension quantifiers. If a programmer implements an algorithm that relies on operations that are supported in `shared3p` they can write a templated procedure that supports all protection domains of kind `shared3p` and all data types. For example:

```
template<domain D : shared3p, type T>
D T[[1]] sort(D T[[1]] vector) {
    // Implementation here
}
```

If a templated procedure is called it is instantiated with the concrete types, domains and dimensions used in the call. The domain kind constraint can be omitted (i.e. just `domain D` can be used) if the procedure relies solely on operations that are expected to be implemented in each PDK. SecreC also supports overloading which means a generic procedure can be written that works in different PDKs and a kind-specific optimised version which will be used when the domain used in the call is from that kind. For example, we could implement a PDK-generic sorting procedure and the signature would be the same as the given example except there would be no `shared3p` constraint. These two procedures would both be in scope and when sorting a `shared3p` value, the compiler would choose the procedure with the kind constraint.

The following listing is an example of SecreC code which implements the logistic function and calculates the mean of a private vector:

```
import stdlib;
import shared3p;
```

```
domain pd_shared3p shared3p;

template<domain D : shared3p>
D float64[[1]] logistic(D float64[[1]] x) {
    D float64[[1]] exp(size(x));
    __syscall("shared3p::exp_float64_vec",
              __domainid(D), -x, exp);
    return 1 / (1 + exp);
}

template<domain D>
D float64 mean(D int64[[1]] x) {
    D int64 sum = 0;
    for (uint i = 0; i < size(x); ++i) {
        sum += x[i];
    }
    return (float64) sum / (float64) size(x);
}

void main() {
    pd_shared3p int64[[1]] x = argument("input");
    publish("result", mean(x));
}
```

A vector of private inputs is received from the client application which has executed the SecreC program. The mean of the input vector is calculated using the domain polymorphic procedure `mean`. The result is published to the client application. There is also an example of the logistic function to illustrate the system call feature of SecreC. The `shared3p` PDK protocol for computing the exponential function is used with the `__syscall` syntax to implement `logistic`. This protocol is implemented externally in the module that contains `shared3p` protocols. System calls are usually wrapped in SecreC procedures and are not called directly.

2.3 Problem statement

A protection domain kind in SecreC is currently declared as follows:

```
kind kind_name;
```

A protection domain kind consists of:

- data types supported in the PDK;
- implementations of classification and declassification; and
- implementations of arithmetic, relational and type conversion operators.

The first issue is that the SecreC kind declaration mentioned none of these. Currently the SecreC compiler assumes that all of the primitive types built into the language and all of the operators are supported in each PDK. If, for example, there is a domain `pd_shared3p` of kind `shared3p` and the program contains a multiplication

of two `pd_shared3p uint32[[1]]` vectors, the compiler will use a system call named `shared3p::mul_uint32_vec`. This is a problem because it is possible that the PDK module does not contain this system call. The program will compile without errors and an error occurs when the virtual machine in the SHAREMIND server starts interpreting the program. This should be a compile time error to make programming in SECREC easier.

The second issue is that the set of data types is fixed. Although secret sharing signed and unsigned integers is straightforward, it is not clear what is the best way to approximate real numbers. The additive secret sharing PDK in SHAREMIND supports floating-point numbers similar to the ones used in modern computers [KW14a]. Because operations on this representation of floating-point numbers are expensive compared to integers, other representations have been considered, like fixed-point numbers [KW14b]. Although fixed-point numbers are more efficient, they have a smaller range and are less accurate near zero. It should be possible to define multiple alternative private data types so that the programmer can choose one according to the balance of accuracy and efficiency most suited to the problem at hand.

The third issue is that the representation of the data types in the PDK module is opaque to the SECREC compiler. For example, the compiler does not know how to allocate a private vector because the size of a value is unknown. Allocations are compiled as allocation system calls. In addition to classification, declassification and operators, the implementer of the PDK module must provide system calls for allocating, deleting, initialising, indexing and assigning to vectors. This is a problem because the system calls have a significant overhead. For example, values of a private vector are sometimes rearranged as in the following code:

```
pd_sharedp3 uint[[1]] arrange(pd_shared3p uint[[1]] input,
                              uint[[1]] permutation)
{
    uint length = size(input);
    pd_shared3p uint[[1]] result(length);
    for (uint i = 0; i < length; i++) {
        result[i] = input[permutation[i]];
    }
    return result;
}
```

Even though `input` and `result` are private, this requires no network communication because there is no arithmetic on private values and should be consequently very fast. But the compiled code performs $\mathcal{O}(n)$ system calls and due to the overhead of system calls, it is significantly less efficient than the equivalent code on public values. It should be possible to define the bit width of private types so that the compiler would be able to generate indexing and assignment code without system calls.

The goal of this thesis is to extend the SECREC language and compiler to support:

- data type declarations in kind declarations;
- arithmetic, logic and relational operator definitions; and
- private data type conversion definitions.

For each implemented PDK module, there would be a SECREC library module defining the kind, the data types in the kind and the supported operators and type conversions. A

programmer using SecreC would import the PDK's SecreC library and the compiler would give errors when unsupported types, operations or conversions are used.

Defining a new PDK should be convenient for the implementer of a PDK. SecreC supports operators that have a private and public operand. Currently the public operand is classified to the same domain as the private operand. This means there are up to three versions of a binary operator ($private \otimes private$, $private \otimes public$, $public \otimes private$). Operations where one of the operands is a scalar and the other one has higher dimension are also supported. This also means there are up to three versions of a binary operator ($N \otimes N$, $N \otimes 1$, $1 \otimes N$). If there are n data types and m supported binary operators in a PDK, there would be up to $n \cdot m \cdot 3 \cdot 3$ definitions. This would be too much boilerplate and should be reduced as much as possible. At the same time, the implementer of a PDK should be able to write efficient implementations of special cases.

2.4 Related work

In this section we review other programming languages designed for secure multi-party computation. They are compared to SecreC 2 and their approach to the problems of this thesis is discussed.

One of the distinguishing features of SecreC is its extensive standard library. It includes modules for matrix algebra, oblivious conditionals (on values, not blocks of statements), an implementation of the AES (Advanced Encryption Standard) [Nat01] cipher on secret-shared values, a database system for storing secret-shared values, a module for generating random secret-shared values and shuffling secret-shared vectors, procedures for sorting values, procedures for working with private ASCII strings and different statistical methods such as hypothesis testing procedures and linear regression.

2.4.1 SMCL

SMCL (Secure Multi-party Computation Language) is a procedural programming language designed for writing SMC applications [Nie09].

In SMCL both client and server-side code are written in the same language. The server-side program is compiled to a Java program which uses the SMCR (Secure Multi-party Computation Runtime) Java library. SecreC programs are compiled to bytecode which is executed by the Sharemind servers so multiple server-side programs can be installed without changing the server.

Currently, SecreC programs are provided inputs before the program is executed. Running a SecreC program is equivalent to secure function evaluation. SMCL, on the other hand, is a concurrent programming language where the client and server can communicate while the program is running using channels (called tunnels) or remote procedure calls.

The language is proven to be trace secure – executions of the same program on different private inputs seem the same externally (until a private value is declassified). This is the same guarantee provided by SecreC.

Unlike SecreC, SMCL also allows conditionals with private conditions. The type system checks whether it is possible to securely evaluate a private conditional by tracking whether the branches have side effects like assignment to public variables or I/O.

SMCL provides a statement `open(e|x, y, z)` which reveals the value of expression `e`. The list of variables after the `|` symbol annotates which variables affect the value of `e`.

It is statically tracked that the programmer lists all variables. This system of annotations makes it easier to determine which inputs may leak in the open statement. SecreC does not provide such annotations.

SMCL differentiates private (e.g. `sint`) and public data types (e.g. `int`) but only a single mechanism of SMC is supported so PDKs are not distinguished like in SecreC. The concerns of this thesis do not exist in SMCL but it also does not have the benefits of multiple PDKs.

2.4.2 Fairplay and FairplayMP

Fairplay is a secure multi-party computation system that uses garbled circuits [MNPS04]. It consists of two programs representing the roles of Alice and Bob in Yao's garbled circuits approach. FairplayMP extends the system to more than two computing parties [BDNP08].

The programmer can decide the roles of different parties. Some parties provide inputs, some participate in constructing and evaluating the circuit and some get an output. Different outputs can be provided to different parties.

Compiled Fairplay programs evaluate a boolean circuit described in SHDL (Secure Hardware Definition Language). Programs are written in a high-level procedural language called SFDL (Secure Function Definition Language) which is translated to SHDL.

Like SecreC, a secure program takes inputs once and returns outputs, there is no communication with a separate client while the program is running. SFDL programs do not have public inputs. SFDL supports procedures, conditionals and for loops with a constant number of iterations. Both branches of conditionals are evaluated and the result is chosen obliviously based on the condition. Procedures are always inlined and for loops are unrolled (the body is copied for each iteration). SecreC is more flexible because it allows computations on private and public values. Declassifying aggregate values when it is determined to be semantically safe (i.e. the aggregate value does not leak information about individual inputs that were used to compute it) can be used to write more efficient programs.

Since SFDL only targets boolean circuits, issues related to supporting multiple PDKs are irrelevant.

2.4.3 TASTYL

TASTY is a tool for describing and executing SMC protocols [HKS+10]. It uses a two-party scheme based on Yao's garbled circuits and Paillier's additively homomorphic encryption [Pai99]. The authors show that homomorphic encryption is more efficient than garbled circuits when multiplying integers while garbled circuits are more efficient for comparisons so a system for using both is more efficient than either one alone. They demonstrate this in [HKS+10, Figure 5] by computing the minimum of the point-wise product of two vectors using combined garbled circuits and homomorphic encryption. TASTY includes algorithms for converting values between the two schemes. The TASTY framework can generate Yao circuits for arithmetic operations or read them from files (including the SHDL format used by Fairplay).

A language called TASTYL is used to write programs using the TASTY tool. TASTYL is implemented as a Python domain specific language (DSL). It supports arithmetic and comparisons on private (both garbled and encrypted) and public values. The two parties are given names (e.g. "client" and "server") and variables must be explicitly declared

on either side. The programmer has to declare whether a value is garbled or encrypted using homomorphic encryption.

TASTYL is not as high-level as SMCL, SFDL from Fairplay or SecreC because the programmer has to think about the two computation parties instead of describing a high-level algorithm. Unlike SFDL, it supports public values and unbounded iteration on public values. Since TASTYL is designed specifically for the combination of garbled circuits and homomorphic encryption, it does not support multiple PDKs and the issues of this thesis are irrelevant.

2.4.4 L1

L1 is a procedural language for implementing SMC protocols [SKM11]. It supports procedures, arithmetic, conditionals, loops, big integers, native integers and booleans. Programs written in L1 are compiled into multiple Java programs which implement the different parties of the SMC protocol. Multiple SMC techniques are supported, such as secret sharing, homomorphic encryption and Yao's garbled circuits. Different parties of the computation are given numeric identifiers and the programmer can write sections of code which are only executed by one party. The parties can send messages to each other through TCP/IP channels. Special syntax allows for loops to be executed in parallel.

L1 enables implementation of efficient protocols by using both public and private computations, by combining multiple SMC techniques and by using explicit communication. Functions can be implemented externally as a Java library which allows L1 to be extended with new SMC mechanisms although extensions such as homomorphic encryption are used directly as opposed to implementing an arithmetic black box as in SecreC.

Since L1 requires the programmer to understand the cryptographic primitives of SMC and how to use them securely it is a relatively low-level language. It is more useful for protocol designers as opposed to programmers implementing high-level algorithms or business logic which is what SecreC is designed for.

Multiple SMC mechanisms are supported by L1 which is the topic of this thesis but it is not directly comparable to SecreC because SecreC relies on an arithmetic black box but L1 requires the programmer to explicitly use the underlying cryptographic methods such as encryption. While the designers of L1 are concerned with supporting different cryptographic primitives, SecreC is designed to support different implementations of a PDK and the programmer is not required to understand cryptography.

2.4.5 DSL of Launchbury et al.

Launchbury et al. created a Haskell library and embedded DSL (EDSL) for developing SMC programs [LDDAM12]. The library uses three party additive secret sharing for numbers and XOR sharing for bit strings. To share an integer value x using XOR sharing, shares x_1, x_2, x_3 are generated such that $x_1 \oplus x_2 \oplus x_3 = x$ where $\oplus$ is the bitwise exclusive or operator.

Their EDSL supports addition, subtraction, multiplication and logical operators on shares and vectors of shares. The three computing parties are connected in a circle. The language provides an operation for sending a value to the next neighbour and receiving a value from the previous neighbour. Using network communication, generating randomness and computations with shares, the programmer can write protocols operating on secret shared values.

Multiple program transformations are implemented which optimise the EDSL program. Operations of the same type which do not depend on each other are grouped, i.e. a single operation is applied to vectors of values point-wise. This reduces latency when the SMC protocol requires multiple rounds of communication. For example, if multiplication requires r rounds and the program contains m separate independent multiplications, there will be $r \cdot m$ total rounds. When the operations are grouped, there will be r rounds. The optimiser can also unroll loops which may allow independent arithmetic operations in different loop iterations to be grouped. The execution of an SMC program alternates between local computations and network communication. To further reduce network latency, the EDSL allows multiple instances of a protocol to be executed concurrently which allows computation and network communication of the protocol instances to overlap.

The EDSL has a different goal than SecreC because the programmer works with shares directly and communicates explicitly with the other parties while SecreC uses an arithmetic black box. It is more suitable for programming arithmetic protocols or combining protocols and logic in the same program. Although this language does not support multiple SMC mechanisms in the same program (except arithmetic and XOR sharing which are used for different types of values and are not completely interchangeable), the issues addressed in this thesis do not apply.

2.4.6 DSL of Mitchell et al.

Mitchell et al. describe a language for writing SMC programs with formal static and dynamic semantics and a proof of security [MSSZ12]. The language is implemented as a Haskell EDSL and there is also a front-end compiler from specialised syntax to the EDSL. Private and public conditionals, functions, mutable state and recursion are supported.

The language is defined in a generic manner assuming the existence of a secure execution platform which provides protocols for classifying and declassifying values and performing arithmetic on private values. Thus multiple SMC methods that fit their formal definition of a secure execution platform are supported. They show that fully homomorphic encryption and Shamir secret sharing [Sha79] are secure execution platforms.

The semantics are described abstractly in terms of some set of primitive data types (e.g. booleans, integers) and some set of operations for working with values. The type system uses information-flow to track the privacy level of values. The formal dynamic and static semantics have been used to prove that the language is trace secure if the underlying secure execution platform is secure.

Although the concept of a secure execution platform is similar to the PDK definition and the language is thus PDK agnostic, multiple different PDKs can not be used in the same program which is supported in SecreC. There is also no description of whether and how the programmer can add secure execution platforms without changing the compiler.

2.4.7 PICCO

PICCO is an extension of the C language that is compiled into privacy-preserving C programs that use Shamir's secret sharing [ZSB13, ZBA15].

The generated C program is compiled and executed by the computing parties. PICCO adds private values and conditionals to C. Due to performance concerns, the user can specify the size of the primitive data types. For example, `private int<20> x;` declares a private 20-bit integer variable. A large number of operators on private values have been implemented, including division and bit-level operators such as shifting and bitwise logic.

As in SecreC, operations on arrays are supported which are applied point-wise. The compiler disallows private values in contexts where public values are expected. PICCO supports parallel for loops where the body of the loop is executed in parallel with different loop indices. Statements outside of loops can also be executed in parallel by enclosing them in square brackets.

Inputs are provided before the program is run by input parties and outputs can be sent to different output parties. Clients do not communicate with the program while it is running.

Since PICCO uses only Shamir's secret sharing, the issues addressed in this thesis are not relevant.

2.4.8 Wysteria

Wysteria is a language for writing SMC programs using boolean circuits and secret sharing [RHH14]. It is a functional programming language with an advanced type system and formally defined semantics that allows both private and public computations.

The parties (called principals) participating in the computation are first class values. When binding the value of an expression to a variable, the mode of computation has to be specified. There are two modes – parallel and secure. Parallel mode is used for computations that are performed independently by the different parties, i.e. computations on public values. Secure mode is used for distributed secure multi-party computations on private values. When declaring the mode of an expression, the set of principals involved in the computation must be declared. For example, if principal Alice has input `x1` then

```
let x2 =par({Alice})= x1 * 2
```

binds `x1 * 2` to variable `x2` on Alice's machine. Special values called wires are used for private values in secure mode computations. A wire with type `W {Alice} nat` contains a natural number owned by Alice. It can be used in an expression with mode `sec(`*ps*`)` if Alice is in the set *ps*. Values from multiple principals can be included in a wire which is then called a wire bundle. For example, the following function solves the millionaires problem [RHH14]:

```
is_richer = λv : W {Alice, Bob} nat.
  let out =sec({Alice, Bob})= v[Alice] > v[Bob] in
  out
```

The input `v` is a wire bundle containing the values of both Alice and Bob. The secure computation that binds variable `out` securely compares the private value of Alice and Bob which can be retrieved from the bundle using the square bracket syntax. Functions also have a mode depending on the returned value. The mode of `is_richer` is `sec({Alice, Bob})` which means it can only be called in expressions where both Alice and Bob participate.

Principals and principal sets are first class values. For example, the following is the signature of a function that computes the smallest element in a list where each element is provided by a different principal.

```
min : (all : ps) → W all nat → W all ps(ν ⊆ all ∧ single ν)
```

The first parameter is the set of principals (`ps` is the type of a principals set). The second parameter is the wire bundle containing the input of each principal. The returned value is a wire bundle of principal sets. Wysteria has refinement typing and the type

`ps(`$\nu \subseteq$ `all` $\wedge$ `single` ν`)` means that the principal set returned to all principals is a single element subset of the input principals set, i.e. each principal will receive the principal who has the minimum element. Since Wysteria is designed specifically for garbled circuits and secret sharing, it is not concerned with supporting multiple PDKs so the issues of this thesis are not relevant.

2.4.9 ObliVM-lang

ObliVM is a framework for secure computation [LWN+15]. ObliVM programs use an implementation of Yao's garbled circuits called ObliVM-GC which is written in Java. Programs are written in a programming language called ObliVM-lang which is similar to the C and C++ programming languages.

ObliVM distinguishes public and private values, supports private conditionals, integers with arbitrary bit length (like PICCO), polymorphism using templates (like SecreC), higher-order procedures (procedures that take other procedures as arguments) and oblivious RAM (ORAM). ORAM allows vectors to be indexed with private values without leaking the index. This is useful for programming secure implementations of common data structures and algorithms. The authors show implementations of Dijkstra's shortest path algorithm and an oblivious stack data structure.

ObliVM-lang supports phantom procedures which can be called in private conditionals. Normally, a procedure call in a private conditional can be unsafe if it writes to a public memory location because that would leak the condition. ObliVM always executes procedure calls in both branches. For example, if the condition is false and the true branch contains a procedure call, all writes to a variable in the procedure will obliviously re-write the current value of the variable which leaves the state unchanged but produces the same execution trace as if the procedure was actually called.

ObliVM-lang supports user defined types. The following example is from the ObliVM article. Suppose that there is a Java class called `BigInteger` implementing an alternative to the built-in arbitrary precision integers. This class can be used with the following declaration [LWN+15]:

```
typedef BigInt@m = native BigInteger;
```

The `@m` variable is called a *generic constant* and designates the number of bits. Methods can be defined on ObliVM-lang types like this [LWN+15]:

```
BigInt@m BigInt@m.add(BigInt@m x, BigInt@m y)
    = native BigInteger.add;
BigInt@m BigInt@m.multiply(BigInt@m x, BigInt@m y)
    = native BigInteger.multiply;
BigInt@m BigInt@m.fromInt(int@m y)
    = native BigInteger.fromInt;
int@m BigInt@m.toInt(BigInt@m y)
    = native BigInteger.toInt;
```

These are similar to procedures in SecreC which use system calls to invoke external code. ObliVM-lang thus also allows adding data types and alternative SMC methods by programming protocols in Java. But ObliVM-lang is tailored to Yao's garbled circuits and the built-in types and operators will still use garbled circuits. While methods such as these can be added to external types, operators can not be overloaded as is proposed in this thesis.

3 Syntax

To describe the changes to the SecreC programming language, we will use a simplified version of the language. We will describe the syntax and then give typing rules for terms in this simplified language. We will use Backus Naur Form with a few extensions to describe the syntax although the syntax is abstract, not concrete. The `*` operator is used for repetition (zero or more occurences), `+` is used for repetition with at least one occurence and `?` means that the item is optional.

The real SecreC programming language supports modules and imports. Most modules do not define PDKs, operators and casts. It is assumed that a module with the kind, operator and cast definitions is programmed for each protection domain kind. This module can be imported by the programmer who wishes to use the PDK.

In this thesis, we add syntax for protection domain kind definitions and cast definitions. The operator definition syntax is already in the SecreC language for operator overloading. The rest of the syntactical constructs are included for completeness or because they are affected by the features added in this thesis.

The syntax of a program is given in Figure 2. A program consists of PDK definitions, domain declaration, procedure definitions, operator definitions and cast definitions. The procedure named "main" is executed when the program runs.

$$
\begin{array}{rcl}
\langle \texttt{top\_level} \rangle & ::= & \langle \texttt{kind\_definition} \rangle \\
& | & \langle \texttt{domain\_declaration} \rangle \\
& | & \langle \texttt{procedure\_definition} \rangle \\
& | & \langle \texttt{operator\_definition} \rangle \\
& | & \langle \texttt{cast\_definition} \rangle \\
\langle \texttt{program} \rangle & ::= & \langle \texttt{top\_level} \rangle^{+}
\end{array}
$$

Figure 2: SecreC program grammar

The syntax of basic expressions and statements is given in Figure 3. The definitions of integer and float literals and identifiers are omitted. Identifiers consist of alphanumeric characters or underscores and do not start with a number. The set of operators and built-in data types is minimal in this simplified language.

Using this simplified syntax, multi-dimensional arrays are relatively useless because there is no indexing syntax. We can assume that there are built-in procedures for indexing and assigning elements of vectors.

Note that the block statement rule only allows a single variable declaration in the beginning of a block. This is a simplification and a C-style block {`t1 x1; ...; tn xn; s`} can be re-written as {`t1 x1;` {`t2 x2;` {`...`{`tn xn; s`} `...`}.

The value of a list of expressions between braces is a multi-dimensional array. For example, a matrix can be written as {{`1, 2`}, {`3, 4`}}.

The rule `'('` ⟨`data_type`⟩ `')'` ⟨`expression`⟩ is used for type conversions. For example, `(int) x` converts the value of the variable `x` to an integer.

The `declassify(e)` expression is used to declassify a value. The value is made public and revealed to all the servers. Before a SecreC program is installed on the Sharemind servers, the `declassify` expressions must be inspected to make sure they do not reveal too much information. This expression can be used to implement efficient algorithms when some information can be leaked. For example, in statistics we often compute with

```
⟨statement⟩ ::= ⟨identifier⟩ ⟨assign_operator⟩ ⟨expression⟩ ';'
              | 'while' '(' ⟨expression⟩ ')' ⟨statement⟩
              | 'if' '(' ⟨expression⟩ ')' ⟨statement⟩
                'else' ⟨statement⟩
              | 'return' ⟨expression⟩ ';'
              | ⟨expression⟩ ';'
              | ⟨statement⟩ ⟨statement⟩
              | ⟨block_statement⟩
⟨block_statement⟩ ::= '{' ⟨type⟩ ⟨identifier⟩ ';'
                      ⟨statement⟩ '}'
⟨expression⟩ ::= ⟨expression⟩ ⟨binary_operator⟩ ⟨expression⟩
               | ⟨unary_operator⟩ ⟨expression⟩
               | ⟨expression⟩ ⟨incdec⟩
               | 'declassify' '(' ⟨expression⟩ ')'
               | ⟨identifier⟩
               | ⟨integer_literal⟩ | ⟨float_literal⟩
               | 'true' | 'false'
               | '{' ⟨expression_list⟩ '}'
               | '(' ⟨data_type⟩ ')' ⟨expression⟩
               | ⟨identifier⟩ '(' ⟨expression_list⟩ ')'
⟨expression_list⟩ ::= (⟨expression⟩ (',' ⟨expression⟩)*)?
⟨assign_operator⟩ ::= '=' | '*=' | '+=' | '&='
⟨incdec⟩ ::= '++' | '--'
⟨binary_operator⟩ ::= '*' | '+' | '&' | '==' | '<='
⟨unary_operator⟩ ::= '!' | '-'
⟨type⟩ ::= ⟨identifier⟩? ⟨data_type⟩ ⟨dimension_type⟩?
⟨data_type⟩ ::= ⟨identifier⟩ | ⟨primitive_data_type⟩
⟨dimension_type⟩ ::= '[[' ⟨integer_literal⟩ ']]'
⟨primitive_data_type⟩ ::= 'bool' | 'int' | 'float'
```

Figure 3: Statement and expression grammar

filtered data. Patients in medical studies are separated into case and control groups. Before calculating a statistic on filtered data we can reduce the data to just the values that satisfy the filter which leakes the number of elements where the filter is true but not which values satisfy the filter [BKLS14]. Leaking this number is generally not an issue.

```
⟨procedure_definition⟩ ::= ⟨template⟩? ⟨return_type⟩ ⟨identifier⟩ '('
                           ⟨procedure_parameter⟩?
                           (',' ⟨procedure_parameter⟩)* ')'
                           ⟨block_statement⟩
        ⟨return_type⟩ ::= 'void' | ⟨type⟩
           ⟨template⟩ ::= 'template' '<'
                           ⟨quantifier⟩ (',' ⟨quantifier⟩)* '>'
         ⟨quantifier⟩ ::= 'domain' ⟨identifier⟩
                        | 'domain' ⟨identifier⟩ ':' ⟨identifier⟩
                        | 'type' ⟨identifier⟩
```

Figure 4: Procedure definition grammar

The grammar of procedure definitions is given in Figure 4. The template syntax is used for parametric polymorphism. The programmer can use a domain quantifier variable to write a procedure which works with different domains. The domain variable is replaced by a concrete domain used in the procedure call. For example, the following procedure defines a procedure for sorting integer values of all domains (body omitted):

```
template<domain D>
D int[[1]] sort(D int[[1]] x) { }
```

The `domain` *variable* : `kind` form constrains the range of the domain variable to protection domains from a specific kind. The syntax `type` *variable* is used for quantifying over data types. The real SecreC programming language also allows quantification over dimensionality (e.g. `template<dim N>`) but this has been left out for the sake of simplicity.

3.1 Kind definitions

Currently, a PDK is defined as follows:

```
kind shared3p;
```

This definition includes no information about which data types and operations are supported by the PDK `shared3p`. Using the syntax added in this thesis, a PDK definition consists of the name of the kind and a list of data type definitions. The PDK definition syntax is given in Figure 5.

Most private data types have a corresponding public type. For example, a public `int` can be converted to a private `int` (called classifying) and when a private `int` is published it will be converted to a public `int`. Thus a data type definition should include a corresponding public type parameter. There is not always a direct mapping between public and private types. For example, we may want to define a private fixed point type for efficient arithmetic on real numbers. To initialise the private fixed point values, we want to use floating point literals so the corresponding public type would have to be `float`.

It is possible that a type has no public representation. For example, SecreC currently has types such as `xor_uint8`, `xor_uint16`, etc which are shared bitwise. That is, a value x is shared as random values x_1, x_2, x_3 such that $x = x_1 \oplus x_2 \oplus x_3$ where $\oplus$ is the XOR operator. This sharing scheme has more efficient comparison operators and can be used

```
⟨kind_definition⟩ ::= 'kind' ⟨identifier⟩ '{'
                      ⟨data_type_definition⟩+ '}'
⟨data_type_definition⟩ ::= 'type' ⟨identifier⟩ ';'
                         | 'type' ⟨identifier⟩ '{'
                           ⟨data_type_def_parameter⟩
                           (',' ⟨data_type_def_parameter⟩)* '}' ';'
⟨data_type_def_parameter⟩ ::= 'public' '=' ⟨primitive_data_type⟩
                            | 'size' '=' ⟨integer_literal⟩
⟨domain_declaration⟩ ::= 'domain' ⟨identifier⟩ ⟨identifier⟩ ';'
```

Figure 5: Protection domain kind definition and domain declaration grammar

for sorting values by converting to the XOR-shared type from an integer or unsigned integer type before sorting and convert back after sorting. This type does not have a public representation and is only used for optimising algorithms. To support data types like this, the public type parameter should be optional.

To support memory management, assignment and indexing by the SecreC compiler, a data type definition should be able to specify the size of the private values in bits. It is possible though that some data types do not have a flat representation in memory. For example, a lot of cryptographic methods use arbitrary-precision arithmetic which requires dynamic memory. Thus the size parameter should also be optional and the compiler should use system calls for types without a specified size as it does now.

The following is an example of a protection domain kind definition with two private data types:

```
kind my_kind {
    type int { public = int, size = 64 };
    type fix { public = float, size = 64 };
}
```

3.2 Operator definitions

Operator definitions already have limited support in SecreC. They are used to overload operators for different combinations of operand types. For example, the compiler only assumes the existence of multiplication system call that takes two private integer vectors as inputs. In the case of arithmetic secret sharing, a more efficient protocol exists for multiplying a private and a public integer vector. Overloaded operator definitions allow the user to define more efficient operators for special cases. The operator definition syntax remains unchanged in this thesis although type checking and semantics are improved.

```
⟨operator_definition⟩ ::= ⟨template⟩? ⟨type⟩ 'operator' ⟨binary_operator⟩
                          '(' ⟨procedure_parameter⟩ ','
                          ⟨procedure_parameter⟩ ')'
                          ⟨block_statement⟩
                        | ⟨template⟩? ⟨type⟩ 'operator' ⟨unary_operator⟩
                          '(' ⟨procedure_parameter⟩ ')'
                          ⟨block_statement⟩
```

Figure 6: Operator definition grammar

The following is an example of an operator definition in the full SecreC language:

```
template<domain D : shared3p>
D int[[1]] operator + (D int[[1]] x, D int[[1]] y) {
    __syscall("shared3p::add_int_vec",
              __domainid(D), x, y, y);
    return y;
}
```

This defines the addition operator for type `int` for all protection domains of kind `shared3p`. The `__syscall` syntax is used in the actual SecreC language to invoke system calls. In this case, a system call with the name `shared3p::add_int_vec` is called with parameters `x, y` and the result is written to `y`. The identifier of the domain is also passed to the syscall. This is how most operator definitions will be written: they invoke a system call which implements the protocol in a module written in C++ or some other programming language. In some occasions, when an optimised protocol has not been implemented, the operator can be defined in terms of other operators. In the case of real numbers, it makes sense to implement an inverse protocol and use it with multiplication to implement division.

3.3 Cast definitions

We extend the language to support cast definitions similarly to how operators are defined. Syntactically, a cast definition is just a procedure named "cast" that has one argument.

```
⟨cast_definition⟩ ::= ⟨template⟩? ⟨type⟩ 'cast' '(' ⟨procedure_parameter⟩
                      ',' ⟨procedure_parameter⟩ ')'
                      ⟨block_statement⟩
```

Figure 7: Cast definition grammar

The following is an example of a cast definition in the full SecreC language:

```
template<domain D : shared3p, type T>
D T[[1]] cast(D int[[1]] x) {
    D T[[1]] res(size(x));
    __syscall("shared3p::conv_int_to_$T\_vec",
              __domainid(D), x, res);
    return res;
}
```

Note `$T` in the name of the syscall. The SecreC compiler replaces it with the name of the type bound to `T`. In a variable declaration, the expression in the parentheses after the variable name gives the shape of the value. In this case, `res` is a vector as long as `x`. This definition defines conversion from `int` to all types in the `shared3p` kind. If conversions to all types are not supported, cast definitions can be written for all supported cases using overloading. Unlike C++, SecreC supports overloading based on the type of the returned value which is required for cast definitions.

4 Type system

This section describes the formal type system of the subset of SecreC described in Section 3. The subset includes language features that have been added in this thesis or are affected by the changes.

4.1 Value types

Information flow control is a method of protecting privacy in programming languages [Mye99]. Values are labeled with security classes and the type system is designed to facilitate static tracking of the flow of sensitive information. Derived values must preserve the security class of the value. For example, in an assignment `x = e`, the label of the variable `x` must be at least as restrictive as the label of the value of `e` [ML98]. The label of the variable ensures that the assigned variable is not used in a context which leaks the value.

In our case, the protection domain of the type designates the privacy of the value. There are two kinds of protection domains: private protection domains and the special `public` domain of public values. The relation $\sqsubseteq$ defines a partial ordering of the domains:

$$d \sqsubseteq d$$
$$\texttt{public} \sqsubseteq d,$$

where d is a variable designating a private protection domain. The definition says that public values can flow into every protection domain and private values from domain d can only flow in domain d.

To find the domain where values of domains d_1, d_2 can flow, we use the least upper bound (join) operator which is designated as $d_1 \sqcup d_2$. There is a special value $\top$ which is at the top of the hierarchy. That is, $d \sqsubseteq \top$ for every domain d. In our case, $d_1 \sqcup d_2 = \top$ means that there is no protection domain where values from both d_1 and d_2 can flow. A partial ordering with a least upper bound is an algebraic structure called a join-semilattice.

For a user-defined data type t of some PDK, we want to be able to use a public value of the public type corresponding to t in a context where t is expected. For example, if we define a kind with a `fix` type as follows:

```
kind shared3p {
   type fix { public = float };
}
domain pd_shared3p shared3p;
```

we want the following program snippet to type check:

```
float x = 42;
pd_shared3p fix y = x;
```

Semantically, the value of `x` should be implicitly classified (converted to a private value) as `float` is the corresponding public type of `fix`. In this thesis, we also need subtyping of data types to support statements and expressions where the private data type and its corresponding public data type co-occur. We can not just use a lattice of data types because the relation between private and public data type is in the PDK definition. Due to this, we use a lattice of pairs of protection domain and data type.

Let public(k, t) be the public type corresponding to t in kind k. Let public$(k, t) = \top$ if there is no corresponding public type. Let $\mathcal{D}$ be the set of protection domains, types(k)

the set of user-defined data types in kind k and kind(d) be the PDK of protection domain d. We can now define the partial ordering on the combination of protection domain and data type:

$$\begin{aligned}(d,t) \sqsubseteq (d,t), &\quad \text{if } d \in \mathcal{D} \wedge t \in \text{types}(\text{kind}(d))\\ (\texttt{public}, t_1) \sqsubseteq (d,t_2), &\quad \text{if } d \in \mathcal{D} \setminus \{\texttt{public}\} \wedge\\ &\quad t_1 \in \text{types}(\texttt{public}) \wedge\\ &\quad t_2 \in \text{types}(\text{kind}(d)) \wedge\\ &\quad \text{public}(\text{kind}(d), t_2) = t_1.\end{aligned}$$

The first rule states reflexivity, that is, data can flow inside a domain if the type does not change. The second rule states that information can flow from the public domain to a private domain if the data types match according to the public type parameter of the private data type definition.

We will use the notation $d\,t\,n$ to represent the triple of domain, data type and dimensionality. The variables d, t, n, x are metavariables used for domains, data types, dimensionalities and variables in the language.

4.2 Inference rules and coinduction

The type system is formalised using an axiomatic system. An axiomatic system consists of axioms which are statements that we accept as true, and inference rules which allow true statements to be derived from axioms. If the set of all syntactically valid programs in our programming language is P, we want to use type checking to find the subset of type-correct programs $X \subseteq P$. To show that a program is correct we produce a derivation tree which has the program as the root, inference rules as edges and axioms as leaves. This derivation is a proof of that the program is correct according to our typing rules.

Let c_f be a metavariable ranging over floating-point literals and c_i a metavariable ranging over integer literals. Let `float` and `int` be the types of floating-point and integer values. Let us use the following simple language of arithmetic expressions:

$$\begin{aligned}\langle \texttt{e} \rangle \;::=\; & c_f\\ \mid\; & c_i\\ \mid\; & \langle \texttt{e} \rangle \text{ '*' } \langle \texttt{e} \rangle\\ \mid\; & \langle \texttt{e} \rangle \text{ '+' } \langle \texttt{e} \rangle\end{aligned}$$

Figure 8: Grammar of arithmetic expressions

We can give axioms which state that integer and floating-point literals have types `int` and `float`:

$$\frac{}{c_i : \texttt{int}} \qquad \frac{}{c_f : \texttt{float}}$$

Figure 9: Axioms

The statement (typing judgement) is below the horizontal line and the premises are above the line. Since these two rules are axioms, there are no premises, that is, the typing judgements are always true. Let t be a metavariable ranging over types. We use the following rules for the two operators:

$$\frac{e_1 : t \qquad e_2 : t}{e_1 \texttt{ + } e_2 : t} \qquad\qquad \frac{e_1 : t \qquad e_2 : t}{e_1 \texttt{ * } e_2 : t}$$

Figure 10: Inference rules

The rules require that the operands of an arithmetic expression have the same type as the expression. We can now write a derivation tree proving that the expression $1 + (2 * 3)$ has type `int`:

$$\dfrac{1 : \texttt{int} \qquad \dfrac{2 : \texttt{int} \qquad 3 : \texttt{int}}{2 * 3 : \texttt{int}}}{1 + (2 * 3) : \texttt{int}}$$

Figure 11: Example derivation

The typing rules define a relation on the set of syntactically correct programs P which gives us the set of well-typed programs $X \subseteq P$ (programs that can be assigned a type). There are two ways to interpret a set of inference rules: inductively and coinductively. Inductive interpretation starts from the smallest set of programs which have a type (that is, integer and floating point literals) and adds programs to the set which can be derived from well-typed programs using inference rules. This process is repeated on the resulting set until no more programs can be added to the set. If we consider the step of this process as a function f operating on a set of programs, then X is the least fixed-point of f.

Using coinduction, we start from the set P and remove all programs that can not be proven using our inference rules. The set X is thus the greatest fixed-point. Coinduction allows programs with an infinite derivation to be type-correct. For example, if we also interpreted the grammar coinductively, we could have an infinite term $1 + 1 + \ldots + 1$ with type `int`. This term is not well-typed using inductive interpretation of the typing rules.

The typing rules in this thesis use coinductive interpretation which is emphasised by double lines in the inference rules. Because SecreC uses procedure templates, a procedure can not be checked before the template is instantiated with type variables. The typing rule for procedure calls checks that the procedure definition is correct. If we used inductive interpretation, such a rule would prohibit mutually recursive procedure definitions. For example, assume that we have defined procedures a and b. The definition of a calls b and the definition of b calls a. To prove that a is well-typed, we need to prove that b is well-typed, which requires us to again prove that a is well-typed and so on. The derivation is infinite which means that the program is not well-typed using inductive interpretation.

4.3 Typing rules

The typing rules for the language extensions in this thesis are based on [BLR14]. The rules of assignments and variable declarations have been changed due to the addition of user-defined types. Rules have been added for declassification, type conversion, binary, unary and postfix expressions, operator definitions and type conversion definitions. The rules of PDK definitions and domain declarations are given informally due to their simplicity.

Currently the language has no reference types so there can be no aliasing. Due to this, we only consider the types of bound variables and not the types of memory locations.

The typing rules do not formalise every static check that would be implemented in a real compiler. For example, the SecreC compiler checks that every execution path in a non-void procedure ends with a return statement. It also checks for unreachable statements (such as statements after return).

In SecreC, procedure names are not required to be unique. This is used for overloading – giving different definitions of a procedure based on the types of arguments and returned value. Due to this, we refer to functions with their locations. Let $\mathcal{L}$ be the set of program source locations. A function definition with name f and location $\ell \in \mathcal{L}$ will be referred to as f^ℓ. We will use the same notation for operator and cast definitions in our type system because they are procedures with a special name. We will use the following notational conventions:

- $\mathrm{pdk}(P)$ and $\mathrm{pd}(P)$ designate the set of protection domain kinds in P and the set of protection domains in P. We assume that the protection domain `public` and its PDK are implicitly defined in every program.

- $\mathrm{kind}_P(d)$ designates the PDK of protection domain d. We treat it as a set of data type definitions. The PDK of the `public` domain includes the built-in types `bool`, `int`, `float`.

- $\mathrm{arg}_P(\ell)$ designates the list of declarations of formal arguments and $\mathrm{ret}_P(\ell)$ the return type of the definition at location ℓ.

- $\mathrm{impl}_P(f; d_1\, t_1\, n_1, \ldots, d_m\, t_m\, n_m \to d) \in \mathcal{L} \cup \{\bot\}$ is the location of the best matching definition of all procedures with name f or $\bot$ if there is no matching definition or multiple matching definitions that are equally fitting. The match is determined based on the domains $d_1, \ldots, d_m$, data types $t_1, \ldots, t_m$ and dimensionalities $n_1, \ldots, n_m$ of the arguments and the domain d of the output. Note that the real SecreC programming language also allows overloading on other components of the return type, not just the domain but for simplicity we only use the domain here. Likewise, for selecting operator and cast definitions, we use implop_P. It is not specified how impl_P selects the best matching procedure definition. The algorithm for selecting the best one from matching operator and cast definitions will be explained later.

- $\delta(\ell)$ is the set of protection domain quantifiers of the definition at location ℓ. $\tau(\ell)$ is the set of data type quantifiers of the definition at location ℓ.

- $\mathrm{body}_P(\ell)$ designates the body of the definition at location ℓ. Given mappings $\Delta : \delta(\ell) \to \mathrm{pd}(P)$ and $T : \tau(\ell) \to \mathrm{pd}(P)$, $\mathrm{body}_P^{\Delta,T}(\ell)$ designates the body of the definition at location ℓ where domain quantifier variables $d \in \delta(\ell)$ have been replaced by $\Delta(d)$ and type quantifier variables $t \in \tau(\ell)$ have been replaced by $T(t)$. $\mathrm{arg}_P^{\Delta,T}(\ell)$ and $\mathrm{ret}_P^{\Delta,T}(\ell)$ are defined similarly.

- $\mathrm{unif}_\ell(d'_0\, t'_0, \ldots, d'_m\, t'_m) = (\Delta, T)$ gives a mapping Δ from $\delta(\ell)$ to protection domains declared in the program such that $\Delta(d_i) = d'_i$ for every d_i quantified by the definition at ℓ. Likewise, T is a mapping from type quantifiers $\tau(\ell)$ used in the definition at ℓ to data types. The function unifop_ℓ returns these mappings for operator definitions.

- $\mathrm{main}(P)$ designates the main procedure of program P.

- numeric(t) is a predicate that is true if t is a numeric data type (`int`, `float`).
- relational(o) is a predicate that is true if o is a relational operator (e.g. $<$, $\geq$).
- c_f and c_i are metavariables that range over floating point and integer literals.

The meanings of the judgements used in the typing rules are:

- $\vdash P$ states that the program P is well-typed.
- $P; \Delta; T \vdash f^\ell$ means that the definition f^ℓ in the program P is well-typed if domain quantifiers in $\delta(\ell)$ are replaced by the domains given by Δ and type quantifiers in $\tau(\ell)$ are replaced by the types given by T.
- $P; d\,t\,n; x_1 : d_1\,t_1\,n_1, \ldots, x_m : d_m\,t_m\,n_m \vdash s$ means that s is well-typed in a procedure that has return type $d\,t\,n$ and the free variables x_i have types $d_i\,t_i\,n_i$. We will use Γ as shorthand for $x_1 : d_1\,t_1\,n_1, \ldots, x_m : d_m\,t_m\,n_m$. The notation $x : d\,t\,n, \Gamma$ means that x is given type $d\,t\,n$ in context Γ (Γ is extended or the type of x is replaced).
- $P; \Gamma \vdash e : d\,t\,n$ means that the expression e has type $d\,t\,n$ in the given context.

The typing rules for basic expressions are given in Figure 12.

$$\frac{}{P; \Gamma \vdash \texttt{true} : \texttt{public bool } 0} \qquad \frac{}{P; \Gamma \vdash \texttt{false} : \texttt{public bool } 0}$$

$$\frac{}{P; \Gamma \vdash c_f : \texttt{public float } 0} \qquad \frac{}{P; \Gamma \vdash c_i : \texttt{public int } 0} \qquad \frac{(x : d\,t\,n) \in \Gamma}{P; \Gamma \vdash x : d\,t\,n}$$

$$\frac{\begin{array}{c}(d_1, t_1) \sqsubseteq (d, t) \;\ldots\; (d_m, t_m) \sqsubseteq (d, t) \\ P; \Gamma \vdash e_1 : d_1\,t_1\,(n-1) \;\ldots\; P; \Gamma \vdash e_m : d_m\,t_m\,(n-1)\end{array}}{P; \Gamma \vdash \{e_1, \ldots, e_m\} : d\,t\,n}$$

$$\frac{P; \Gamma \vdash e : d'\,t'\,n \qquad (\texttt{public}, t) \sqsubseteq (d', t') \qquad d' \neq \texttt{public}}{P; \Gamma \vdash \texttt{declassify(}e\texttt{)} : \texttt{public}\ t\ n}$$

Figure 12: Expression typing rules

The rule for multi-dimensional array literals checks that the dimensionality of the components of the array is one less than the dimensionality of the array. That is, a vector consists of scalars, a matrix consists of vectors and so on. The condition $(d_i, t_i) \sqsubseteq (d, t)$ checks that the pair of domain and data type of the sub-expression type precedes the pair of the array type according to the partial ordering defined in Section 4.1. This allows data from the public domain to flow into a private domain. The definition of the ordering also requires that the data types match. For example, we can write `pd_shared3p fix[[1]] x = {1.2, 3.4};` where the public floating point scalars are implicitly classified. This rule is a simple example of how we handle the addition of user-defined data types in this thesis.

The rule for declassification checks that the expression e being declassified is well-typed. The domain of the declassify expression must be public. The dimensionality of

expression e and the result must match. The condition $(\texttt{public}, t) \sqsubseteq (d', t')$ is required due to the addition of user-defined private data types. It checks that the data type t of the result is the public type corresponding to t'.

The rules for the program, procedure definitions and procedure calls are given in Figure 13.

$$\frac{P; \bot\bot\bot; \emptyset \vdash \text{main}(P)}{\vdash P}$$

$$\frac{\begin{array}{c} P; \Gamma \vdash e_1 : d_1\, t_1\, n_1 \;\; \ldots \;\; P; \Gamma \vdash e_m : d_m\, t_m\, n_m \\ \ell = \text{impl}_P(f; d_1\, t_1\, n_1, \ldots, d_m\, t_m\, n_m \to d) \\ (\Delta, T) = \text{unif}_\ell(d\, t, d_1\, t_1, \ldots, d_m\, t_m) \quad P; \Delta; T \vdash f^\ell \quad \text{ret}_P^{\Delta,T}(\ell) = d\, t\, n \end{array}}{P; \Gamma \vdash f(e_1, \ldots, e_m) : d\, t\, n}$$

$$\frac{P; \text{ret}_P^{\Delta,T}(\ell); \text{arg}_P^{\Delta,T}(\ell) \vdash \text{body}_P^{\Delta,T}(\ell)}{P; \Delta; T \vdash f^\ell}$$

Figure 13: Procedure and program typing rules

A program P is well-typed if it contains a function named "main" which is well-typed. The rule for procedure calls checks that:

- There is a single best matching definition of f (found by the function impl_P) in program P. Note that if $\ell = \bot$, the procedure call does not type check.
- The body of the procedure f^ℓ is well-typed if the quantified domain variables are replaced by the domains used in the procedure call arguments.
- The arguments of the procedure call are well-typed.
- The types of supplied arguments match the types of formal arguments. We do not use implicit classification with regular procedure calls.
- If the domain quantifier variables are replaced by domains used in the procedure call, the type of the returned value is the same as the type of the procedure call expression.

The rule which checks the procedure definition has access to the domain quantifier mapping Δ and type quantifier mapping T. When type checking the body of the procedure, the quantifiers in the return type and formal parameter list are substituted and the formal parameters are added to the context. This is where the return and parameter types are introduced to the context $P; d\, t\, n; \Gamma$. Note that a procedure is type-checked at a call site and requires the quantifier mappings Δ and T. In practical terms, this means that a procedure definition is not checked until a call of the procedure is checked.

The typing rules for basic statements are given in Figure 14.

$$\frac{P; d\,t\,n; \Gamma \vdash s_1 \qquad P; d\,t\,n; \Gamma \vdash s_2}{P; d\,t\,n; \Gamma \vdash s_1 \ \texttt{;}\ s_2}$$

$$\frac{d \in \mathrm{pd}(P) \qquad t \in \mathrm{kind}_P(d) \qquad n \geq 0 \qquad P; d_0\,t_0\,n_0;\ x : d\,t\,n, \Gamma \vdash s}{P; d_0\,t_0\,n_0; \Gamma \vdash \{d\,t\,\texttt{[[}n\texttt{]]}\ x\texttt{;}\ s\}}$$

$$\frac{x : d\,t\,n \in \Gamma \qquad P; \Gamma \vdash e : d'\,t'\,n' \qquad (d', t') \sqsubseteq (d, t) \qquad n' = 0 \vee n' = n}{P; d_0\,t_0\,n_0; \Gamma \vdash x\ \texttt{=}\ e\texttt{;}}$$

$$\frac{P; \Gamma \vdash e : \texttt{public bool}\ 0 \qquad P; d\,t\,n; \Gamma \vdash s}{P; d\,t\,n; \Gamma \vdash \texttt{while}\ e\ \ s}$$

$$\frac{P; \Gamma \vdash e : \texttt{public bool}\ 0 \qquad P; d\,t\,n; \Gamma \vdash s_1 \qquad P; d\,t\,n; \Gamma \vdash s_2}{P; d\,t\,n; \Gamma \vdash \texttt{if (}e\texttt{)}\ \ s_1\ \ s_2}$$

$$\frac{P; \Gamma \vdash e : d'\,t'\,n \qquad (d', t') \sqsubseteq (d, t)}{P; d\,t\,n; \Gamma \vdash \texttt{return}\ e\texttt{;}} \qquad \frac{P; \Gamma \vdash e : d\,t\,n}{P; d_0\,t_0\,n_0; \Gamma \vdash e\texttt{;}}$$

Figure 14: Statement typing rules

The block statement rule checks the following conditions:

- The domain of the declared variable has been declared in the program.
- The data type of the variable is defined in the kind of the domain of the variable type.
- Dimensionality is a natural number.
- The statement of the block is well-typed in the context extended with the type of the declared variable.

The assignment given here only applies to regular assignment, not arithmetic assignment. This rule takes into account user-defined private data types which were added in this thesis. It checks the following conditions:

- x has been declared.
- The expression e is well-typed.
- The pair (d', t') of domain and data type of the expression e matches the pair (d, t) of the variable x according to the partial ordering.
- Dimensionality of the value of e is zero (scalar) or equal to the dimensionality of the variable. If a scalar is assigned to a higher dimensional variable, all values are replaced with the scalar.

It is required that the conditions of while-loops and conditionals are public boolean scalars. If a private boolean would be allowed as the condition of an if-then-else statement, there would be two main options to evaluate the statement:

- The runtime system stops when evaluating the condition (exception).
- The runtime system declassifies the condition to decide which branch to evaluate. This leaks the private condition.

Neither behaviour is desirable. If if-then-else was an expression, it would be possible to evaluate both branches and then combine them. For example if b is the condition (represented as zero or one) and t, f are the values of the branches, the result would be $b \cdot t + (1 - b) \cdot f$. This does not work well in a language with side effects and can be very inefficient so we require that the condition is always public. Arithmetisation can be used manually if the condition is not public. The SecreC standard library provides a procedure `choose(condition, trueValue, falseValue)` for this purpose.

The return statement rule again allows a public value to be returned when a private value is expected like the assignment rule.

The expression statement rule requires that the expression e is well-typed and has some type $d\,t\,n$.

4.4 Kind definitions and domain declarations

We have now described the rules of the basic language features, including the changes required to support user-defined data types, and now turn to the extensions added by the author that will allow PDKs (including private data types), operators and type conversions to be defined.

The rules for kind definitions and domain declarations are very simple so they will be given in natural language without formal rules. The rules for PDK definitions are:

- The names of PDKs must be unique.
- The names of data types in a PDK must be unique. It is allowed to have data types with the same name in different PDKs.
- The size of a data type defined in a PDK definition must be one or a multiple of eight. If the size of a value is some number of bytes, the compiler can easily generate memory management (allocation, deallocation, assignment, indexing) code. Data type size of one is supported to efficiently represent booleans in bit vectors.
- If the defined type has the name of a built-in type then the corresponding public type must be the built-in type. For example, `type int { public = int }` is legal but `type int { public = float }` is not. This is used to avoid confusing the programmer who will be using the PDK. In this case the public type parameter is optional and will be set to the correct type if the parameter is omitted.

The rules for protection domain declarations require that the names of protection domains are unique and the PDK of each protection domain is defined in the program.

4.5 Operator definitions and arithmetic expressions

In SecreC, arithmetic expressions can have multi-dimensional arrays as operands. Operators are applied point-wise. When one operand is a scalar and the other is not, the scalar is supplied as one operand when computing each value of the higher dimensional

operand. An issue with allowing the user to define operators is that there are a lot of combinations of parameter types. The programmer of a protection domain kind would have to write a lot of definitions. For example, if an operator definition is a normal procedure and we require actual parameter types to be equal to formal parameter types, all the following combinations would require a separate definition for each binary operator:

- *private scalar* $\otimes$ *private scalar*
- *private scalar* $\otimes$ *public scalar*
- *private vector* $\otimes$ *private vector*
- *private vector* $\otimes$ *public vector*
- *private vector* $\otimes$ *private scalar*
- *private vector* $\otimes$ *public scalar*

As described in Section 2.3, if there are n data types and m supported binary operators in a PDK, there would be up to $n \cdot m \cdot 3 \cdot 3$ definitions. Since most definitions are similar (they call a system call with a structured name depending on the operand types), it would probably be possible to generate the source code of the definitions using an external program. This is not desirable because the module containing the PDK definition should be readable and extensible by a human. Our goal is to reduce the number of definitions that the programmer has to write.

PDK modules usually define a single system call for each combination of data type and operator which takes vectors as inputs. This is actually the only required protocol to support all the different combinations. For example, let us assume that we have a definition for multiplying private vectors. To multiply two private scalars, we can consider the scalars as one-element vectors. To multiply a private vector a with a private scalar b we can create a vector b' with the shape of a where all elements are b and multiply a and b'. If one operand is public we can classify it. Thus, it should be sufficient to write just a definition on private vectors for each operator and data type combination which calls the system call implementing the arithmetic protocol on vectors. This definition could then be used in all expressions.

Sometimes, it is possible to implement a more efficient protocol for some combination of operands. We should also allow definitions where one of the operands is public or scalar. An arithmetic expression should be compiled to use the most specific matching implementation which requires the smallest number of implicit classifications and reshaping.

$$\frac{\begin{array}{c} P;\Gamma \vdash e_1 : d_1\, t_1\, n_1 \;\; \ldots \;\; P;\Gamma \vdash e_m : d_m\, t_m\, n_m \\ \ell = \mathrm{implop}_P(o; d_1\, t_1\, n_1, \ldots, d_m\, t_m\, n_m \to d\, t\, n) \\ (\Delta, T) = \mathrm{unifop}_\ell(d\, t, d_1\, t_1, \ldots, d_n\, t_n) \\ P;\Delta;T \vdash o^\ell \;\; \mathrm{ret}_P^{\Delta,T}(\ell) = d_0\, t_0\, n_0 \\ \mathrm{arg}_P^{\Delta,T}(\ell) = x_1 : d'_1\, t'_1\, n'_1, \ldots, x_m : d'_m\, t'_m\, n'_m \\ (n_1 = \ldots = n_m) \vee n_1 = 0 \vee \ldots \vee n_m = 0 \\ (n_1 = n'_1 = 0 \vee n'_1 = 1) \wedge \ldots \wedge (n_m = n'_m = 0 \vee n'_m = 1) \\ d_1 \sqcup \ldots \sqcup d_m = d_0 \\ (d_1, t_1) \sqsubseteq (d'_1, t'_1) \;\; \ldots \;\; (d_m, t_m) \sqsubseteq (d'_m, t'_m) \end{array}}{P;\Gamma \vdash o(e_1, \ldots, e_m) : d\, t\, n}$$

Figure 15: Arithmetic expression typing rule

The typing rule for arithmetic expressions is given in Figure 15. We treat arithmetic expressions as calls to special functions so it is similar to the procedure call rule. We assume that there are built-in implementations for all public types. The variable o ranges over operator names. The rule checks the following conditions:

- The operands are well-typed.
- There is a single best matching definition. If $\ell = \bot$, the definition is missing and type checking fails.
- The judgment $P;\Delta;T \vdash o^\ell$ checks that the operator definition type checks when the domain quantifier variable is replaced with a domain used in the arithmetic expression and the type quantifiers are replaced with data types used in the arithmetic expression. The method for finding the mappings Δ and T is explained later.
- The condition $(n_1 = \ldots = n_m) \vee n_1 = 0 \vee \ldots \vee n_m = 0$ checks that the operands have the same dimensionality or one of them is zero. Note that we only have unary and binary operators so $m \Leftarrow 2$ which means that in practice we check that in a binary operation, if one of the operands is a multi-dimensional array then the second operand must be a scalar or have the same dimensionality.
- The condition $(n_1 = n'_1 = 0 \vee n'_1 = 1) \wedge \ldots \wedge (n_m = n'_m = 0 \vee n'_m = 1)$ checks that the dimensions of the operands match the dimensions of the operator definitions. If the definition expects a scalar, the operand must be a scalar ($n_i = n'_i = 0$). If the definition expects a vector then it does not matter what the dimensionality of the operand is because scalars and multi-dimensional arrays can be converted to vectors. The type checking rule of operator definitions does not allow parameters with a dimensionality other than 0 or 1. Arrays with higher dimension can still be used in arithmetic expressions because a value with any dimension can be converted into an flat array. This is described later with the operator definition rules.
- The least upper bound of the domains of the types of the operands is equal to the domain of the return type of the operator definition ($d_1 \sqcup \ldots \sqcup d_m = d_0$). This condition is used to ensure that the compiler does not use a private definition when a public definition can be used. For example, let us assume that this rule is omitted, that there is a private integer multiplication definition and that the

expression has two public operands and requires a private result. The definition matches the expression because the operands can be implicitly classified due to the $(d_i, t_i) \sqsubseteq (d'_i, t'_i)$ rule. This would happen in a statement such as `pd_shared3p int x = 3 * 4;`. It is more efficient to multiply public values and then classify the result than to classify the operands and multiply private values so we want to avoid using the private definition for public multiplication.

- The $(d_i, t_i) \sqsubseteq (d'_i, t'_i)$ condition allows operations on a mix of public and private operands. For example, if there is a definition for private integer multiplication, the expression `x * 2` where `x` is a private integer, is legal. The public operand is implicitly classified and the private definition is used.

$$\frac{\begin{array}{c} n, n_1, \ldots, n_m \in \{0,1\} \quad \max(n_1, \ldots, n_m) = n \\ (d_1, t_1) \sqcup \ldots \sqcup (d_m, t_m) = (d', t') \neq \top \quad t' = t \text{ if } \neg\text{relational}(o) \\ |\delta(\ell)| \leq 1 \quad d' = d \quad t \neq \texttt{void} \\ \arg_P^{\Delta,T}(\ell) = x_1 : d_1\, t_1\, n_1, \ldots, x_m : d_m\, t_m\, n_m \quad \text{ret}_P^{\Delta,T}(\ell) = d\, t\, n \\ P; \text{ret}_P^{\Delta,T}(\ell); \arg_P^{\Delta,T}(\ell) \vdash \text{body}_P^{\Delta,T}(\ell) \end{array}}{P; \Delta; T \vdash o^\ell}$$

Figure 16: Operator definition typing rule

The operator definition typing rule is given in Figure 16. It is similar to the procedure definition rule. The conditions are:

- The operands and the returned value are either scalars or vectors. A definition taking two vectors can be used for all pairs of dimensionalities because a scalars and arrays with a dimension higher than one can converted to a vector. The resulting value can be converted back to a multi-dimensional array, if necessary, by copying the shape of the operand. This simplification allows a single definition (taking two vectors and returning a vector) to be written for all combinations of input dimensions. Scalars are allowed in order to write optimised definitions for expressions taking a scalar and a multi-dimensional array.

- The dimensionality of the returned value is the maximum of the dimensionalities of the operands. A definition taking two vectors and returning a scalar does not make sense when operators are expected to work point-wise.

- Let (d', t') be the least upper bound of the domain and data type pairs of the operands. If operator o is not relational then t' must equal the data type of the returned value. If o is relational then the least upper bound must not be $\top$ (i.e. it must be defined). This condition ensures that definitions are not written for different types of operands or return value. For example, if a definition takes `float` inputs and returns an `int` then $t' = \texttt{float} \sqcup \texttt{float} = \texttt{float} \neq \texttt{int}$ so the definition does not type check. When an operator takes an `int` and a `float` as inputs then $t' = \texttt{int} \sqcup \texttt{float} = \top$ which does not equal any data type. Relational operators usually return boolean values so we can not require $t' = t$, otherwise `int` or `float` comparison can not be defined. Sometimes, for the sake of simplicity, a PDK developer may wish to emulate booleans using numbers as in the C programming

language. Thus, when defining relational operators, the only requirement for the operand data types is that they match (the least upper bound is not $\top$). We can not write these conditions for relational and other operators as the equality of operand data types or the equality of operand and return data type because operands can also be public. For example, if we have defined `type fix { public = float }` we may want to define a multiplication of a private `fix` and a public `float`. This should be legal since `float` is the corresponding public type of `fix`.

- There is at most one domain quantifier. Operators are defined on data types of PDKs. In a single arithmetic expression, only two domains make sense: some private domain and the public domain. There is only a single public domain so we only need a quantifier for private domains. The procedure `reclassify` should be used to explicitly convert values between different protection domains if necessary.

- The least upper bound of the domains of the types of the operands is equal to the domain of the returned value ($d' = d$). This is used to avoid definitions that break the assumption about data flow between protection domains. For example, a definition of multiplication that takes two private values as inputs and returns a public value is illegal because it would have to declassify the result. A definition taking a private and a public value and returning a private value is valid and can be used to write an optimised implementation for this special case. Note that this rule also disallows a definition that takes two public values and returns a private value.

- The type of the returned value is not `void`. This is disallowed by the $t' = t$ and $t' \neq \top$ rule for non-relational operators but needs to be checked for relational operators.

The typing rule for postfix increment expression is given in Figure 17. The rule for postfix decrement is exactly the same except the subtraction operator is used. The full SecreC language also has prefix increment and decrement operators which are type checked in the same manner.

$$\frac{\begin{array}{c} t' = \text{public}(\text{kind}_P(d), t) \quad \text{numeric}(\text{t}') \\ (\Delta, T) = \text{unifop}_\ell(d\,t, \texttt{public}\,t') \\ \ell = \text{implop}_P(+; d\,t\,n, \texttt{public}\ t'\ 0 \to d\,t\,n) \\ P; \Delta; T \vdash +^\ell \quad P; \Gamma \vdash a : d\,t\,n \end{array}}{P; \Gamma \vdash a\texttt{++} : d\,t\,n}$$

Figure 17: Postfix increment typing rule

An increment expression again calls an operator definition like unary/binary expressions. The typing rule checks the following conditions:

- The value being incremented type checks.

- The data type of the value has a corresponding numeric public type. If $t' = \text{public}(\text{kind}_P(d), t) = \bot$ or t' is not numeric, type checking fails because we do not know how to create the value one for this private type. This could be solved by allowing the user to define increment and decrement operators as well. Since this

case is not common, we use the addition definitions so that the PDK programmer can write fewer definitions.

- There is a definition of addition for the type of the value that is being incremented. Note that the user can not define an increment operator. implop_P searches for an addition definition that takes an operand with the type of a and a public scalar with the corresponding public data type of t. The compiler creates a value of one with the type t' and calls the addition definition with a and the public one. implop_P considers the fact that scalars can be converted to vectors and public values can be classified so definitions taking two private values and/or two vectors will also work.
- The operator definition type checks if the domain quantifier is replaced by the domain of the value being incremented.

The rule for arithmetic assignment is given in Figure 18. For the sake of simplicity, it is given for the expression `a += c`. The rule is the same in other cases: in the expression `a *= c`, we search for a definition of multiplication and so on.

$$\frac{\begin{array}{c}\ell = \text{implop}_P(+; d_0\, t_0\, n_0, d_1\, t_1\, n_1 \to d_0\, t_0\, n_0) \\ (\Delta, T) = \text{unifop}_\ell(d_0\, t_0, d_0\, t_0, d_1\, t_1) \quad P; \Delta; T \vdash +^\ell \\ P; \Gamma \vdash a : d_0\, t_0\, n_0 \quad P; \Gamma \vdash b : d_1\, t_1\, n_1\end{array}}{P; \Gamma \vdash a \texttt{ += } b : d_0\, t_0\, n_0}$$

Figure 18: Arithmetic assignment typing rule

The arithmetic assignment rule is straightforward. We find an addition definition which can take a and b as inputs. The rule checks the following conditions:

- Variables a and b are well-typed. The type of the expression is the type of a.
- There is a single best matching definition of addition for parameters with the types of a and b.
- The operator definition type checks when the quantifier variables are replaced using the mapping Δ or T.

4.6 Cast definitions and cast expressions

$$\frac{\begin{array}{c}P; \Delta; T \vdash \texttt{cast}^\ell \quad t \in \text{kind}_P(d) \quad t' \neq \texttt{void} \\ \ell = \text{implop}_P(\texttt{cast}; d\, t' \to d\, t) \quad (\Delta, T) = \text{unifop}_\ell(d\, t, d\, t') \\ P; \Gamma \vdash e : d\, t'\, n\end{array}}{P; \Gamma \vdash \texttt{(}t\texttt{)}\ e : d\, t\, n}$$

Figure 19: Cast expression typing rule

The typing rule for cast expressions is given in Figure 19. We assume that there are built-in definitions for cast definitions between types in the public domain and they are not type checked. The typing rule checks the following conditions:

- The expression e is well-typed.
- The type of e is not `void`. This is for emphasis but does not make a difference because a definition of a cast from `void` to another type is invalid.
- The data type t, which the value is being converted to, exists in the PDK of d.
- There is a single best matching cast definition.
- The cast definition is well-typed when the quantifier variables are replaced using the mapping Δ or T.

The function unifop_ℓ finds the quantifier mappings Δ and T. Here we consider a cast as a unary operator which allows us to use unifop_ℓ which binds the single allowed domain quantifier variable to the domain of the expression whose result is being converted.

The function implop_P finds the best matching cast definition. The weighing system described for ordering operator definitions is also used for cast definitions. In practice, we do not need to count implicit classifications for casts because this is not allowed by the cast definition rules.

$$\frac{\begin{array}{c}|\delta(\ell)| \leq 1 \quad d = d_0 \neq \texttt{public} \quad n = n_0 = 1 \quad t \neq \texttt{void} \\ \text{ret}_P^{\Delta,T}(\ell) = d\,t\,n \quad \text{arg}_P^{\Delta,T}(\ell) = x : d_0\,t_0\,n_0 \\ P; \text{ret}_P^{\Delta,T}(\ell); \text{arg}_P^{\Delta,T}(\ell) \vdash \text{body}_P^{\Delta,T}(\ell)\end{array}}{P; \Delta; T \vdash \texttt{cast}^{\ \ell}}$$

Figure 20: Cast definition typing rule

The rule for cast definitions is given in Figure 20. It checks the following conditions:

- The body of the definition is well typed when the quantifier variables are replaced using the mapping Δ or T.
- The returned data type is not `void`. This is added for emphasis. It is also guaranteed by the return domain and dimension rules.
- The domain of the argument and returned value is not public and it is the same. This means casts can be defined only between private types in a single domain. Allowing the parameter of the cast to be public is not useful because the rule for assignment already allows us to write a statement like `uint b; pd_shared3p int a = (int) b;` where the result of the cast is implicitly classified.
- The dimensionality of the argument and returned value is one. The definition must thus operate on a vector. Like in the case of operator definitions, a definition written for vectors can be used with scalars (which can be converted to one element vectors) and arrays with higher dimensionality (which can be flattened before the cast and reshaped to the original dimensions after the cast).

4.7 Unification and ordering of definitions

The mapping Δ from template domain quantifier variables to concrete domains and mapping T from type quantifiers to concrete data types is returned by the function unif_ℓ, in the case of procedures, or unifop_ℓ, in the case of operator definitions. The procedure unifier iterates over pairs of formal parameter type and supplied parameter type. If quantifier variables occur in the formal parameter type, bindings are added to Δ or T which map the variables to the types in the supplied parameter. In the following example, `T` is bound to `int` since the value passed in the position of the formal parameter `x` has type `int`:

```
template<type T>
T factorial(T x) {
    if (x == 0) return 1;
    return x * factorial(x - 1);
}
void main() {
    int x = 5;
    print(factorial(x));
}
```

Unification may fail in the following cases:

- A quantifier variable declared in the `template<...>` list does not appear in the signature of the procedure.
- The same variable is used in the position of different components of the type. For example a parameter is declared `T T x` which would mean that `T` is both a data type and a domain.
- A variable is bound to two different types.
- A domain quantifier variable has a constraint kind but the domain bound to the variable is not from that kind.

Unifying operator definition quantifiers proceeds almost exactly the same way as for procedure templates. The typing rule for operator definitions required that there is at most one domain quantifier. In the case of unary operators, the domain quantifier is bound to the domain of the operand. If we have a binary expression with operand domains d_1 and d_2, the domain quantifier is bound to $d_1 \sqcup d_2$. If $d_1 \sqcup d_2 = \top$ then unifop_ℓ fails. If a data type variable t is bound to data types $t_1, \ldots, t_m$ with domains $d_1, \ldots, d_m$, then instead of requiring t_i to be equal (as unif_ℓ does), the bound type is t' where $(d', t') = (d_1, t_1) \sqcup \ldots \sqcup (d_m, t_m)$. If the least upper bound is $\top$, unifop_ℓ fails and the operator does not match the expression.

The following program snippet illustrates why it was decided to bind t' like this:

```
kind shared3p {
    type fix { public = float };
}
domain pd_shared3p shared3p;

template<domain D : shared3p, type T>
```

```
D T[[1]] operator * (D T[[1]] x, D T[[1]] y) {
    // body omitted
}

void main() {
    pd_shared3p fix x;
    float y;
    x * y;
    y * x;
}
```

The definition should be a match for both expressions because we can classify `y` to a value of type `fix`. In the expression `x * y`, if we first bind `T = fix` then unification fails because the second operand binds `T = float` $\neq$ `fix`. If instead of equality, we require that the type being bound is less than or equal to the type already bound (according to the partial ordering) then the expression `y * x` would fail because `fix` $\sqsubseteq$ `float` does not hold. When computing the least upper bound of the types bound to `T` we get `fix` $\sqcup$ `float` = `fix` which instantiates the operator template correctly.

To find the best matching operator definition, implop_P finds all matching definitions, assigns each a weight and returns the location of the match with the lowest weight. Weighing is designed to prefer more specific definitions to general definitions. If multiple matches have the same weight then the expression does not type check. The weight of an operator definition consists of three components which are compared one after the other (i.e. the second component is compared only if the first is equal). The components are:

(1) Domain weight. Definitions without a domain quantifier have weight zero, definitions with a domain quantifier with a kind constraint have weight one and the weight of quantifiers without a kind constraint is two. The reasoning is that operator definitions written for a specific PDK are more specific than operator definitions that can be used with all PDKs.

(2) The number of times quantifiers appear in the return type or operand type of the definition. The reasoning is that types with quantifiers are more general. A definition with fewer general types is more specific.

(3) The number of implicit classifications and operand reshapings necessary to use the definition. To explain the reason for this component, let us assume that we are multiplying two scalars. A definition that takes two scalars and a definition that takes two vectors both match. Using this weighing, the definition on scalars is more specific because it does not require scalar to vector conversion. This is desired because we assume that the scalar multiplication definition is more specific. The vector multiplication definition can be used with scalars, vectors, matrices, etc. while the scalar definition can only be used when the inputs are scalars.

The following example illustrates how multiple matching definitions can coexist:

```
template<domain D : shared3p, type T>
D T[[1]] operator * (D T[[1]] x, D T[[1]] y) {
    __syscall("shared3p::mul_$T\_vec",
              __domainid(D), x, y, y);
    return y;
}

template<domain D : shared3p, type T>
D T[[1]] operator * (D T[[1]] x, T y) {
    D T[[1]] res(size(x));
    if (y < 0) {
        y = -y;
        x = -x;
    }
    for (T i = 0; i < y; ++i) {
        res = res + x;
    }
    return res;
}

template<domain D : shared3p>
D float[[1]] operator * (D float[[1]] x, float y) {
    __syscall("shared3p::mulc_float_vec",
              __domainid(D), x, y, x);
    return x;
}
```

Let us assume that we are multiplying a private float vector with a public float scalar. All three definitions can be used. The first two definitions both have a constrained domain quantifier. The second one has a lower weight because it requires no reshaping or classification of the second operand. It uses the definitions of addition and negation (which we assume exist in this example). In the case of additive secret sharing, adding integer values requires no network communication so this definition is relatively efficient (if we ignore the overhead of repeated system calls to the addition protocol). Floating point numbers can not be added locally which is why the third definition uses a more efficient specialized protocol. Out of the three definitions, the third one will be chosen because it has the least occurences of quantifiers in the parameters and this is also the one most likely to be preferred by the programmer.

5 Implementation

5.1 Architecture and overview of changes

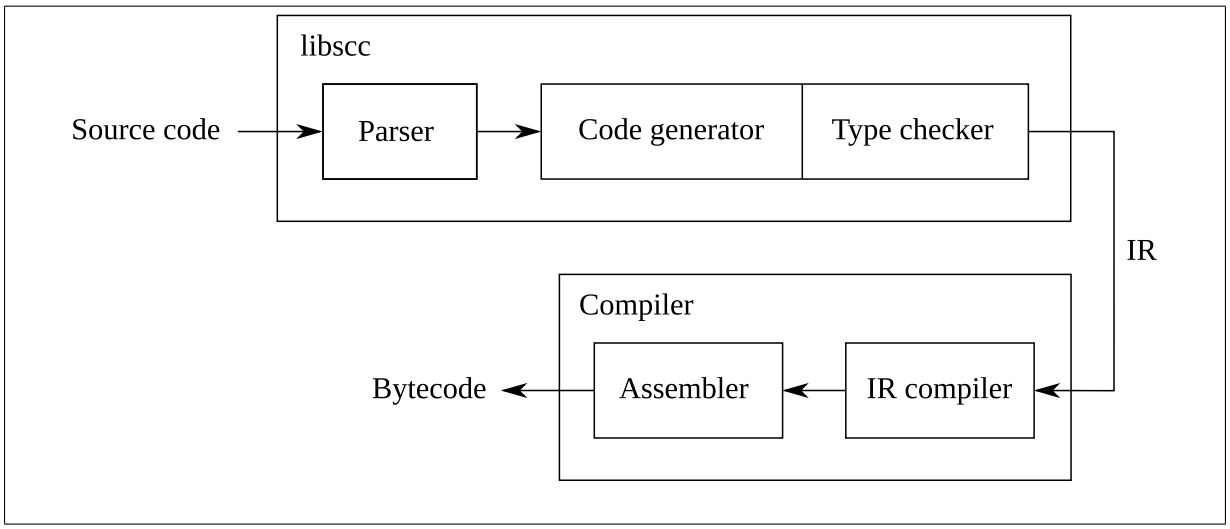

Figure 21: SecreC compiler architecture

The SecreC compiler[1] is implemented in the C++ programming language. Figure 21 shows the architecture of the compiler. The libscc library contains a parser implemented using flex[2] and the GNU bison parser generator[3]. After parsing, code in the intermediate representation is generated from the abstract syntax tree. The compiler checks types during code generation instead of type checking the whole program first. A compiler executable uses the libscc library to generate code in the intermediate representation which it then compiles to the Sharemind virtual machine assembly language which is compiled to the Sharemind virtual machine bytecode using an assembler library. libscc also contains a framework for statically analysing SecreC programs based on the intermediate representation [Ris10]. System calls are linked to C procedures implemented in the Sharemind server modules when the program is loaded by the virtual machine. Most of the work by the author was in the code generator and type checker. The compiler component remaind mostly unchanged. The system in the compiler that generated operator system call names when compiling intermediate representation arithmetic expressions on private values, was removed.

The author changed the abstract syntax tree, lexical analyser and parser to allow cast definitions, operator definitions, protection domain kind definitions with user defined types and overloaded prefix, postfix, arithmetic assignment and cast expression. Unary and binary expressions were already overloaded. When the type checker has found a matching cast/operator definition, the symbol of the definition in the symbol table is stored in the abstract syntax tree node of the expression. When the code generator has to compile a procedure call to the definition, it can retrieve the correct definition from the expression node.

[1] GitHub - sharemind-sdk/secrec: Sharemind SecreC Compiler and Analyzer `https://github.com/sharemind-sdk/secrec`

[2] flex: The Fast Lexical Analyzer `http://flex.sourceforge.net/`

[3] Bison - GNU Project - Free Software Foundation `https://www.gnu.org/software/bison/`

5.2 Finding and instantiating matching definitions

The procedure that searches for matching operator definitions first looks for monomorphic definitions (without template quantifiers) with the name of the operator in the active symbol table. This fits the search procedure described in Section 4.7. A monomorphic definition has no domain quantifiers so its domain weight component is less than that of templates with a domain quantifier. No quantifier variables appear in the parameters so it also precedes all templates with type quantifiers. This avoids template unification when a monomorphic definition exists. Monomorphic definitions are ordered according to the last scoring component in Section 4.7: the number of implicit classifications and reshapings. It is an error when multiple definitions share the best score.

When no monomorphic definition matches, operator definition templates with the name of the used operator are looked for. Each definition template is unified with the types used in the expression. If unification fails the definition is not considered. The best score and the set of templates with the best score is updated when iterating over the templates. It is again an error if multiple templates share the best score.

The procedure body of the best matching template is copied, the quantifier variables are bound to the unified types in the scope of the copied procedure definition and the procedure is type checked and compiled. We use something similar to the SFINAE (*substitution failure is not an error*) principle in C++. If type checking fails (for example, if the template body is not well-typed), the reported error claims that there is no matching definition. The error report should not blame the template definition itself because the error may have been caused by the quantifier variable substitution and the template may be valid for other substitutions. Template instances are cached so if the same template is instantiated with the same parameters multiple times, it will only be type checked and compiled once.

5.3 Types of integer and floating point literals

A problem in the full SecreC language is that there are multiple public integer and floating point types. There are 8-, 16-, 32- and 64-bit versions of unsigned and signed integers. This means that it is not clear which public numeric type to assign to a literal such as `42`. Integer and floating point literals are assigned the abstract `numeric` type which is a subtype of all numeric types. When comparing types according to the partial ordering described in Section 4.1, we consider $(\texttt{public}, \texttt{numeric}) \sqsubseteq (d, t)$ for all pairs (d, t) for which $\text{public}(\text{kind}(d), t)$ is a numeric type.

In an arithmetic expression, operands with the abstract `numeric` type must be assigned a concrete numeric type. If the expression has a context, such as `pd_shared3p uint x = y + 1;`, the literal `1` is assigned the corresponding public data type of the context data type. In this case it is $\text{public}(\texttt{shared3p}, \texttt{uint}) = \texttt{uint}$. In a binary expression with operand types $(d_1, t_1), (d_2, t_2)$ the least upper bound of the operand types $(d', t') = (d_1, t_1) \sqcup (d_2, t_2)$ is computed and the operands with type `numeric` are assigned data type $\text{public}(\text{kind}(d'), t')$. Essentially, this means that the literal is assigned the corresponding public type of the other operand. If the upper bound is $\top$ or there is no corresponding public data type an error is reported.

As an example, let us assume the following PDK definition and domain declaration:

```
kind shared3p {
    type bool;
    type int64;
    type fix64 { public = float64 };
}
domain pd_shared3p shared3p;
```

The following table illustrates how the `numeric` data type is replaced in this case:

First operand	Second operand	Assigned type
`public numeric`	`pd_shared3p int64`	`public int64`
`public numeric`	`pd_shared3p fix64`	`public float64`
`public numeric`	`pd_shared3p bool`	$\top$

5.4 Implementation details

In this section we describe details about implementing the changes in the SecreC compiler.

Data types defined in a protection domain kind definition are represented as a C++ data type. A single value is created for each data type name and added to the global symbol table. The data type representation includes a mapping from PDK to public type (to implement public(k, t)). If a data type is defined in multiple protection domain kinds, the corresponding public types are added to the mapping. The user-defined data type is also added to a map in the PDK value. The PDK is added to the global symbol table. This is necessary to check if a type belongs to a PDK (required by the variable declaration rule in section 4.3) or when processing a domain declaration.

Multi-dimensional arrays are currently represented as a continuous block of memory with separate symbols containing the number of elements and the sizes of each dimension. When an array with dimension higher than one is reshaped into a vector to pass to a operator/cast definition procedure, it is copied and new shape symbols are created. When a scalar has to be converted to a vector, a vector is allocated that is as large as the other operand (in the case of binary operators) or has one element (unary operators). The scalar is assigned to all elements of the vector. Calling a procedure has an overhead because SecreC is a call-by-value language so the supplied parameters are always copied. This overhead will be reduced in the future when function and cast/operator calls will be inlined.

When a value has to be implicitly classified (for example, an operand in a binary expression), a classify node is inserted into the abstract syntax tree. This node is only used by the compiler and is never created by the parser.

Before a binary operator is called, we check that the shapes of the supplied operands match. After the call, we check that the size of the result is the size of the largest parameter because it is assumed that operators work point-wise. These conditions can not be checked statically because the size and shape of an array is dynamic and implementations of system calls are external to the language.

The result of the call to the operator/cast procedure is copied if the dimensionality does not match the dimensionality of the definition (for example, when a definition on vectors is used for point-wise matrix multiplication). The copy is given shape symbols of the larger input. This is safe because at this point it has been checked that the size is as expected.

In addition to the restrictions on data types in the operator definition typing rule, the compiler also disallows operator definitions on strings and record types which are supported in the full SecreC language. This condition was omitted from the typing rule because it is trivial to check in practice while adding record types to the language would have complicated the grammar and typing rules. The subset of SecreC used in the thesis does not have dimensionality quantifiers. In the implementation, dimensionality quantifiers are forbidden in operator definitions.

6 Practical applications

6.1 Practical implications for users of SecreC

SecreC already supports different protection domain kinds. The extensions of this thesis make using them more convenient. Currently, when a program is executed, all system calls are linked to their C implementations. The linker reports an error if a system call which is expected to implement an arithmetic operation is missing. Debugging will be difficult in this situation because the programmer does not know where the missing operator was used if the program consists of thousands of lines of code [4]. The changes of this thesis allow the compiler to give compile time errors when an arithmetic expression uses an operator that is not supported by the PDK. This will make debugging easier because errors reported by the compiler include the source code location of the incorrect expression.

SecreC supports different PDKs because they have different performance characteristics, installation requirements (e.g. the number of parties) and security guarantees. By considering arithmetic as a black box defined by the PDK module, the programmer can write the algorithms and logic without considering the differences. For example, if the programmer has written a linear regression function with the following signature:

```
template<domain D>
D float[[1]] linearRegression(D float[[2]] independent,
                              D float[[1]] dependent)
{
    // body omitted
}
```

then the function can be called with inputs from different protection domains. The current implementation of the compiler makes this dangerous because if the function uses operations that are not supported in some PDK, the program will give a runtime error that will be difficult to debug. With the extensions of this thesis the function body will be checked after it is instantiated with a concrete protection domain. If it is instantiated with a domain that misses some operator that is required in the function body the compiler will give an error. Being able to use the same functions with different PDKs means that existing code can be reused in different projects with different requirements. If a project requires the highest level of security, a PDK with active security can be used. If the passive adversary model is acceptable, the efficient three-party additive secret sharing PDK can be used. If finding three independent project partners who will host servers is difficult, a two-party PDK can be used. This allows for high code reuse and flexibility. Due to overloading, SecreC also allows a polymorphic version of a procedure to co-exist with a monomorphic version optimised for a specific PDK. The monomorphic version will be preferred if it matches the types of the inputs.

The extensions of this thesis allow creating new primitive private data types. Alternative data types are useful because some protocols have more efficient implementations when using a specific representation. For example, the current implementation of SecreC has special `xor_uint8`, `xor_uint16`, etc data types. Values of this types are shared bitwise and they are always private. That is, to secret share x, random values x_1, x_2 are generated and x_3 is computed such that $x = x_1 \oplus x_2 \oplus x_3$, where $\oplus$ is the XOR operation.

[4] This problem is exacerbated by the lack of debugging tools.

Bitwise operations and comparisons are more efficient on this representation but addition and multiplication is less efficient so values are sometimes converted into this type before sorting and converted back to additively secret-shared values after sorting. These data types can now be described in SecreC without changing the compiler.

Other unusual data types may be required when optimising computations on real numbers. The additive secret sharing PDK has an implementation of floating point numbers which uses a representation similar to IEEE 754 [KW14a]. There is also another set of protocols that uses a combination of secret sharing and Yao's garbled circuits that supports the full IEEE 754 specification but is slightly less efficient [PS15]. In some cases, floating point numbers are prohibitively inefficient and fixed point numbers are required for implementing efficient algorithms on real numbers. Currently, SecreC only has `float32` and `float64` data types which means only one representation can be used at the same time. It would be useful to have different representations that procedures can use depending on whether the programmer needs IEEE 754 compatibility, accuracy or maximum efficiency. These three sets of floating point protocols could all be used by adding special floating point data types.

It is not reasonable to add every special data type to the compiler. The kind definition feature described in this thesis allows private data types such as the `xor_uint` types and different floating point types to be added to the language without changing the compiler.

6.2 Proposed structure for the SecreC standard library

In this section we describe the proposed structure of the SecreC standard library while taking into account the changes of this thesis.

Every protection domain kind should be defined in a SecreC library module along with its arithmetic, logic and relational operator definitions and type conversion definitions. For example, a module named `shared3p` defines a PDK with three-party additive secret sharing and `shared2p` defines a PDK with two-party additive secret sharing.

Algorithms that rely on standard arithmetic operations should be programmed using domain polymorphism and organised into a module with a descriptive name. If more efficient implementations can be written for some PDK then they should be organised into a module with the name of the PDK prepended to the name of the generic module. For example, matrix operations that use standard arithmetic can be implemented in a module named `matrix` while an optimised matrix multiplication protocol implemented for the `shared3p` PDK can be used to write a more efficient matrix multiplication procedure in the module `shared3p_matrix`. Currently SecreC does not support hierarchical module names so we use the `shared3p_` prefix but if support for hierarchical module names was implemented, the `shared3p` version would be named `shared3p.matrix`.

Having the definition of a PDK in a single SecreC module is also useful for documentation purposes. Currently, Doxygen [5] is used to generate documentation for the SecreC standard library. The documentation generator could be extend by writing a program which parses the module defining a PDK and finds the kind definition and all the operator and cast definitions. This information can be compiled into a table in the documentation which lists the supported private data types, the operators supported on each data type and the conversions supported between different data types. This documentation would reduce the need for trial and error because the programmer can see which operations in a PDK are supported before writing any SecreC code.

[5] Doxygen `http://www.stack.nl/~dimitri/doxygen/`

7 Future work

If a protection domain kind supports all operations on all data types defined in the kind, it is sufficient to write a single definition for each operator. For example, if we have defined the kind `shared3p` then we can define multiplication for all data types in `shared3p` using the following operator definition template:

```
template<domain D : shared3p, type T>
D T[[1]] operator + (D T[[1]] x, D T[[1]] y) {
    __syscall("shared3p::add_$T\_vec",
              __domainid(D), x, y, y);
    return y;
}
```

In the system call name string, `$T` is replaced with the name of the data type bound to `T`. This assumes that all the system call names have the same structure and the number and order of arguments is the same. The problem is that even if just one data type does not support addition, like booleans, we have to write separate definitions for all data types. It would be more convenient if we could constrain the domain of the quantifier `T`. This could be done using type predicates. The user can specify the types for which the predicate holds. Template definitions can then use the predicate to constrain the domains of quantifiers. This is similar to the C++ extension called Concepts Lite [SSDR13] which allows limiting the set of accepted template parameters. For example, by using type predicates, the definition of addition could be written as:

```
template<domain D : shared3p>
predicate IsNumeric<D, int>;
template<domain D : shared3p>
predicate IsNumeric<D, float>;

template<domain D : shared3p, type T> requires IsNumeric<D, T>
D T[[1]] operator + (D T[[1]] x, D T[[1]] y) {
    __syscall("shared3p::add_$T\_vec",
              __domainid(D), x, y, y);
    return y;
}
```

Since the predicate is nominative, we can reuse it when defining other operators. Type predicates could also benefit SecreC programming in general. Currently the standard library contains polymorphic procedures that actually only work on certain types. For example, a procedure for algebraic matrix multiplication is defined but it only works on numeric data types. If a programmer accidentally calls the procedure template with boolean matrices they will get an error due to some expression in the body of the procedure. They should instead receive an error stating that no matching procedure is found. A workaround is to define the procedure template with a different name and call it from monomorphic procedures which are only defined for the supported data types. With type predicates we could just write a single procedure template as follows:

```
template<domain D : shared3p,  type T>
D T[[2]] matrixMultiplication(D T[[2]] x, D T[[2]] y)
    requires IsNumeric<D, T>
{
    // body omitted
}
```

Another similar and useful feature would be type level functions. Currently operator templates with a private and public argument have to be written as follows:

```
template<domain D : shared3p, type T, type TPub>
D T[[1]] operator * (D T[[1]] x, TPub y) {
    // body omitted
}
```

We can not use `T` for both the private and public parameter since the public type corresponding to `T` may not be the same. For example, a definition that used `T` for both parameters would match when the operands are `pd_shared3p int` and `int` but not when they are `pd_shared3p fix` and `float`. In the case of operator definitions, using two type variables is sufficient but we may want to use the corresponding public type of a variable `T` in the body of a procedure template where the public type does not appear in the parameters. This is illustrated in the following example:

```
template<domain D : shared3p, type T>
T[[1]] tabulate(D T[[1]] x) {
    T[[1]] shuffled = declassify(shuffle(x));
    T greatest = max(shuffled);
    T lowest = min(shuffled);
    uint length = (uint) (greatest - lowest + 1);
    T[[1]] res(length);
    for (uint i = 0; i < length; i++) {
        res[i] = sum((T) (shuffled == lowest + (T) i));
    }
    return res;
}
```

This procedure counts occurences of values in a vector. Counting after declassifying is more efficient than doing it privately. It leaks the values but we do not know where each value originated because the input vector is shuffled privately. Unfortunately, this procedure will not always type check. If we apply it to a vector with type `pd_shared3p fix64[[1]]`, we get a type error because `fix64` does not exist in the public domain which is required on the first line of the body. The workaround is to add a type quantifier for the corresponding public type:

```
template<domain D : shared3p, type T, type TPub>
TPub[[1]] tabulate(D T[[1]] x) {
    // body omitted
}
```

If the public type does not appear in the signature of the procedure then we have to add a parameter to the procedure which is unused in the body (such as `TPub proxy`) and is only used to bind the correct type.

The type function `PublicType<D, T>` would evaluate to the corresponding public type of `T`. We could then write the definition like this:

```
template<domain D : shared3p, type T>
PublicType<D, T>[[1]] tabulate(D T[[1]] x) {
    // body omitted
}
```

If the user was able to define type functions, it would also be useful in general. The SecreC standard library contains procedures that take integers as inputs but need to convert to floating point numbers due to division or a special function (e.g. logarithm). We could write the arithmetic mean procedure like this:

```
typefun FloatFriend<int32> = float32;
typefun FloatFriend<int64> = float64;

template<domain D, type T>
D FloatFriend<T> mean(D T[[1]] x) {
    return (FloatFriend<T>) sum(x) /
        (FloatFriend<T>) size(x);
}
```

Currently a private operator definition is called for each arithmetic expression. Procedure calls have overhead and procedures are currently optimised independently. A useful optimisation would be to inline procedure calls. This would reduce procedure call overhead and would allow the optimiser to work on the whole block where the arithmetic/cast expression is used. For example, procedure parameters are passed by value which means they are copied. Intermediate copies could be reduced by an optimisation pass if they are unnecessary. Internally, all private values are vectors with separate variables describing the shape of the the private value. When a system call is called, the reshaping from e.g. matrix to vector is actually unnecessary since the system call only sees the vector. Reshaping could be possibly removed.

When defining a data type in a protection domain kind it is possible to give the size of the data type as its serialised. Currently the compiler parses this information but ignores it. It is always assumed that system calls for memory management (allocation, deallocation, assignment, indexing etc) exist and are named according to a specific pattern. System calls in SecreC have a significant overhead which is a problem when dealing with large data. We expect that generating memory management code using the data type size parameter would be a significant optimisation. The overhead is especially bad when reordering values in a private vector according to some permutation. This occurs in many useful algorithms such as privacy preserving database table join. Implementing this optimisation would also require work in the protection domain kind module implementing the arithmetic protocols. All the current modules represent values as C++ objects but this change would require system calls to receive pointers to byte arrays.

8 Conclusion

Secure multi-party computation (SMC) allows different parties to compute a function on shared inputs without leaking their values to the other parties. SecreC is a programming language designed for writing privacy-preserving programs using SMC. SecreC relies on the definition of a protection domain kind (PDK) which describes operations for performing arithmetic with private values. Different cryptographic techniques can be used to implement a PDK. The choice of PDK depends on the application because different techniques have different security guarantees, performance characteristics and deployment requirements. There is no single best PDK for all use cases. Due to this, SecreC supports multiple PDKs. This thesis allows the set of data types supported by a PDK and the arithmetic, relational, logic and type conversion operators on these data types to be defined in the SecreC language. This makes it easier for the programmer to use different PDKs because the compiler has information about PDKs which can be used during compiling to check whether types and operators used in a program exist in the PDK.

Support for defining private data types in a PDK allows extending the SecreC language with non-standard private data types. This gives the programmer further flexibility by allowing to choose between different floating point representations that have a different balance between accuracy and efficiency or by using private data types that have some efficient operators to optimise the internals of algorithms.

In this thesis, the author developed a formal type system of a subset of SecreC with the proposed extensions. The author also changed the type checker and code generator of the SecreC compiler to accomodate the new features and described practical applications of the extensions, open problems and possible solutions.

References

[ABPP15] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. Cryptology ePrint Archive, Report 2015/1039, 2015. `http://eprint.iacr.org/`.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 257–266, New York, NY, USA, 2008. ACM.

[BKK+16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *PoPETs*, 2016(3):117–135, 2016.

[BKLS14] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014. `http://eprint.iacr.org/`.

[BLR14] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-Polymorphic Programming of Privacy-Preserving Applications. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 53–65. ACM, 2014.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, pages 247–264, 2003.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for Automating Secure Two-party Computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 451–462, New York, NY, USA, 2010. ACM.

[Jag10] Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[KW14a] Liina Kamm and Jan Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. *International Journal of Information Security*, pages 1–18, 2014.

[KW14b] Toomas Krips and Jan Willemson. Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations. In *Proceedings of the 17th International Information Security Conference, ISC 2014*, volume 8783 of *LNCS*, pages 179–197. Springer, 2014.

[LDDAM12] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. *SIGPLAN Not.*, 47(9):189–200, September 2012.

[LR15] Peeter Laud and Jaak Randmets. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1492–1503. ACM, 2015.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376, 2015.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 186–197, May 1998.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - a secure two-party computation system. In *In USENIX Security Symposium*, pages 287–302, 2004.

[MSSZ12] John C Mitchell, Ritu Sharma, Dumitru Stefan, and Jeramy Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 45–60. IEEE, 2012.

[Mye99] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[Nat01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). *Federal Information Processing Standards Publications*, FIPS-197, 2001.

[Nie09] Janus Dam Nielsen. *Languages for secure multiparty computation and towards strongly typed macros*. PhD thesis, PhD thesis, University of Aarhus, Denmark, 2009.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 223–238, 1999.

[PS15] Pille Pullonen and Sander Siim. Combining secret sharing and garbled circuits for efficient private ieee 754 floating-point computations. In *Financial Cryptography and Data Security - FC 2015 Workshops, BITCOIN, WAHC and Wearable 2015, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *LNCS*, pages 172–183. Springer, 2015.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[Ris10] Jaak Ristioja. An analysis framework for an imperative privacy-preserving programming language. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[SKM11] Axel Schröpfer, Florian Kerschbaum, and Guenter Mueller. L1 - An Intermediate Language for Mixed-Protocol Secure Computation. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference. COMPSAC'11*, pages 298–307, 2011.

[SSDR13] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts lite: Constraining templates with predicates. *Undated pre-publication draft posted*, pages 02–08, 2013.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS '08. 23rd Annual Symposium on*, pages 160–164, Nov 1982.

[ZBA15] Yihua Zhang, Marina Blanton, and Ghada Almashaqbeh. Implementing support for pointers to private data in a general-purpose secure multi-party compiler. *CoRR*, abs/1509.01763, 2015.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 813–826, New York, NY, USA, 2013. ACM.

Non-exclusive licence to reproduce thesis and make thesis public

I, Ville Sokk (date of birth: 29th of March 1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

An improved type system for a privacy-aware programming language and its practical applications

supervised by Dan Bogdanov and Jaak Randmets

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **19.05.2016**