UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Roman Jagomägis

SecreC: a Privacy-Aware Programming Language with Applications in Data Mining

Master's Thesis (30 EAP)

Supervisor: Dan Bogdanov, MSc

Author: …….. "……." May 2010
Supervisor: …….. "……." May 2010

Allowed to defend
Professor: …… "……." …... 2010

TARTU 2010

Contents

1. Introduction

1.1. Problem statement

Information is a resource of vital importance to our society. Depending on its accuracy, meaning, potential or particular intended use, the information can be economically, strategically, medically, socially or otherwise valuable to one or more entities. By giving away some information one inevitably puts his interests at risk. Moreover, as a result, other individuals and organizations may suffer as well. For this reason the preservation of data privacy has become an important task.

It is often possible to gain significant added value by combining confidential information gathered from different sources. For instance, medical institutions maintain disease or DNA repositories to find new cures, or companies can collaboratively find out new knowledge about their customers. However, this leads to serious security issues, as sensitive data may be misused by those who collect and process it.

The processing of such data is regulated by laws [1, 2], but the law does not provide sufficient technical and organizational details for achieving the desired level of privacy. An attempt to address these details is made by development of standards [3, 4] for protection of information. Furthermore, the researchers in information security have proposed several technical solutions to deal with the problem. There exist frameworks like SHAREMIND [5], VIFF [6] and FAIRPLAYMP [7] that base on the idea of secure multi-party computation (MPC) allowing to process sensitive data with strong security guarantees. These frameworks are used to store the data, construct secure, though complex data mining algorithms and run them on the data without compromising the security of input sources. Nevertheless, these solutions have not reached practical real-life systems due to complicated deployment of business logic using such frameworks.

We compared the three mentioned secure multi-party computation frameworks and their pros and cons. The SHAREMIND system appears to be the most promising regarding secure large-scale applications. Still, despite the good performance and a comfortable underlying architecture of SHAREMIND, the system still relies on the assembly language. The development of complex privacy-preserving algorithms using low-level instructions is very difficult, tedious and error-prone. Large algorithms often rely on big amount of flow control. In assembly this means many labels with conditional and unconditional jumps. The problem is amplified by the absence of methods, making it hard writing reusable code. Additionally, there are no means for matrix management which is often a necessary feature in complex data mining algorithms. Joining all the named drawbacks the resulting assembly code is usually far from human-friendly as it is completely unreadable and unmanageable.

The main goal of this thesis is to make the development of advanced privacy-preserving algorithms as simple as possible. The goal is to hide most of the cryptographic building blocks from the programmer enabling to concentrate on the business logic. To achieve this goal we will design a higher level privacy-aware programming language

called SECREC (pronounced 'secrecy'). The name was inspired by the security aspect of the language.

1.2. Outline

This thesis describes the concepts and principles of our privacy-aware programming language and its compiler, and discusses the practical use-cases for our solution. The work is structured into chapters as follows:

- Chapter 2 gives an overview of the underlying execution environment. The chapter describes the SHAREMIND virtual machine and its important properties. We also describe the hybrid execution model that the language will be based on. Finally, we discuss the improvements made to SHAREMIND since its paper [5] was published.

- Chapter 3 describes the design of the SECREC language with security in mind. We give an overview of the major language features such as the structure of the program, data types and constants, identifiers, variables, expressions, vectors and matrices, functions and flow control structures. In the end we also describe the standard library of the language.

- Chapter 4 is dedicated to the compiler of SECREC language. We describe how various SECREC structures are translated into the SHAREMIND assembly.

- Chapter 5 elaborates on writing privacy-preserving algorithms using SECREC. The chapter discusses the aspects the programmer has to keep in mind when developing the algorithms for SHAREMIND with SECREC.

- Chapter 6 presents several practical use cases of our new privacy-aware language. We present novel privacy-preserving versions of four data mining algorithms written in SECREC and executed on SHAREMIND. The algorithms are: Histogram, Apriori, Eclat and Hybrid-Apriori. We also evaluate the security and performance of these algorithms.

- Chapter 7 describes the related work and future plans concerning our solution.

1.3. Author's contribution

In this section we list the author's original contributions to this thesis. Chapter 2 gives an overview of the underlying execution environment – the SHAREMIND platform. The author has made numerous improvements to SHAREMIND virtual machine and its assembly language in collaboration with Dan Bogdanov. The main contributions of the author include: the design of the privacy-aware high-level programming language SECREC discussed in chapter 3; the construction of the SECREC compiler described in chapter 4; a programmer's guide for developing algorithms with SECREC presented in chapter 5; and the design, implementation using the SECREC language, and verification of four privacy-preserving data

mining algorithms documented in chapter 6. The compiler and case studies are substantial contributions in size, difficulty and importance. By solving several practical data mining tasks in a privacy-preserving way the author shows, that the SECREC language and its compiler are functional and sufficient to solve real-life problems. As an additional confirmation, two papers [8, 9] have been submitted based on this work. The first paper concentrates on building privacy-preserving applications in SHAREMIND's hybrid execution model. It also introduces the SECREC language and discusses on principles and techniques for developing secure applications. The second paper analyses SHAREMIND as a practical toolkit for privacy-preserving data mining, and gives an in-depth discussion of the three frequent itemset mining algorithms presented in chapter 6 of this thesis. Finally, the author has been of active help in the "related work" projects mentioned in chapter 7.

2. Execution environment and principles

2.1. Secure multi-party computation

The aim of secure *multi-party computation* (MPC) is to enable a number of networked players to carry out distributed computing tasks on their private information while under attack by an external entity – the adversary – or by a subset of malicious, colluding players. The purpose of the attack is to learn the private information of non-colluding, honest players or to cause the computation to be incorrect.

A traditional example of MPC is the *millionaire problem* introduced by Andrew C. Yao [10]. It involves a number of millionaires who want to find out who is richer without disclosing the precise amount of their wealth.

2.2. Overview of existing solutions

Before any solution can be created it is necessary to analyze the problem in more depth and select a reasonable basis for further elaboration. Whenever we construct algorithms we are constrained by the computation system that executes them. Therefore, the underlying platform defines the terms and conditions used to express those algorithms. The more sophisticated and scalable the computation system is the more advanced algorithms it can execute. Hence, a preliminary assessment of available MPC systems is required.

Most of secure multi-party computation systems are built to solve very specific tasks, which makes them not flexible enough to overcome more general challenges. Thus such systems will not be observed as a potential basis for our solution. However, there exist some well-defined secure MPC frameworks capable of carrying out complex privacy-preserving computations on a more general level. In the following we will briefly review three frameworks and choose the most promising one in terms of security, architecture, scalability and performance as an underlying computation system for our privacy-aware language.

FAIRPLAYMP. This is an extended multi-party version of the function evaluation system FAIRPLAY [11]. It combines a constant round protocol by BMR [12] with the protocol by BWG [13] with security against a passive adversary.

The programs are written in a high-level language SFDL (Secure Function Definition Language) that is used to describe a predefined number of players. The system compiles the SFDL program into a low-level Boolean circuit. The compiler inlines the functions (direct or indirect recursion is forbidden), unrolls the loops (the number of loop iterations must be a compile-time constant) and turns all operations into a series of primitive hardware operations that operate on single bits only. After peephole optimization and dead code removal the Boolean circuit is securely evaluated revealing nothing but the outputs.

Such restrictions reveal several disadvantages of the system. The system is not dynamically scalable, as the number of input players is predefined before compilation. Also, the performance hit is inevitable as the Boolean circuits must compute every possible runtime step. Because of the nature of Boolean circuits there is a huge performance gap be-

tween an efficient array access when the index is a compile time constant and the non-efficient access otherwise. According to the FAIRPLAYMP paper [7] the auction with 10 bidders completed in over 33 seconds which is a good result but not enough for real-time large-scale applications. The performance drops linearly w.r.t. the size of the circuit and security parameter and partially quadratically w.r.t. the number of players.

VIFF. The Virtual Ideal Functionality Framework [6] is a python MPC library with building blocks for developing cryptographic protocols with n parties. All the computation is conducted in arbitrary finite fields $\mathbb{Z}_p$ or $\boldsymbol{GF}(2^8)$. Based on protocols presented in [14] the library provides security against an active adversary corrupting less than $n/3$ parties.

VIFF aims to be usable by parties connected by real world asynchronous networks such as the Internet. To make this possible the library provides automatic use of deferred values. During the evaluation of the computational circuit, VIFF uses a system of callbacks to make sure that the next operation is executed as soon as required data is available. This technique allows VIFF to perform automatic parallelization.

The library is well scalable and reasonably fast, allowing development of real life applications. The tests [14] show that VIFF is capable of doing 770 secure multiplications per second with 4 parties, which should cover many practical applications.

SHAREMIND. This is another interesting secure multi-party computation framework capable of processing secret data without disclosing. It uses the additive secret sharing scheme in the ring $\mathbb{Z}_{2^{32}}$ and is proven to give a strong privacy guarantee in the honest-but-curious security model. Currently the system consists of three parties connected over secure asynchronous network channels and tolerates one passive corrupted party. With four parties it would be possible to achieve security with one actively malicious party. In a five-party implementation two passive corrupted parties are allowed.

The system resembles a hybrid virtual machine – it contains a processor unit that can sequentially and in parallel execute a number of operations on secret or public data. In fact, it allows the creation of algorithms to describe the way the data should be processed. SHAREMIND already has an assembly programming language [15] to serve this purpose. Each MPC operation in SHAREMIND is implemented as an assembly instruction. It also supports vectorized operations allowing parallelization of algorithms significantly boosting their performance.

Although the SHAREMIND system currently supports only three computation parties, there is no limitation on the number of data donors. All the data is secret-shared and distributed among the computation parties not motivated in opening the values. Moreover, the system performs significantly better comparing to FAIRPLAYMP and VIFF. According to recent benchmarks, in good conditions SHAREMIND can do about 10^8 secure multiplications in 70 seconds. This means that complex enough data processing algorithms can be executed nearly in real time.

2.3. The SHAREMIND machine

Structure. SHAREMIND is a practical multi-party computation framework based on share computing. The framework has secure MPC protocols for performing private addition, multiplication, comparison and individual bit extraction operations on 32-bit integers or integer vectors. The protocols are designed to be universally composable, which allows running them sequentially and in parallel without compromising security. SHAREMIND makes use of this fact by implementing a distributed virtual machine [5] with a range of machine code instructions [15] for public and private computations.

The virtual machine currently consists of three computation nodes – the miners. The structure of a single miner is depicted on Figure 1. A miner consists of three main modules: the database, the network node, and the central processing unit. These are used to handle the secure data storage, control program execution and execute multi-party computation protocols to perform private operations. While data management and flow control is handled on every miner locally, the global execution is synchronized with miners and remote users – the clients – by using network protocols.

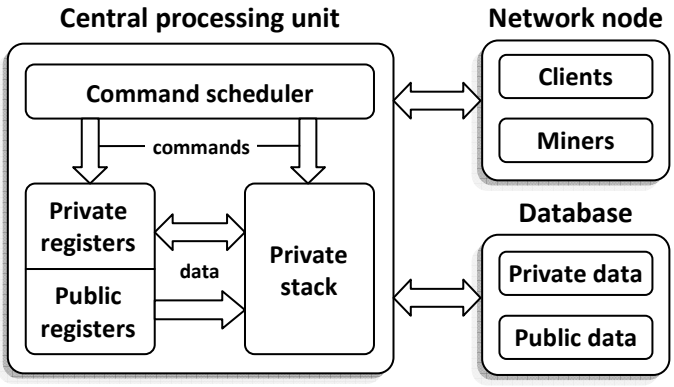

Figure 1. The structure of a single node of the Sharemind virtual machine.

The central processing unit (CPU) is the most important component in the system. The command scheduler block inside the CPU runs SHAREMIND machine code that determines the program execution flow. The named registers and the private stack are used to store intermediate results during the runtime. While registers may contain public and private data, the private stack is used strictly for private operations. All private data – whether in registers, database or on the stack – is stored in secret-shared form. Before running a private operation the programmer must make sure that the input data is pushed on the stack. After the command completes its execution, the results can be retrieved from the top of the stack. Hence, SHAREMIND can be considered a modified stack machine.

Deployment model. The most suitable usage scenario for SHAREMIND incorporates several organizations who want to perform computations on private data without disclosing. They choose three representatives among them to host the well-guarded miner nodes which will be mining the data. These hosts should have enough motivation to commit resources to complete the computation, but not to breach the privacy of the data.

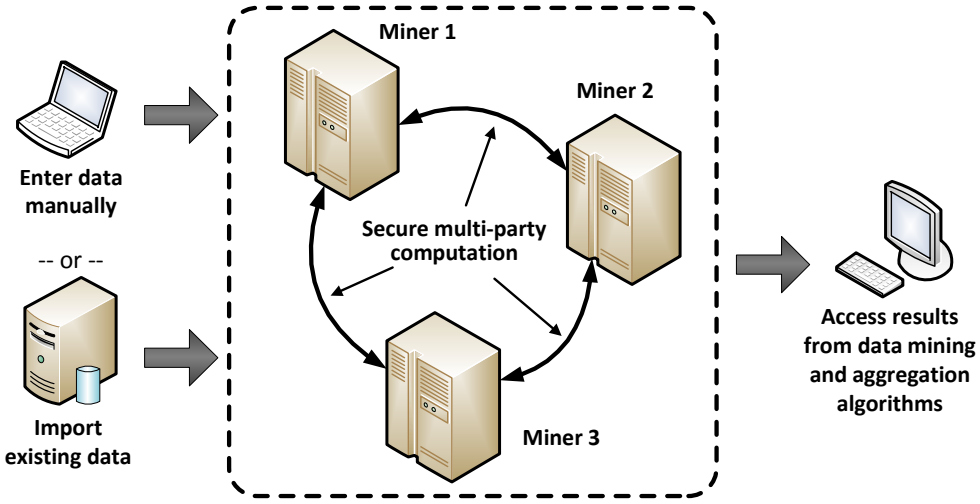

Figure 2. The Sharemind deployment model.

Given a setup of three networked miners a typical privacy-preserving application involves three phases as illustrated on Figure 2. First, the data donors worldwide distribute their data values into shares right at the source. Each share is then sent to a different miner, where it is stored in the database. This guarantees that nobody but the source will see the original data. During the computational phase, the miner nodes jointly create and modify shared values to compute desired outcomes. Finally, the shares corresponding to outputs are published and respective values are reconstructed.

Security. SHAREMIND uses the additive secret sharing scheme that assures that the entries stored by individual miners are completely random bit strings. Therefore, a data donor does not have to trust any of the miners. Instead, the donor must believe, that miners as a group obey certain rules. Currently, in the three-miner setting Sharemind tolerates attacks where a single miner node is passively corrupted during the computations.

All SHAREMIND security guarantees were proved [5] in the information-theoretical setting, where each pair of participants is connected with a private asynchronous communication channel so that a potential adversary can only delay or reorder the messages without reading them. It is also assumed that the communication links are authentic.

2.4. The hybrid execution model

Every piece of information can be a secret or common knowledge. Let us denote secrets as private data, and common knowledge as public data. It is clear that in order to guarantee the privacy of our secrets these kinds of data may not be mixed together in the public computation environment (Figure 3). Otherwise, one could witness the secrets while working with common knowledge. The lack of control mechanisms for limiting the flow of private data into the public data domain is a great security risk that has to be taken care of.

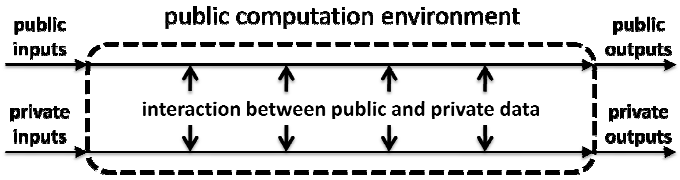

Figure 3. The flow of public and private data in the standard execution model.

To reduce security risks there has to be a distinction between how public and private data is used. It is possible to construct an alternative hybrid execution model, where public and private data is processed in separate environments (Figure 4). The environments can be two computation systems that are specially built to handle the specific type of data. In a setup like this it is crucial to limit the publishing of information derived from private data into public data domain. While original secrets should never be published, some aggregated values such as a median or a sum can usually be published without leaking too much information. Therefore, some strict means for controlling the data flow are mandatory.

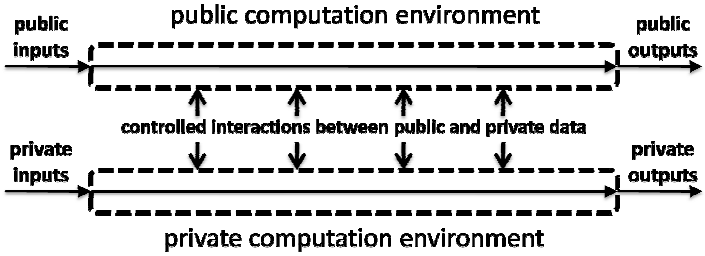

Figure 4. The flow of public and private data in the hybrid execution model.

The concept of the hybrid execution model is a generalization of the SHAREMIND virtual machine which, in fact, may be considered a hybrid virtual machine. While the public data and basic flow control is handled in the assembly interpreter (the public virtual machine), the private data is processed in the SHAREMIND processing unit (the private virtual machine) by core privacy-preserving computation protocols. The data can be processed using assembly instructions for all SHAREMIND protocols, the public data manipulation and flow control. While the public data is stored and processed in public memory, the private data is processed on a special private stack in secret-shared form. Instructions for core protocols read the input from the stack, execute corresponding protocols and push the results back on the stack.

The described system contains all the necessary parts for the more secure model. However, it lacks the ease of algorithm implementation and the means of data flow markup. The design of SECREC aims at simplifying the programming task and preventing the developer of privacy-preserving data mining algorithms from making trivial privacy leaks. Furthermore, SECREC enables security analysis of private data flow by tracking code locations, where private data is explicitly declassified.

It is important that all decisions during the program flow in the hybrid execution model have to be done on the public data. Otherwise, it would be necessary to publish the

observable secret value, which is a security threat. It is up to the algorithm designer to minimize the number of required declassifications.

2.5. Vectorization

The SHAREMIND framework is known to be more efficient when executing several operations in parallel [5, 9]. This is due to the internal architecture of SHAREMIND – during each operation the system gets private inputs from the private stack, and then communicates over the network to perform some computations. Generally, the less a system communicates over the network, the faster it is. Intuitively, to minimize the network traffic, we have to send a smaller number of bigger packets instead of a large number of smaller packets. Up to a certain size, a bigger packet takes almost the same amount of time to transfer than a smaller packet. After that, sending time is linear w.r.t. packet size. This is due to the bandwidth overhead caused by the headers of each packet sent over the network.

On the basis of these properties it is advantageous to execute the same operation on a vector of data rather than on single values. The extensive use of operation vectorization is one of the main objectives in the SECREC design. The language must support intuitive and simple means of operation parallelization and allow taking advantage of it by optimization of the algorithms.

2.6. Improvements to the SHAREMIND framework

Since the Sharemind paper [5] release the framework has received a number of changes and improvements.

1. We introduced the Sharemind assembly language [15] that effectively linked the functionality of secure MPC protocols to mnemonic instructions with parameters. The language allows flow control through conditional and unconditional jumps.
2. Together with the assembly language the public and private named registers were introduced. The public registers store public data like loop counters and public results, while the private registers hold some intermediate secret-shared values.
3. The database inside the virtual machine was improved to support named columns of various data types.
4. The interfaces to the virtual machine were improved to allow several simultaneous users and support more complex interactions with SHAREMIND.
5. New secure multi-party computation protocols were developed to improve the speed and security of the system.
6. The instruction set of the machine was extended to support the newly created SECREC programming language.
    a. We enhanced support for boolean and integer vectors.
    b. We added support for functions by creating global and local register environments for both public and private data. Local environments can be switched by pushing and popping them from a special stack leaving the

global environments intact. This technique enables communication with functions through parameters and return values. For tracking function instances the `call` and `return` instructions were added to the virtual machine. The `call` instruction pushes the current instruction pointer onto the stack of return pointers and jumps to the designated label. The `return` instruction pops the return pointer and jumps to it.

c. We addid the `unregister` instruction to remove an unneeded register from the environment. This is used in several compilation schemes, for example returning value. The return value is stored into a specific global register before the local register environment gets destroyed after completing the function. After the register environment of the caller is popped from the stack, the returned value is moved to the target local register from the temporary global one. The temporary global register is then removed.

3. Language design

3.1. The structure of a program

The SECREC language is a procedural imperative domain-specific language inspired by C. Generally speaking, a SECREC program is a list of statements that are executed one after another until the end or an error is reached. However, there are some syntax rules.

A program consists of a number of variable declarations and function definitions. In the very beginning of a program the developer is allowed to declare and initialize global variables. A variable declaration is one possible type of statement, and each statement in the program ends with a semicolon '`;`'.

Variable declarations are followed by function definitions. Every program must have at least one function called 'main'. This is the first function to be executed after possible global variable declarations.

Program code may also contain line or block comments which are omitted by the SECREC compiler and do not affect the program flow in any way. Line comments begin with '`//`' characters and consume everything up to the end of the same line. Block comments reside between the '`/*`' characters and the first appearance of '`*/`' characters allowing to include more than one line.

3.2. Data types and constants

The main innovation of the SECREC language is the use of secret data types. This is achieved by adding the security type to each fundamental data type. Data security type can be either 'public' or 'private' and it is followed by the data type. Note, that if no security specifier is given then the public security type is used by default.

Since the language is created with security in mind, the security type should not be confused with the corresponding object-oriented programming terms despite their similarity. Public security type means that data is stored in the public virtual machine and its value can be processed publicly. Private security type means that data is stored in the secret-shared form on the private stack or in a private register of the virtual machine. The shares of the value are distributed between multiple computation nodes – the miners – of the private virtual machine, so that each node has only one share of the actual value. This is the basis of the underlying security scheme.

Each security type can be used in conjunction with certain fundamental data types. SECREC supports both public and private integer and boolean data types. For public security type SECREC additionally supports the `string` data type. The void data type is also considered to be public for the sake of generality. Table 1 summarizes the data types, their possible security types and constants, if available.

Table 1. Data types, their security types and constants.

Security type	Data type	Constants
`public`	`void`	None
`public`	`string`	"…"
`public, private`	`bool`	`true / false`
`public, private`	`int`	0 – 4294967295

3.3. Identifiers

All names in SECREC are identifiers. A valid identifier is a sequence of one or more upper or lower case ASCII letters, digits or underscores '_'. An identifier may begin with a letter or an underscore but not the digit.

Another rule the programmer has to consider when inventing his own identifiers is that they must not match any reserved keywords of the SECREC language. The reserved keywords are: `void`, `string`, `int`, `bool`, `public`, `private`, `true`, `false`, `if`, `else`, `do`, `while`, `for`, `return`, `continue`, `break`.

3.4. Variables

Variables are used to store data values, for example, intermediate computation results. In order to use a variable one must declare it first. This is done by writing the security type specifier followed by the desired fundamental data type, which in its turn is followed by a valid variable identifier. For example the syntax for defining a public integer variable '`x`' is:

```
public int x;
int x;
```

Note, that both declarations are identical because the public security type specifier is optional despite the recommendation to include security specifiers for code unification and clarity. It is also possible to declare and initialize a variable in one step using the following syntax:

```
public int a = 3;
private int b = a + 4;
```

Currently this kind of variable initialization is only allowed if the right side of the initialization is constant, a variable or an expression, but not a function call.

There are two types of variables: global and local variables. Global variables are declared in the very beginning of a program outside all functions, and can be accessed anywhere in the code. Local variables can be declared anywhere within a function body or a block enclosed in curly brackets '`{}`'. The scope of local variables is limited to the block where they are declared starting from the point of declaration. For example, if they are declared at the beginning of the function body, their scope is between the declaration point and the end of that function. This means that if there exist two functions then the local variables declared in the first function cannot be accessed from the other function and vice versa. Additionally, if there are several blocks enclosed into one another then variables declared in the outermost block are also accessible in the inner blocks. It is not allowed to

declare a variable with a name of a variable that is already in scope. The following program code example demonstrates discussed scope rules.

```
 1:  private int a = 5;                  // ok
 2:  void main () {
 3:       public int n = 1;              // ok
 4:       while (n>0) {
 5:            public int n;             // wrong!
 6:            public int b;             // ok
 7:            n = 0;                    // ok
 8:            b = 1;                    // ok
 9:       }
10:       b = 0;                         // wrong!
11:       a = 3;                         // ok
12:  }
13:  public int f (public int x) {
14:       public int n = 2;              // ok
15:       public int a;                  // wrong!
16:       a = 4;                         // ok
17:  }
```

The declaration of variable n on line 5 is wrong because we are trying to redeclare an existing variable declared on line 3. The variable declared on line 3 is accessible in all the following inner blocks of the main function, including the while block. For similar reasons the variable a on line 15 cannot be declared, as the global variable declared on line 1 is accessible in all functions and their inner blocks.

Note that the variable b declared inside the while block on line 6, falls out of scope on line 9, and therefore is not accessible on line 10. This is because the scope of every variable is limited to the block it was declared in.

3.5. Expressions

The SECREC language supports a number of operators that allow creation and evaluation of complex expressions (Table 2). Most of the supported operators are binary. These include multiplication, addition, comparison, logical and assignment operators. The composition of different constants, variables and operators forms an expression that can be evaluated.

After evaluation the expressions have a resulting value. This necessitates us to describe expressions, similarly to any other data unit, with the security type and the data type. Since some expressions are public and the other ones are private then the question arises regarding what happens if expressions of different security type but same data type are combined into one single expression. Because the whole idea of privacy-preserving computation is keeping the secrets private, then publishing private data is out of the question. The solution is to convert expressions that contain private elements into private expressions. This is done by secret-sharing all the public elements and moving them to the private execution environment.

However, it is sometimes necessary to open the secrets to gain some knowledge for good intentions. As the private expressions are combined into a well-built complex algorithm, it becomes possible to gain useful knowledge with high-enough entropy, so that

nobody's privacy is compromised in the process. Additionally, to verify this it is necessary to have some kind of means of tracking the flow of private data in complex algorithms. SECREC introduces a special '`declassify`' operator for explicitly publishing the private information and allowing the data flow analysis of algorithms. By tracking the usage of this operator the analyzer will know all the places in the program, where private data is moved from private execution environment into the public execution environment. This enables it to identify unwanted privacy leaks.

Current state of function calls in expressions. Currently a function call cannot be part of an expression due to shortcomings in the compiler that are yet to be fixed. Therefore, expression can only contain constants, variables or expressions with the same rules. Note, that this is not a functional restriction, but a mere cause of slightly longer SECREC programs. Function calls can be used in conjunction with assignment operators as a right hand expression while assigning the result of a function to an existing variable e.g. `a = function_name(5);` As the development of SECREC goes on, the distinction will no longer be made after allowing function calls to be part of expressions.

Table 2. Operators and supported argument data type / security type combinations.

Operator	Left side data type	Right side data type	Left side security	Right side security	Short description
assign: =	all	All	public	public	Allowed
			public	private	Allowed by using the declassify operator
			private	public	Allowed, right side expression is converted to private
			private	private	Allowed
add: + sum: - mult: *	integers, integer matrices	integers, integer matrices	public	public	Allowed
			Other combinations		Allowed, any public expressions are converted to private
matrix product: #	integer matrices	integer matrices	public	public	Not allowed
			Other combinations		Allowed, any public expressions are converted to private
div: / mod: %	integers	integers	public	public	Allowed
			Other combinations		Not allowed[1]
or: || and: &&	booleans, boolean matrices	booleans, boolean matrices	public	public	Allowed
			Other combinations		Allowed, any public expressions are converted to private
gt: > lt: < gte: >= lte: <= eq: ==	integers, integer matrices	integers, integer matrices	public	public	Allowed
			Other combinations		Allowed, any public expressions are converted to private
not: !	booleans	booleans		public	Allowed
				private	Allowed

[1] Although SHAREMIND does not currently support private division operation, it will do so in the future.

3.6. Vectors and matrices

In addition to the basic data types SECREC supports public and private matrices of integer or boolean types. Since vectors are inherently similar to matrices, the language does not distinguish the two. We state that a vector is a one-column matrix and the language processes it as such. Matrices play a crucial role in SECREC, because the virtual machine running the compiled code is known to be more effective when performing operations on arrays of data. The language offers simplified means for working with matrices.

To define a matrix of integers or booleans a pair of square brackets must be added right after the data type and include the expressions for dimensions between them like this:

```
private int[4][5] a = 3;      // current version
private int[][] a[4][5] = 3;  // future version
```

The examples above define a private matrix named ‘a’ with dimensions of 4 rows and 5 columns such that all its elements are initialized to the value 3. The first example represents the current implementation. As the language grammar was revised, the decision has been made in favor of second example for the future version of SECREC.

The language also provides a fast and simple way for extracting or assigning matrix rows and columns. The following syntax is used for this purpose:

```
private int[4] x;       // syntactic sugar for one column matrices
private int[5] y;
private int[4][5] z;
z[*][3] = x;            // assign a vector to 3rd column
z[4][*] = y;            // assign a vector to 4th row
x = z[*][2];            // extract the second column
y = z[1][*];            // extract the first row
```

A remarkably short way of doing elementwise operations on vectors is the usage of standard arithmetical and logical operators on matrix expressions. This effectively removes the need for looping over the rows and columns using looping statements. Since the elementwise operators execute the same operation on all matrix elements at the same time, these operators are less time-consuming and therefore much faster, considering the nature of SHAREMIND. For example, the code for squaring all the elements of the matrix ‘z’ is:

```
z = z * z;
```

The private virtual machine will push the matrix twice on its stack, perform elementwise private multiplication and pop the resulting shares back into the matrix variable ‘z’.

3.7. Functions

A function is a group of statements that is executed whenever it is called from some point of the program. A SECREC program may define any number of functions. The language supports function polymorphism. It allows defining functions with the same name, as long as their parameter signatures are different. The following syntax must be used to define a function:

```
security_type data_type name ( param1, param2, … ) { statements }
```

where

`security_type` - the security type of function return value (optional),
`data_type` - the data type of the function return value,
`name` - function name,
`param1, …` - function parameters. Each parameter consists of parameter security type (optional), data type and a name. Parameters can be considered as predefined variables that will be accessible in the scope of function body.

Here is an example function:

```
private int fn (public int a, private int b) {
    return a + b;
}
```

It takes two parameters, adds them, converting the public value to private value, and then returns the resulting private value. Next we want to use that function. We can call it from the main or any other function like this:

```
void main () {
    private int x;
    x = fn(3,5);
}
```

Note that passing parameters to the function or returning a value is done by value, since SECREC does not support pointers. Most importantly, the data types as well as parameter and argument count must match. The security types are allowed to differ to some extent. They work exactly as in the assignment operator, where the left side of an operator is a function parameter and the right side is the passed value to that parameter. One can pass public values to private parameters, but not vice versa.

A function may or may not have a return value. If no value is returned, then the return data type of the function is set to `void` and its security type to public. If any other return data type is specified, then the appropriate value must be returned by adding a return statement at some point inside the function body.

The SECREC language also supports recursion. Functions can call themselves inside their own body. Caution is advised in this case because of the risk of being trapped in a never-ending recursion. This eventually causes the machine to run out of memory.

3.8. Condition statements and loops

A program is usually not limited to a linear sequence of instructions. During the execution process it may branch or repeat some steps. For that purpose SECREC provides flow control structures that serve to specify what, when and under which circumstances has to be done by the program.

Conditional statements. The `if` keyword is used to execute a statement or a block of statements if a condition is fulfilled. The corresponding syntax is:

```
if (condition) statement
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword `else`. It is used in conjunction with `if`:

```
if (condition) statement1 else statement2
```

The language also allows repeating some statements for a certain number of times or while a condition is satisfied. There are three possible ways to loop: using `while`, `do-while` and `for` constructions.

The while loop. Its format is:

```
while (condition) statement
```

and its purpose is simply to repeat the statement while the condition is true.

The do-while loop. Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that `condition` in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of the `statement` even if the `condition` is never satisfied.

The for loop. Its format is:

```
for (initialization; condition; increase) statement
```

and similarly to the while loop its main function is to repeat the `statement` while the `condition` remains true. Additionally, the `for` loop provides a specific location to contain an `initialization` statement and `increase` statement. This type of loop is specially designed to perform a repetitive action with a counter which is initialized and then increased after each iteration. The `initialization` part may only contain an expression and not a variable declaration. All variables should have been declared in advance. The increasing of a value must currently be done in the old fashioned way:

```
a = a + 1;
```

Skipping and breaking the cycles. To skip the current loop iteration and proceed with the next one SECREC provides the `continue` statement. The loop can be exited completely by the `break` statement. For these to take effect they must reside within the loop's statement.

3.9. Standard library

The SECREC programming language would not be as useful and usable without a set of core standard functions for programmers to rely on. These functions form the SECREC standard library that extends and complements the capabilities of the language. The standard functions are defined in the SECREC compiler which inlines the corresponding code whenever such a function is used. Potentially the library can be expanded with any frequently reused code. This section describes current contents of the SECREC standard library. As the language is still a work in progress, the library is far from complete and is subject to change at any time.

3.9.1. Vector and Matrix operations

The support for vectors and matrices essentially involves the manipulation of these data structures. Therefore the library includes vector and matrix functions for adding or removing elements or subsets of elements, getting some metadata and performing other operations with data structures in question. Table 3 and Table 4 list all currently supported vector and matrix functions respectively.

Table 3. Vector functions.

Function signatures	Function descriptions
private int **vecSum**(private int[] src) private int **vecSum**(private bool[] src)	Returns the sum of elements of a vector. Boolean vectors are considered zero-one vectors.
public int **vecLength**(private int[] src)	Returns the number of elements in a vector.
void **vecAppend**(public int[][] dest, public int src) void **vecAppend**(private int[][] dest, private int src)	Appends an element to the end of a vector.
void **vecRemove**(public int[][] vec, public int i) void **vecRemove**(private int[][] vec, public int i) void **vecRemove**(public bool[][] vec, public int i) void **vecRemove**(private bool[][] vec, public int i)	Removes an element from a vector at given index.

Table 4. Matrix functions.

Function signatures	Function descriptions
private int **matSum**(private int[] src) private int **matSum**(private bool[] src)	Returns the sum of elements of a matrix. Boolean matrices are considered zero-one matrices.
public int **matColumnCount**(private int[][] src)	Returns the number of columns in a matrix.
public int **matRowCount**(private int[][] src)	Returns the number of rows in a matrix.
void **matAddColumn**(private int[][] src)	Adds a column of zeroes at the right of a matrix.
void **matAddRow**(private int[][] src)	Adds a row of zeroes at the bottom of a matrix.
void **matAppendRow**(public int[][] dest, public[][] int src) void **matAppendRow**(private int[][] dest, private[][] int src)	Appends a row to the bottom of a matrix.
void **matCopy**(private int[][] dest, private int[][] src)	Copies contents of source matrix to destination matrix.
void **matDimensions**(private int[][] src, public int rows, public int cols) void **matDimensions**(public int[][] src, public int rows, public int cols)	Change the shape of a matrix by setting new dimensions. The contents of a matrix remain unchanged.
void **matInsertColumnAt**(private int[][] src, public int i)	Inserts a column into the matrix at given position.
void **matInsertRowAt**(private int[][] src, public int i)	Inserts a row into the matrix at given position.
void **matRemoveRow**(public int[][] mat, public int i) void **matRemoveRow**(private int[][] mat, public int i)	Removes a row from a matrix at given index.
void **matResize**(private int[][] src, public int rows, public int cols)	Resizes the matrix zeroing out the contents.

3.9.2. Database functions

As part of the SHAREMIND virtual machine the database is used to store large amounts of private data. This data can be analyzed or otherwise used during the privacy-preserving computations. Table 5 lists current SECREC functions for accessing the database.

Table 5. Database functions.

Function signatures	Function descriptions
public void **dbLoad**(public string dbname)	Loads a database making it active.
public int **dbColumnCount**(public string tablename)	Returns the number of columns in a table.
public int **dbRowCount**(public string tablename)	Returns the number of rows in a table.
private int[][] **dbGetColumn**(public string colname, public string tablename)	Returns the data of a table column as a vector.

3.9.3. Stack functions

Although higher-level languages aim at simplifying the program syntax and hiding the hardware functionality, in some cases it is necessary to fine-tune or optimize the program by explicitly calling low-level instructions and manipulating the execution flow. SECREC programs can be optimized by smart use of SHAREMIND assembly calls. Functions described in Table 6 are simple wrappers for a number of assembly instructions for manipulating the stack of the SHAREMIND virtual machine. These are currently used for vectorization of case study algorithms presented in Section 6. Note, that compiler optimization techniques may obsolete these functions in the future.

Table 6. Stack functions.

Function signatures	Function descriptions
private int **stack_pop**()	Pops and returns an element from the top of the VM stack.
private int **stack_stacksize**()	Returns the number of elements on the stack.
private int[][] **stack_popVector**(public int n)	Pops n elements from the stack and returns them as a vector.
void **stack_dbPushColumn**(public string colname, public string tablename)	Orders the VM to push a table column onto the stack.
void **stack_dup**(public int n)	Duplicates top n elements on the stack.
void **stack_mul**(public int n)	Replaces two n-element vectors on the top of the stack with their elementwise multiplication
void **stack_push**(private int src)	Pushes an element on the top of the stack.
void **stack_pushVector**(private int[][] src) void **stack_pushVector**(private bool[][] src)	Pushes a vector of elements on the top of the stack.
void **stack_sumVectors**(public int n, public int s)	Replaces n vectors of size s on the top of the stack with their sums.

3.9.4. Utility functions

The standard library also contains functions for various other purposes. Currently these can be categorized into declassification, conversion, output and publishing functions and they are described in more detail in Table 7.

Table 7. Various utility functions.

Purpose	Function signatures	Function description
output	void **print**(public int src) void **print**(public bool src) void **print**(public string src) void **print**(public int[][] src) void **print**(public bool[][] src)	Prints an expression on the console. Used for debugging.
	void **matPrint**(public int[][] src) void **matPrint**(public bool[][] src)	Prints a vector or a matrix expression on the console as a matrix. Used for debugging.
declassification	public int **declassify**(private int src) public int[] **declassify**(private int[] src) public bool **declassify**(private bool src) public bool[] **declassify**(private bool[] src)	Declassifies an expression changing its security type from private to public. This explicitly moves the secret-shared value into the public execution environment.
publishing	void **publish**(public string dest, public int src) void **publish**(public string dest, public int[][] src) void **publish**(public string dest, public bool src) void **publish**(public string dest, public bool[][] src) void **publish**(public string dest, public string src)	Publishes an expression as a named return value. Used for returning values from the program to the virtual machine.
conversion	private int **boolToInt**(private bool src) private int[][] **boolToInt**(private bool[][] src)	Takes in an expression of one type and converts it to an expression of other type.

4. A compiler to SHAREMIND assembly

In order to make the SECREC language usable, it is necessary to execute the programs written in that language. We have chosen to implement a compiler that translates SECREC programs into SHAREMIND assembly, as SHAREMIND has all the means for running privacy-preserving algorithms. Theoretically, it is also possible to compile SECREC to other multi-party computation machines. In the following, we will describe the main aspects of our compiler.

4.1. Compiler architecture

A compiler is a program that translates source code written in the source programming language into the target programming language. The SECREC compiler is written in Java using the ANTLR parser generator [16]. Compiling the source code into the target language is a complicated process with several steps as depicted on Figure 5.

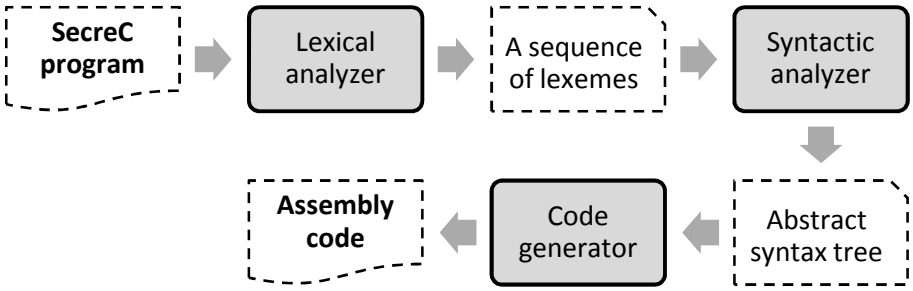

Figure 5. The structure of the SECREC compiler.

The first important compiler block is the lexical analyzer – the scanner. It takes in the source code of a SECREC program and applies the specified SECREC lexical grammar to find the corresponding lexemes. Lexemes are the smallest independent meaningful pieces of source code – such as comments, strings, textual representations of supported data, language keywords, identifiers, operators and separators. The redundant whitespaces are removed to compact the source code and accelerate the following phases. As a result, the lexical analyzer converts symbols in the source code into a sequence of lexemes.

The next block is the syntactic analyzer – the parser – that takes in the sequence of lexemes from the lexical analyzer. Its task is to convert the lexemes into program instructions and expressions and to build an abstract syntax tree out of those. The tree is constructed by applying the specified parser grammar to the sequence of lexemes. Both grammars are interleaved with Java code to track variables and function signatures.

The last block is the code generator. It takes the abstract syntax tree and applies specified rewriting rules to it in order to generate Sharemind assembly code. Code generation is implemented by using the StringTemplate library provided with ANTLR. The output code corresponds to the SECREC program, as the abstract syntax tree is the connecting structure between the two.

The resulting Java application SecreCCompiler acts as a compiler from SECREC inputs to SHAREMIND assembly outputs.

4.2. Compilation schemes

Compilation schemes describe how the particular language constructions are translated into the SHAREMIND assembly. These schemes are hardcoded into the SECREC compiler using the StringTemplate template language. StringTemplate is a template engine library that powers the ANTLR code generator, and is used for generating text from data structures.

The current implementation of the SECREC compiler has compilation schemes for most of the language features. We will not document all these schemes here, but they are provided in the source code of the compiler. See Appendix 1 for details.

4.3. Standard library

The standard library of the SECREC language is currently built in to the compiler. If a function call is found in the source code, the compiler verifies that this call matches either a function specified in the program or a built-in function. In the latter case, the abstract syntax tree is modified with new nodes that will be translated into the relevant assembly operations later in the code generation phase. Special compilation schemes exist for each function call. In future versions of the compiler this approach will be replaced with a more generic one that allows the users to create their own libraries.

5. Developing algorithms with SECREC

While knowing the main features of the SECREC language, writing efficient and secure programs still remains a non-trivial task. It requires an understanding of base principles of the underlying system. Based on the prerequisites discussed so far one can begin programming privacy-preserving data mining algorithms in SECREC. However, there are still some techniques that are important to keep in mind.

Data mining is closely bound with statistics, where trends and aggregations are more important than individual values. This is good, because in privacy-preserving multiparty computations no party is interested in losing control over their secret values. If there are many parties and many values, those can be combined to find useful statistical facts. This observation gives rise to several programming strategies.

The first and most important technique to know is to declassify private data as little as possible. Due to the fact that complex decision making requires the corresponding data to be public, one should also keep the number of decisions depending on the private data as low as possible, as the private data would have to be published first. Ideally, declassification should be used only for final results or intermediate results with a low privacy risk.

Secondly, since the described technique prohibits the decision making on private data it may seem that there is not much to do about it. However, there is another technique that allows decision making based on private data without disclosing. It is possible to replace if-then conditional statements with oblivious selection clauses. For instance consider the conditional statement:

```
if (a) x = y; else x = z;
```

It can be represented as:

```
x = a*y + (1-a)*z;
```

Theoretically the SECREC compiler can be updated to use this method for compiling the if-statements that depend on expressions with private security type.

The third important principle is to use the data aggregation techniques that maximize the measure of uncertainty of published output result. When programming the algorithms, one should gather a reasonably large amount of data to be computed and aggregate it as much as possible. This renders the final results to be statistically more interesting, and at the same time contributes to the uncertainty of the output. The more we aggregate the data, the harder it becomes to derive individual values of the original dataset.

The fourth aspect to consider in the context of the SHAREMIND platform is the amount of vectorization used in the algorithm implementation. The fewer distinct comparison and multiplication operations the program has to execute, the faster it will be in the long run. In reality the performance boost is significant even if there is very little to process. Hence, the programmer should choose to implement well parallelizable algorithms. One should practice the vectorization smartly, as extensive use of this method may unwillingly cause the algorithms to fail because of their very large memory footprint.

In the following we are going to observe a small example in SECREC. The code presented in Algorithm 1 counts needles in the haystack by iterating over the haystack elements and comparing them to the original needle.

Algorithm 1. Naive privacy-preserving implementation for finding needles in a haystack

```
 1:  public int count (private int needle, private int[] haystack) {
 2:       public int i = 0;
 3:       public int count = 0;
 4:       public int n;   n = getRowCount(haystack);
 5:
 6:       for (i = 0; i < n; i = i + 1) {
 7:            private bool r = (haystack[i] == needle);
 8:            public bool equal; equal = declassify(r);
 9:            if (equal) count = count + 1;
10:       }
11:       return count;
12:  }
```

Since the haystack is of a private security type, then every haystack element is privately compared to our original needle (line 7). The needle is secret-shared before the comparison takes place. Next we declassify the comparison result (line 8) and see what it is. If the the compared items were equal, then we increase the counter (line 9). This code is neither secure nor efficient, because it leaks individual comparison results and it does every comparison separately.

Algorithm 2. A better privacy-preserving implementation for finding needles in a haystack

```
 1:  public int count (private int needle, private int[] haystack) {
 2:       public int n;   n = getRowCount(haystack);
 3:
 4:       // An indicator vector of ones and zeroes
 5:       private bool[n] indicator;
 6:       indicator = (haystack == needle);
 7:
 8:       private int sum; sum = vecSum(indicator);
 9:       public int count; count = declassify(sum);
10:       return count;
11:  }
```

A better code will be as presented in Algorithm 2. It first tries to construct a parallelized private comparison of the needle and the haystack (line 6). For this the needle expression is secret-shared and extended into a vector of the same size as the haystack, and every element of the new vector has the value of our original needle. After the elementwise comparison we get a new private zero-one vector with ones where haystack elements were equal to the needle and zeros otherwise. When summed together (line 8), we obtain the count of haystack elements that were equal to the needle. The final result is then declassified (line 9) and returned. Note, that it is a much more secure and efficient version of the same algorithm. It performs a vectorized comparison of the needle and the haystack, declassifying only the final result. The intermediate comparison results of individual values are kept private and never released.

6. Case studies

6.1. Notational conventions

Before we move on, we will agree on notation to be used further in this thesis. For consistency with other work done on similar topic, we will utilize the same notation used in the Cybernetica research report [17].

Data types. To explicitly separate private data and public data, we surround private variables with double brackets. For example, $[\![x]\!]$ denotes that variable x is secret-shared and $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ means that the shares of $z = x \cdot y$ are securely computed from the shares of x and y. The same applies to vectors and matrices, as well. Vectors are denoted by bold lowercase letters ($\boldsymbol{a}, [\![\boldsymbol{b}]\!], \dots$)and matrices are denoted by uppercase letters ($A, [\![B]\!], \dots$). Sets and list are denoted by calligraphic uppercase letters ($\mathcal{A}, \dots$).

Indexing. To simplify notations, we use the semi-standard indexing operator $[\cdot]$ to select elements, rows and columns form vectors, lists and matrices. For instance, $\mathcal{A}[5]$ and $[\![\boldsymbol{b}]\!][3]$ denote respectively the 5th element of the list $\mathcal{A}$ and the 3rd element from the vector of shares $[\![\boldsymbol{b}]\!]$. Similarly, $A[2,4]$ and $[\![B]\!][5,1]$ denote respective elements in the matrices. Additionally, we use index sets and wildcards for brevity and convenience. That is, $A[*,2]$ denotes the second column of the matrix A and $\mathcal{A}[\mathcal{I}]$ denotes a new list $a_{i_1}, \dots, a_{i_k}$ for an index set $\mathcal{I} = \{i_1, \dots, i_k\}$. If $\mathcal{I}$ is a zero-one vector of the same length as $\mathcal{A}$ then $\mathcal{A}[\mathcal{I}]$ denotes a new list $a_1, \dots, a_k$ such that $a_j \in \mathcal{A}[\mathcal{I}]$ iff $\mathcal{I}[j] = 1$ and $a_j \notin \mathcal{A}[\mathcal{I}]$ iff $\mathcal{I}[j] = 0$.

Matrix and vector operations. All operations on matrices are completed elementwise unless specified differently. The shorthand $A \odot B$ denotes the elementwise product of matrix entries with confirming dimensions. Let $\boldsymbol{a} \circledast \boldsymbol{b}$ denote the elementwise product of vectors with potentially different lengths, where the shorter vector is repeated to fill out the missing places.

For instance, if $\boldsymbol{a} = (1,2)$ and $\boldsymbol{b} = (1,1,3,5)$ then $\boldsymbol{a} \circledast \boldsymbol{b} = (1,2,3,10)$. The notation can be naturally extended to matrices. Let $\boldsymbol{vec}(A)$ denote a vector that is obtained by stacking all column vectors on top of each other, i.e. $A[*,1], A[*,2], \dots$. Then $A \circledast B = C$ where C has the same dimensions as the larger matrix and $\boldsymbol{vec}(A) \circledast \boldsymbol{vec}(B) = \boldsymbol{vec}(C)$. If matrices have the same row dimensions, then the smaller matrix is replicated as many times to match the size of the larger matrix and elementwise multiplication is performed afterwards.

6.2. Histogram computation

The algorithm. A histogram is often the simplest and yet a very powerful tool for studying the frequencies of the values. The histogram consists of the frequency distribution of the whole data set and its graphical representation.

The frequency distribution can be described as a mapping m_i that counts the number of observations that fall into various disjoint categories or ranges $[a_i, b_i]$ known as bins. If the values are discrete, it may also happen, that $a_i = b_i$. In the case one bin counts the occurrences of a single value. Thus, if we let n be the total number of observations and k be the total number of bins, the histogram m_i meets the following conditions:

$$n = \sum_{i=1}^{k} m_i.$$

Visually the histogram is a graphical display of tabular frequencies, shown as adjacent rectangles. Each rectangle is towered over an interval with an area equal to the frequency of the interval. A histogram is often used in many data analysis applications. For example, when reporting surveys graphical histograms are used to describe the distribution of answers to questions with fixed choices. Moreover, if the resolution is properly chosen then the histogram does not disclose much about individual data points.

The privacy-preserving histogram computation code is presented in the Algorithm 3. The private data in form of a secret-shared vector $[\![\boldsymbol{v}]\!]$ is passed on directly to the counting code that counts the number of single choices in the data. The second parameter is the number of possible choices n. Next, we make use of the fact, that the count of occurrences of a single choice in the private data can be computed in parallel, as described in Chapter 5. Furthermore, it is possible to compute the counts for all choices in one shot. Hence, we use parallelization so that every possible choice is compared to every value in the input data at once. For every choice $n_i \in \{0, \dots, n-1\}$ we must create a vector of the same size as the private data, so that all its elements have the value of n_i. This is done on line 4 where we replicate each choice into a vector of size $r = |[\![\boldsymbol{v}]\!]|$. Then all the created vectors have to be privately compared to the original data, which is why the latter is also replicated n times on line 2. A big comparison operation on line 5 is used for testing all the choices. On line 6 we take the test result and sum the corresponding n vectors of size r to find the counts of individual choices in the input data. The counts are then declassified on line 7 and returned.

The implementation loads the input data from the SHAREMIND private database and runs the histogram function with different choice ranges. Note, that presented code can easily be adapted to support intervals, if the equality on line 5 is replaced with two greater-than comparisons.

Algorithm 3. Pseudocode for the SecreC Histogram algorithm

```
1:  function HISTOGRAM ([[v]], n)
2:      [[d]] ← Replicate([[v]], n)
3:      r ← Size([[v]])
4:      [[i]] ← Replicate(0, r) | ... | Replicate(n − 1, r)
5:      [[c]] ← ([[d]] == [[i]])
6:      [[f]] ← SumVectors([[c]], n, r)
7:      f ← declassify([[f]])
8:      return f
9:  end function
```

Security analysis. To show that an algorithm is private, we have to make sure that the execution flow can be efficiently reconstructed and declassified values can be efficiently computed from the public parameters and intended outputs. While fulfilling these requirements will not directly give us a cryptographic proof, it will show that the private inputs have not been compromised. It must be noted, that the privacy of computations is guaranteed by the underlying secure protocols of the SHAREMIND platform.

For the histogram, it is easy to see that the execution flow of the algorithm depends only on the public input n. All comparisons are performed in parallel and the results are not published. Instead, we make use of the specific elementwise vector comparison operation in SHAREMIND that returns ones and zeroes depending on the result of the comparison. These ones and zeroes are added together for each choice to find the numbers of values that satisfied the equality. Only the numbers of such values are disclosed, so the algorithm preserves privacy.

Performance results. To test the performance of the algorithm we implemented it in the SECREC language and executed it on the SHAREMIND platform. The benchmarking was conducted in the High Performance Computing Center of the University of Tartu on machines with two 2.5GHz quad-core Intel Xeon CPUs, 32GB of RAM and a very fast network connection. We used the standard SHAREMIND setup with three miner computers and one client computer. The client submitted various queries to the miners. Execution times were measured at the client starting from query formation and ending with receiving the result.

We assumed that the survey is filled by several people and the data is collected into the SHAREMIND's database. So we generated the answer datasets with various sizes and choices. The results are depicted on Figure 6. The costliest operation in the algorithm is the comparison operation, since we try to construct as big comparison as possible to maximize parallelization. Since this operation is constant-time, then execution times for a given dataset are linear in the number of possible choices.

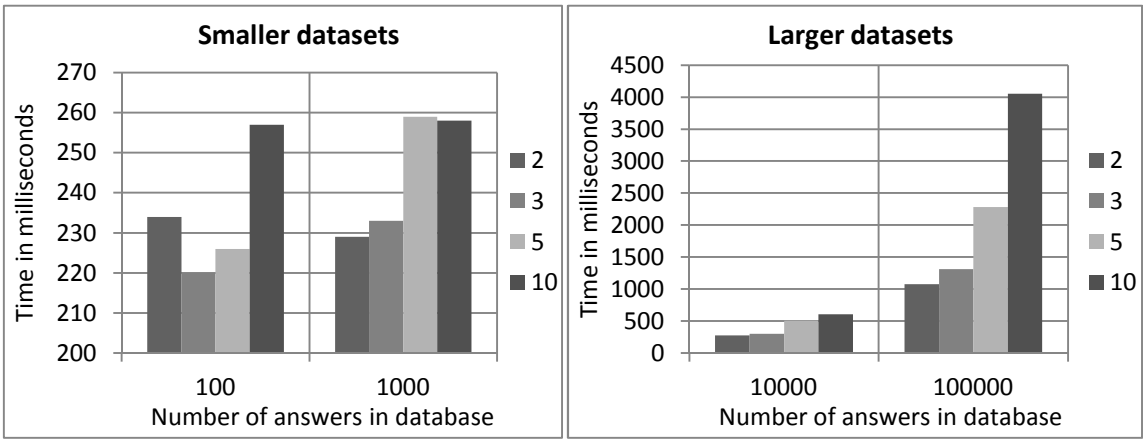

Figure 6. Benchmarks for private Histogram computation with 2, 3, 5 and 10 choices on different datasets.

6.3. Frequent itemset mining

Data analysis tasks often rely on finding frequent patterns [18] in databases, such as association rules, correlations, clusters and many more. The mining of association rules is one of the most popular problems. The original motivation for searching association rules came from the need to analyze the so called supermarket transaction data. The idea is to examine the customer behavior in terms of the purchased products, and find out which items are frequently purchased together.

More formally, the problem of frequent itemset mining (FIM) can be presented as follows. Let $\mathcal{A} = (a_1, \dots, a_m)$ be the list of all attributes, that can appear in a transaction, e.g. all the products available in the supermarket. In this case, every transaction is a subset of $\mathcal{A}$. As depicted on Figure 7, a list of transactions $\mathcal{T}_1, \dots, \mathcal{T}_n$ forms a transactional database $\mathcal{D}^{n \times m}$, such that $\mathcal{D}[i,j] = 1$ iff $a_j \in \mathcal{T}_i$, and $\mathcal{D}[i,j] = 0$ otherwise. The *cover of an itemset* $\mathcal{X} \subseteq \mathcal{A}$ is the set of transaction identifiers that contain the itemset $\mathcal{X}$. The *support of an itemset* $\mathcal{X}$ is the number of transactions that contain all items of $\mathcal{X}$.

	a_1	a_2	...	a_m
$\mathcal{T}_1$	1	1	...	0
$\mathcal{T}_1$	0	1	...	1
$\mathcal{T}_1$	1	0	...	0
...	...	...	...	...
$\mathcal{T}_n$	0	1	...	1

Figure 7. The transactional database.

Despite its benefits in various areas, data mining (e.g. frequent itemset mining) can also pose a threat to privacy and information security if used improperly. Since the data transactions in the database are associated with the customers, one can find out and exploit habits of individuals. Removing this association from the database still does not protect people's privacy well enough. Having some extra knowledge while analyzing the transactions makes it possible to distinguish who is who. This motivates us to use secure multiparty computation systems, such as SHAREMIND, to store and analyze the data in a privacy preserving way.

To conduct frequent itemset mining on SHAREMIND, we need to adapt the ways the common FIM terms used and computed. In privacy-preserving computations covers can be represented as index vectors $\boldsymbol{x}$, such that $x_i = 1$ if $\mathcal{X} \subseteq \mathcal{T}_i$ and $x_i = 0$ otherwise. This technique allows us not to depend on the transaction identifiers by dropping them from the database for additional data privacy. Then given $a \in \mathcal{A}$ the covers and can be computed

$$\begin{aligned} \text{cover}(\{a\}) &= \mathcal{D}[*, a] \\ \text{cover}(\mathcal{X} \cup \mathcal{Y}) &= \text{cover}(\mathcal{X}) \odot \text{cover}(\mathcal{Y}) \end{aligned} \qquad (1)$$

where $\mathcal{D}[*, a]$ denotes a column in $\mathcal{D}$ that corresponds to the attribute a. Since covers are represented as zero-one index vectors, then in order to find the number of transactions that contained an itemset, we have to sum the elements of the corresponding index vector

$$\text{supp}(\mathcal{X}) = |\text{cover}(\mathcal{X})| = |\boldsymbol{x}| = x_1 + \cdots + x_n.$$

Note, that support is an anti-monotone function, as

$$\mathcal{X} \subseteq \mathcal{Y} \Longrightarrow supp(\mathcal{X}) \geq supp(\mathcal{Y})$$

meaning that subsets of frequent itemsets must also be frequent. This observation allows us to develop several frequent itemset mining strategies, as the task boils down to a tree traversal problem. Given a database with four attributes, consider the worst case scenario tree of frequent itemset candidates presented in Figure 8. It contains an optimal amount of candidate itemsets that need to be verified when looking for frequent itemsets.

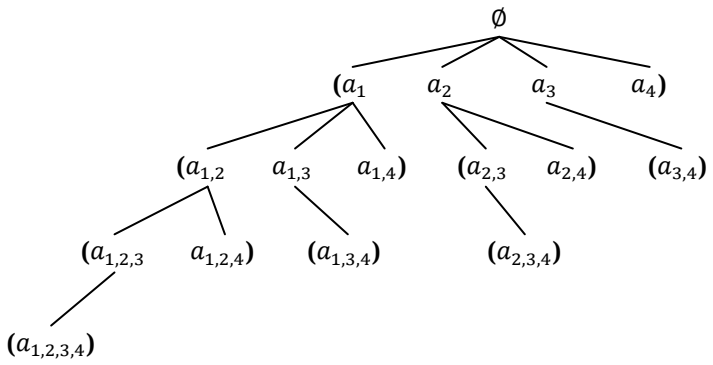

Figure 8. Frequent itemset candidate tree in case of a database with 4 attributes.

In the following sections of this chapter we are going to present three novel privacy-preserving algorithms for traversing the candidate tree. These results have also been recently submitted as a scientific paper [9].

6.4. Apriori

The algorithm. *Apriori* is a frequent itemset mining algorithm originally described by R. Agrawal et al. [19]. The algorithm is based on breadth-first search as it traverses the frequent itemset candidate tree in level-wise order. The brief idea is that it first generates a list of frequent items F_1 and then runs the following cycle. Given a list of frequent i-element itemsets F_i it generates a candidate list of potentially frequent $(i+1)$-element itemsets C_{i+1}, and verifies which of those are really frequent. The candidates are generated with support monotony in mind – an itemset is a candidate if all of its subsets are known to be frequent. This property is also known as the Apriori principle. To decide whether an itemset is frequent or not, *Apriori* is given the threshold t, which is then compared to the support of that itemset. In terms of FIM, itemsets with support of at least t are considered frequent.

In Algorithm 4 we present the pseudocode for the new privacy-preserving version of *Apriori* [9]. The algorithm follows the same description as discussed above, except that it tries to perform the computations in the private execution environment.

Algorithm 4. Privacy-preserving Apriori algorithm with cover caching

```
1:  function APRIORI (⟦D⟧, k, t)
2:      ⟦s_1⟧ ← ColSum(⟦D⟧)
3:      f_1 ← declassify(⟦s_1⟧ ≥ ⟦t⟧)
4:      F_1 ← {A[i] : f_1[i] = 1}
5:      ⟦M_1⟧ ← ⟦D⟧[*, F_1]
6:      for i ∈ {1, ..., k − 1} do
7:          (C_{i+1}, J_1, J_2) ← GenerateCandidates(F_i)
8:          ⟦M_{i+1}⟧ ← ⟦M_i⟧[*, J_1] ⊙ ⟦M_i⟧[*, J_2]
9:          ⟦s_{i+1}⟧ ← ColSum(⟦M_{i+1}⟧)
10:         f_{i+1} ← declassify(⟦s_{i+1}⟧ ≥ ⟦t⟧)
11:         F_{i+1} ← {C_{i+1}[i] : f_{i+1}[i] = 1}
12:         ⟦M_{i+1}⟧ ← ⟦M_{i+1}⟧[*, F_{i+1}]
13:     end for
14:     return F_1 ∪ F_2 ∪ ... ∪ F_k
15: end function
```

Apriori is given the secret-shared transactional database $[\![D]\!]$, the maximum itemset size k we are interested in, and the minimal support threshold t. The first level of 1-element candidates does not have to be generated, as the database attributes already represent them. Hence, the algorithm starts with verifying the 1-element candidates (lines 2-5) and then proceeds with generating and verifying the $(i+1)$-element candidates based on frequent itemsets $\mathcal{F}_i$ (lines 6-13) found in the previous steps. Vectors $\boldsymbol{s}_i$ contain the supports of the corresponding candidates $\mathcal{C}_i$. The supports are computed by privately summing the respective cover vectors without distinguishing the individual rows. To find out which of the candidates are frequent, the supports are privately compared to the threshold t (lines 3,10). The comparison results are declassified in order to select the frequent itemsets $\mathcal{F}_i$ (line 3-4, 10-11) and to store the corresponding covers as columns in a separate private matrix $[\![M_i]\!]$ (line 5, 12). New candidate sets and two index vectors $\mathcal{J}_1$ and $\mathcal{J}_2$ are generated on line 7, such that

$$\mathcal{C}_{i+1} = \mathcal{F}_i[\,\mathcal{J}_1] \cup \mathcal{F}_i[\,\mathcal{J}_2]$$

i.e. every candidate consists of frequent itemsets, that correspond to indexes in $\mathcal{J}_1$ and $\mathcal{J}_2$. Thus, the equation (1) assures that the matrix assignment on line 8 produces respective cover vectors for candidates $\mathcal{C}_{i+1}$ and the algorithm is formally correct.

Security analysis. To evaluate the security of the privacy-preserving version of *Apriori* we have to consider the following. As we discussed earlier, public and private data may not be mixed together in the public computation environment. Neither can we make decisions on the private data without leaking the respective values. As long as the private data stays in the private computation environment and all the security requirements for the deployed SHAREMIND system are met, the miners only learn the execution flow and all public and declassified variables. Hence, it is sufficient to show that the values seen by the miners do not reveal more information than originally intended.

The only way to move the private data into the public environment is to use the `declassify` operator. If we look at the Algorithm 4, we can see that only the comparison

results are declassified (lines 3, 10). Based on these results the algorithm selects the itemsets that appeared to be frequent (lines 4, 11), and returns the frequent itemsets in the end (line 14). Given the list $\mathcal{F}$ containing all the frequent itemsets with up to k elements, it is straightforward to determine the sets $\mathcal{F}_1, \ldots, \mathcal{F}_k$. From these sets we can easily compute the candidate sets $\mathcal{C}_2, \ldots, \mathcal{C}_k$, as it is done by the *Apriori* algorithm (line 7). Thus, the sizes of matrices nor the index sets $\mathcal{I}_1$ and $\mathcal{I}_2$ leak no additional information than is revealed by the declassification operations. Note, that it is also possible to reconstruct the execution flow of the algorithm, as the sets $\mathcal{F}_i$ and $\mathcal{C}_i$ completely determine the declassification results:

$$\boldsymbol{f}_i[j] = 1 \Leftrightarrow \mathcal{C}_i[j] \in \mathcal{F}_i.$$

One may be concerned if this is the most secure way of finding frequent itemsets, if we are only interested in the itemsets of size k. What if there is a way to compute k-frequent itemsets $\mathcal{F}_k$ without revealing intermediate results $\mathcal{F}_1, \ldots, \mathcal{F}_{k-1}$? However, there is no more secure way, since even if the algorithm returned interesting itemsets of size k and nothing else, then thanks to the Apriori principle we can derive from the final result, which its subsets were frequent. Therefore it does not make sense to try to hide the intermediate results. Hence, the declassification reveals nothing that cannot be computed from the outputs and the algorithm is secure.

6.5. Eclat

The algorithm. *Eclat* is another frequent itemset mining algorithm [20] that in contrast to *Apriori* is based on depth-first search. This strategy allows the algorithm to consume much less memory than the level-wise approach. *Eclat* is based on recursive step that takes in a frequent itemset $\mathcal{X}$ and tries to expand by adding new items to it.

Algorithm 5. The privacy-preserving Eclat algorithm

```
1:  function EclatStep (X, N, [[M]], k, t)
2:      [[M]] ← [[M]][*, X] ⊛ [[M]][*, N]
3:      [[s]] ← ColSum([[M]])
4:      f ← declassify([[s]] ≥ [[t]])
5:      F ← ∅,  F* ← {X ∪ N[i] : f[i] = 1}
6:      if |X| ≥ k − 1 then return F*
7:      for Y ∈ F* do
8:          N* ← {Z ∈ F* : Y ≼ Z}
9:          F ← F ∪ EclatStep (Y, N*, [[M]][*, N*], k, t)
10:     end for
11:     return F
12: end function

13: function Eclat ([[D]], k, t)
14:     return EclatStep (∅, A, [[D]], k, t)
15: end function
```

Our privacy-preserving version of *Eclat* is depicted on Algorithm 5. In addition to the frequent itemset $\mathcal{X}$, the recursive step takes in a list of potential extensions $\mathcal{N}$ and the matrix $[\![M]\!]$ of corresponding cover vectors. To find one element bigger frequent itemsets

the algorithm combines itemset $\mathcal{X}$ with all potential extensions $\mathcal{N}$. The corresponding covers are privately multiplied together on line 2 and respective supports computed on line 3. The supports are then compared to the minimal support threshold t and the results are declassified on line 4. If the threshold requirement was met, then the new frequent itemsets are physically constructed on line 5. The recursion step is aborted if the size of $\mathcal{X}$ is equal or larger than $k-1$, as in this case the recursion has found all k-element frequent itemsets and no further search is necessary. Otherwise, the new potential extensions $\mathcal{N}_*$ are generated (line 8) and the new recursion step is executed to find bigger frequent itemsets. The results are merged, as each recursion step finishes its execution. This way the algorithm first reaches as deep into the candidate tree as it can and then checks next unverified branches.

Security analysis. Since the execution path of *Eclat* only depends on the frequency tests, then analogous argumentation holds as for the *Apriori* algorithm.

6.6. Hybrid-Apriori

The algorithm. The problem with the two briefly described FIM algorithms for SHAREMIND is that when it comes to large data sets, then one of them is slow and another may fail before reaching the end due to the unreasonably large memory footprint. To overcome the problem we need a hybrid solution that will join the advantages of the algorithms in question. In the following we will try construct a new algorithm to achieve that.

Since *Apriori* is faster and better parallelizable, it has better potential for using less memory than it currently does, while still being faster than *Eclat*. On the other hand *Eclat* does not seem to be capable of being much faster than it is, since the use of recursion sets makes it difficult to apply vectorization. Moreover, *Eclat* becomes reasonably efficient on large datasets with many rows, and does not need additional vectorization. As a logical corollary of our argumentation we will choose *Apriori* as the basis of our new algorithm.

To reduce the memory requirements we will have to limit the amount of data computed during a single iteration, and if possible also minimize the large cache data. The first goal can be attained by limiting the size of parallel multiplications by setting a constraint on the maximum number of potential frequent itemsets that can be verified in one shot. Let that number be ℓ. This breaks the iterations down into chunks of size up to ℓ. They can be smaller if there is not enough frequent itemset candidates to compute at the maximum capacity. If m is set to high enough, then for some particular dataset the algorithm will become a usual *Apriori*, that was implemented so far. This is because each loop iteration will verify the maximally available candidates for that iteration. Those are all the candidates that the candidate tree would have on a corresponding depth, which is what the conventional *Apriori* does.

After each loop iteration there is a good chance that some new candidates can be generated. Thanks to the Apriori principle this can be done by merging frequent itemsets of found so far. To eliminate duplicate candidates, only frequent itemsets of the same size will be merged in ascending order. Additionally, there have to be two index vectors to contain the information about frequent itemsets that were used to construct the corresponding

candidates. Those indexes will be used to find necessary cached covers of the components when computing frequencies for each candidate.

Once the iteration is done the candidates that appeared to be frequent must, according to the itemset size we are interested in, either be added to the final result or be stored along with the corresponding cover vectors into the cache. However, not all of the stored frequent itemsets will always be needed in the future as the algorithm proceeds. The minimization of memory footprint can be achieved by removing cache data that will no longer be needed for verifying the candidates in the future. To know which frequent itemsets are no longer needed we can create an additional integer vector containing the indexes of the farthest candidates in time that will ever need corresponding frequent itemset cache data for verification. This vector must and can be easily updated every time we generate new candidates out of available frequent itemsets.

All the main steps described will have to be repeated in a loop until there are no more candidates to verify. The update step of the loop presented in Algorithm 6 depends on the cover matrix $[\![M]\!]$, the candidate list $\mathcal{C}$ with corresponding component index vectors $\mathcal{I}_1$and $\mathcal{I}_2$, the maximal interesting set size k, the minimal support threshold t, and the maximum number of candidate sets to verify in one shot ℓ. In the beginning the update step multiplies the cover vectors that correspond to the first ℓ candidates in $\mathcal{C}$ (line 2). Then, as in *Apriori*, the supports are compared to the threshold and the comparison results declassified to select the respective frequent itemsets from $\mathcal{C}$ (lines 3-5). After verifying the candidates, the algorithm removes them and their metadata to save up some memory. The new frequent itemsets and their covers are stored for later use. Additionally, *Hybrid-Apriori* removes the frequent itemset data, which will not be used later for candidate verification, as no candidates will rely on those itemsets in the future. When the loop ends, the final answer should contain all the frequent itemsets that met our conditions.

Algorithm 6. The privacy-preserving update step in Hybrid-Apriori

1: **function** HYBRIDAPRIORIUPDATE ($[\![M]\!], \mathcal{C}, \mathcal{I}_1, \mathcal{I}_2, k, t, \ell$)
2: $[\![M_*]\!] \leftarrow [\![M]\!][*, \mathcal{I}_1[1, \dots, \ell]] \odot [\![M]\!][*, \mathcal{I}_2[1, \dots, \ell]]$
3: $[\![\boldsymbol{s}]\!] \leftarrow$ ColSum($[\![M]\!]$)
4: $\boldsymbol{f} \leftarrow$ **declassify**($[\![\boldsymbol{s}]\!] \geq [\![\mathrm{t}]\!]$)
5: $\mathcal{F}_* \leftarrow \{\mathcal{C}[i] : \boldsymbol{f}[i] = 1\}$
6: Remove ℓ first elements from $\mathcal{C}$ and add new candidates due to itemsets $\mathcal{F}_*$.
7: Merge matrices $[\![M]\!]$ and $[\![M_*]\!][*, \mathcal{F}_*]$ and delete all unnecessary columns.
8: **end function**

On Figure 9 and Figure 10 we can see how, given a database with 4 items, new candidates are generated over time as the algorithm progresses in the worst case with all items being frequent. The index row contains indexes of the candidates in columns under it, and the rows below it depict the iterations. After the iteration ends, the candidates become frequent itemsets. As new candidates are created, the indexes of component frequent itemsets are stored in `comp_I` and `comp_J`. By the end of the algorithm the `furthest_C` row contains the indexes of the candidates that are the last to use the corresponding itemsets. The zeroth row is actually the candidate set before the iterations begin. The candidate columns processed during each loop are marked with darker gray and the loop ends with a

thick black line. If iteration relied on some previously found frequent itemsets, then those are marked with lined cells.

furthest_C	7	8	9	9	11	12	13	12	13	13	14	14	14	14	14
comp_I					1	2	2	3	3	3	5	7	7	8	11
comp_J					0	0	1	0	1	2	4	4	5	6	10
iter \ index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	2	3	4											
1					12										
2						13	23	14	24	34					
3											123				
4												124	134		
5														234	
6															1234
7															
8															

Figure 9: Candidate generation in worst case, given a 4-item DB, $\ell = 2$, interesting size 4

furthest_C	7	8	9	9	11	12	13	12	13	13	14	14	14	14	14
comp_I					1	2	2	3	3	3	5	7	7	8	11
comp_J					0	0	1	0	1	2	4	4	5	6	10
iter \ index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	2	3	4											
1					12	13	23	14	24	34					
2											123	124	134		
3														234	1234
4															

Figure 10: Candidate generation in worst case, given a 4-item DB, $\ell = 4$, interesting size 4

Consider the Figure 9, where the maximum number of candidates to verify during a single iteration is limited to $\ell = 2$. The attributes of a database represent the initial set of candidates $\{\{1\},\{2\},\{3\},\{4\}\}$ that do not have any parent itemsets. During the first iteration we first verify the candidates $\{1\}$ and $\{2\}$ under indexes 0 and 1. After that we can construct a new candidate itemset $\{1,2\}$ out of $\{1\}$ and $\{2\}$, and store it in the end of the candidate vector under index 4. With it we also remember the indexes of component itemsest. On second iteration we process the next two candidates $\{\{3\},\{4\}\}$. After they are verified and found to be frequent, we are able to generate a set of new candidates $\{\{1,3\},\{2,3\},\{1,4\},\{2,4\},\{3,4\}\}$ based on all frequent itemsets found so far. The third iteration similarly verifies the candidates under indexes 4 and 5, and generates a new one under index 10. Note, that during the candidate generation phase we delete all the frequent itemsets with `furthest_C` value equal or smaller than the index of a candidate being currently verified. Hence, on the 4^{th} iteration we delete the frequent itemset $\{1\}$ and its cover

vector after verifying the candidate {1,4}, as the latter is the last candidate to depend on an itemset {1}. We continue analogously until the last candidate {1,2,3,4} is verified. The same applies to the Figure 10.

Security analysis. Note that no matter how many candidate itemsets we verify at once, we will eventually reach the state, where we have verified the whole candidate tree with i-element candidates on the i^{th} depth level and found all the frequent itemsets. For each candidate we will have computed a cover, summed it, compared the sum with the threshold and declassified the comparison result telling us if the candidate itemset was frequent. Nothing else is declassified but the comparison results. Consequently, to prove that the algorithm is secure we must show that the outputs and the declassified values do not leak more information than needed. Hence, the analogous discussion applies as for *Apriori*.

6.7. Performance evaluation of FIM algorithms

To test the performance and compare the discussed privacy-preserving versions of FIM algorithms, we implemented them in the SECREC language and executed with various parameters on the SHAREMIND platform. The testing environment was the same as described in Section 6.2 except that the underlying cryptographic protocols of the SHAREMIND platform were improved. Memory consumption was measured by periodically asking the Linux OS for the virtual memory size of the data miner process at a single computing node.

We tested the algorithms on the Mushroom dataset from the UC Irvine Machine Learning Repository. The dataset contains 119 columns (items) and 8124 rows (transactions) with overall data density of 19.3%. The results depicted on Figure 11 clearly show, that *Apriori* is the fastest and *Eclat* is the slowest of algorithms. However, the memory consumption of *Apriori* is the biggest while in case of *Eclat* it is the lowest. Most importantly, as we expected it to be, our *Hybrid-Apriori* algorithm lies right between the first two in both speed and memory consumption. The results can be easily explained.

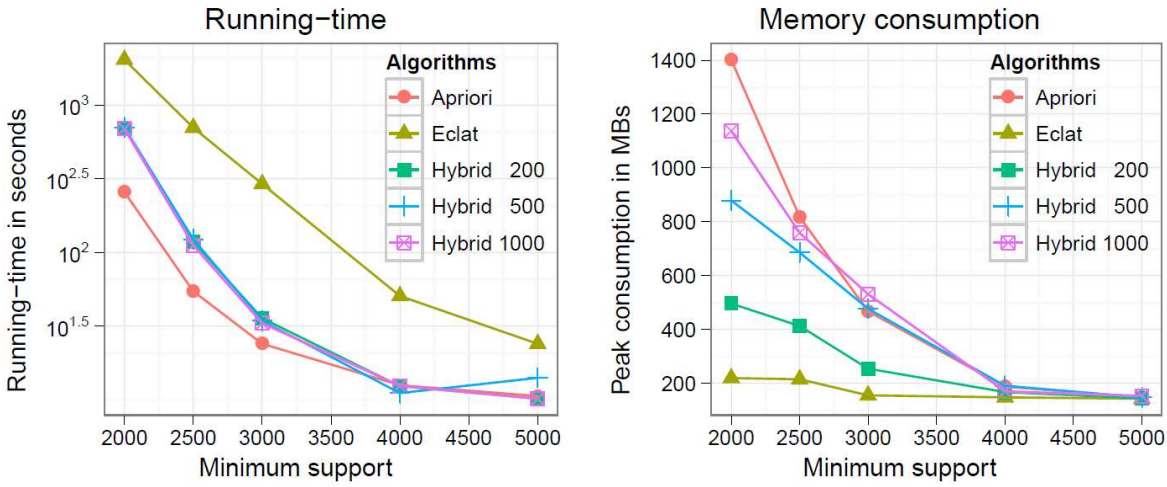

Figure 11. Execution time and memory consumption on the Mushroom dataset.

Apriori. Under the hood *Apriori* tries to parallelize operations as much as possible. Each iteration of the main loop consists of a single multiplication instruction followed by a single comparison instruction. This results in fast performance but excessive memory consumption, especially when the number of frequent itemsets is really large. The issue is amplified even further in the privacy-preserving version of the algorithm, as it stores not only the list of discovered frequent i-element sets but also the corresponding cover vectors.

Eclat. During every recursion step, *Eclat* verifies all direct descendants of a single parent frequent itemset (lines 2-5). The number of those direct descendants is at most one less than the number of frequent itemsets found in previous recursion step (see Figure 8). This is usually much less than the number of candidates on the same level of a candidate tree. Therefore, *Eclat* does many smaller vectorized multiplications and comparisons, which makes it slower than *Apriori*, but much memory friendlier.

Hybrid-Apriori. As we limit the number of candidates to be verified, we effectively limit the number of cover vectors that are stored in memory during the computations. To verify less candidates also means to have less vectorization. Hence, the multiplication and comparison operations are executed on smaller amounts of data but more often. This gives a drop in performance compared to conventional *Apriori*. However, the level of parallelization is higher than it is in case of *Eclat*, which is why the latter is still slower. Note, that we experimented by running Hybrid-Apriori with three different parameters, but all of them had roughly the same execution time. This can be explained by the fact that the efficiency of vector operations varies depending on the number of operations executed in parallel. With large enough vectors these changes are marginal and running-times are similar for all parameter values of the hybrid algorithm. They do, however, have a strong effect on the memory usage and therefore this algorithm reaches the goals set to it.

7. Related work and future plans

Related work. Concurrently with the work presented in this thesis, a number of related work projects were developed. The first project by Jaak Ristioja [21] concentrates on creating a framework for automatic analysis of an imperative privacy-preserving programming language, such as SecreC. As the privacy-preserving algorithms grow increasingly large and complex, it becomes hard to keep track on the amount of unwillingly leaked information in various declassification locations throughout the code. In such cases the need for an automatic analysis arises. Furthermore, the analysis results of the framework are formally correct and can be computed very fast, allowing the algorithm developer to concentrate on fixing the problematic code locations instead of manually searching and verifying them.

The SecreC programming language has evolved to the point, where it can be used for solving real problems. Hence, a good set of programming and debugging tools is mandatory for efficient development. The second important related work by Reimo Rebane [22] is dedicated to building an integrated development environment for the SecreC programming language, called SecreCIDE. The IDE currently supports separate projects for various programs. The programmer can edit both SecreC and Sharemind assembly source code while assisted with syntax highlighting and auto-completion. SecreCIDE also provides a simple interface for compiling SecreC programs into the assembly, and executing the code in debugging mode on the Sharemind platform.

Future plans. Although the language developed in this thesis can already be used in real-life situations, it is far from being complete. There are several improvements that can be made to the language to make it more robust.

Currently the data in SecreC can only be passed to functions by value. The author is concerned, that it is not the best option for two reasons. Occasionally, there is a need to create functions that operate directly on the data passed to them. It may also help reducing the code repetitions. Therefore, one of the next goals is to add a pass-by-reference support to the language.

The SecreC standard library also needs to be reworked. Many functions for many data types are still not implemented and doing so is one of the goals. Additionally, the function declarations and definitions should be moved from the compiler to separate editable files. This would introduce the header and source file concept similar to C language, and allow users to create their own libraries for using and sharing.

The compiler of the SecreC language can be improved as well. Currently it produces the assembly code that can be compacted by removing segments of code that do not do anything, or replacing the segments that can be done in a more efficient way. Hence the peephole optimizations must be added to the compiler. Automatic vectorization or even whole-program optimizations must also be considered, so the programmer would not have to write hard-to-understand programs in order to make them more efficient.

Finally, the ongoing research on the privacy-preserving data mining algorithms is conducted to find more efficient and clever ways to solve data mining problems in MPC.

8. Conclusion

In this thesis we present a higher level privacy-aware programming language called SECREC. Our goal was to make the development of advanced privacy-preserving algorithms as simple as possible by hiding most of cryptographic building blocks from the programmer enabling to concentrate on the business logic. We achieved this goal, and in this work we present the description of the language and its compiler. We also show that SECREC is usable for solving practical problems, and present novel privacy-preserving versions of four data mining algorithms.

The main innovation of the SECREC language is the distinction of public and private data. To accommodate this distinction, the language is based on the hybrid execution model with a different computation environment for each of the security types. We chose the SHAREMIND multi-party computation platform to act as a private computation environment. Therefore, we constructed a compiler that translates the SECREC language into the SHAREMIND assembly.

Although SecreC hides most of the underlying cryptography and architecture, the developer has to rely on several aspects and programming techniques in order write secure and efficient algorithms. We discussed that private data should be declassified as little as possible to maximally preserve the data privacy. While the control flow of the program must not be affected by private data, it is still possible to use techniques like oblivious selection to make branching decisions and hide the chosen branch from the observer. We also reasoned that data aggregation techniques must be used to maximize the measure of uncertainty of published output results. It is also important to parallelize the operations smartly for better execution-times.

Finally, we presented several practical use cases of our new privacy-aware language. We designed and implemented in SECREC novel privacy-preserving versions of four data mining algorithms: *Histrogram*, *Apriori*, *Eclat* and *Hybrid-Apriori*. We also verified the security and tested the performance of these algorithms in a setup of the SHAREMIND platform. All four algorithms proved to be secure and usable for solving real-life problems.

SECREC – privaatsuseteadlik programmeerimiskeel

Magistritöö (30 EAP)

Roman Jagomägis

Resümee

Informatsioon mängib meie ühiskonnas olulist rolli. Vastavalt selle täpsusele, tähendusele ja potentsiaalsele kasulikkusele võib informatsioonil olla väärtus. Andmete avalikustamisel me vältimatult seame oma huvid riski alla. Selle tagajärjel võivad kannatada ka kolmandad osapooled. Seega on andmete privaatsuse säilitamine saanud tänapäeval oluliseks ülesandeks.

Tihti on võimalik saavutada lisaväärtust kombineerides erinevatest allikatest kogutud salajasi andmeid. Näiteks meditsiinilised asutused koguvad ravimite arendamiseks andmeid haiguste ja geeniekspressiooni kohta. Konkureerivad ettevõtted võivad aga koostööd tehes avastada uusi teadmisi oma klientide kohta. Selline tegevus aga tekitab küsimusi andmete privaatsuse kohta, kuna andmeid koguvad osapooled võivad neid pahatahtlikult ära kasutada. Selliste probleemide vältimiseks on arvutiteadlaste poolt loodud mitmeid tehnilisi turvalistel ühisarvutustel põhinevaid lahendusi, nagu SHAREMIND, VIFF ja FAIRPLAYMP, kuid viimased pole oma keerukuse tõttu reaalsetesse rakendustesse jõudnud.

Käesolevas magistritöös kirjeldame privaatsusetundlikku kõrgtasemelist programmeerimiskeelt SECREC. Meie eesmärgiks oli teha keeruliste privaatsust säilitavate algoritmide loomine võimalikult lihtsaks, peites programmeerija eest ära enamik krüptograafilisi ehitusplokke ja lubades tal keskenduda äriloogikale. Me saavutasime oma eesmärgi ning selles töös esitame keele ja selle kompilaatori kirjeldused. Lisaks näitame, et SECREC keel on kasutatav praktiliste probleemide lahendamiseks ning esitame uudsed privaatsust-säilitavad versioonid neljast andmekaevanduse algoritmist.

SECREC keele peamine innovatsioon seisneb avalike ja salajaste andmete keele tasemel eristamises. Selle võimaldamiseks põhineb keel hübriidsel käivitusmudelil avaliku ja privaatse arvutuskeskkonnaga. Privaatseks arvutuskeskkonnaks valisime SHAREMINDi platvormi. Ehitasime ka kompilaatori, mis transleerib SECREC keelt SHAREMINDI assembleriks.

Kuigi SecreC keel peidab enamuse allolevast krüptograafiast, peab programmeerija turvaliste ja efektiivsete algoritmide arendamiseks tuginema mitmele programmeerimise tehnikale. Antud töös väidame, et salajaste andmete privaatsuse säilitamiseks tuleb andmeid avalikustada võimalikult vähe. Leiame, et progammi arvutusvoog ei tohi sõltuda salajastest andmetest. Küll aga on võimalik hargnemistel valitud haru vaatleja eest peitmiseks kasutada tehnikaid, nagu näiteks peitvalikut (*oblivious selection*). Lisaks põhjendame, et algoritmide väljundi määramatuse tõstmiseks tuleb kasutada erinevaid aggregeerimistehnikaid. Samuti soovitame programmeerijale algoritme jõudluse tõstmiseks paralleliseerida.

Lisaks kirjeldame töös uue privaatsusetundliku keele jaoks mitut praktilist kasutusjuhtu. Me koostasime ja realiseerisime SECREC keeles uued privaatsust-säilitavad versioonid neljast andmekaevandusalgoritmist: *Histrogramm*, *Apriori*, *Eclat* ja *Hübriid-Apriori*. Oleme samuti tõestanud algoritmide turvalisuse ning teinud jõudlusteste SHAREMINDI juurutusmudelis. Kõik neli algoritmi osutusid kasutuskõlblikeks päriselu ülesannete lahendamisel.

Autor avaldab tänu oma juhendajale, kelle pühendumus ja põhjalikkus olid hindamatuks panuseks käesoleva magistritöö valmimisele.

References

[1] *RT I 2007, 24, 127, Personal Data Protection Act (Estonia)*, 2008.

[2] *L 281/31, Data Protection Directive 95/46/EC*, 1995.

[3] "Privacy Framework," *ISO/IEC 29100*.

[4] "Privacy Reference Architecture," *ISO/IEC 29101*.

[5] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, LNCS 5283, pp. 192-206, Berlin, Heidelberg: Springer-Verlag, 2008.

[6] "Virtual Ideal Functionality Framework," http://viff.dk, 2007.

[7] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pp. 257-266, New York: ACM, 2008.

[8] D. Bogdanov, R. Jagomägis, and S. Laur, "Privacy-Preserving Applications in the Hybrid Execution Model," Submitted, 2009.

[9] D. Bogdanov, R. Jagomägis, and S. Laur, "Sharemind: a practical toolkit for privacy-preserving data mining," Submitted, 2010.

[10] A. C.-C. Yao, "Protocols for Secure Computations (Extended Abstract)," *23rd Annual Symposium on Foundations of Computer Science*, pp. 160-164: IEEE, 1982.

[11] D. Malkhi, N. Nisan, B. Pinkas *et al.*, "Fairplay - a secure two-party computation system," in SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium, 2004, pp. 287–302.

[12] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pp. 503-513, New York, NY, USA: ACM, 1990.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pp. 1-10, New York, NY, USA: ACM, 1988.

[14] I. Damgård, M. Geisler, M. Krøigaard *et al.*, "Asynchronous Multiparty Computation: Theory and Implementation," *PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography*, LNCS 5443, pp. 160-179: Springer, 2009.

[15] R. Jagomägis, "A programming language for creating privacy-preserving applications," Bachelor's thesis, Institute of Computer Science, University of Tartu, 2008.

[16] T. Parr. "ANTLR - Another Tool for Languate Recongnition," http://www.antlr.org.

[17] D. Bogdanov, R. Jagomägis, and S. Laur, *Privacy-preserving Histogram Computation and Frequent Itemset Mining with Sharemind*, Cybernetica research report T-4-8, 2009.

[18] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 207-216: ACM, 1993.

[19] R. Agrawal, H. Mannila, R. Srikant *et al.*, "Fast Discovery of Association Rules," *Advances in Knowledge Discovery and Data Mining*, pp. 307-328: AAAI/MIT Press, 1996.

[20] M. J. Zaki, S. Parthasarathy, M. Ogihara *et al.*, "New Algorithms for Fast Discovery of Association Rules," *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pp. 283-286: AAAI Press, 1997.

[21] J. Ristioja, “An analysis framework for an imperative privacy-preserving programming language,” Master's thesis draft, Institute of Computer Science, University of Tartu, 2010.

[22] R. Rebane, “An integrated development environment for the SecreC programming language,” Bachelor's thesis, Institute of Computer Science, University of Tartu, 2010.

Appendix 1: The source code of the SecreC compiler

The SECREC scanner and parser grammars for the ANTLR parser generator, as well as the assembly code generation schemes are included on the CD provided with this thesis. The corresponding files and their locations are as follows.

The source code can be found in the src subfolder of the secrec-compiler folder. The source code consists of the Java application that bootstraps the compiler and the ANTLR files that specify the language and its transformation rules.

The following table describes the ANTLR files and their purpose.

Filename	**Purpose**
SecreC.g	The language grammar and parser. This file is used during the parsing of the program.
SecreCTree.g	The tree grammar for the language. This file is used for tree transformations in the compiler.
SecreCTree.stg	The code generation templates for translating the abstract syntax tree into Sharemind assembly.

The gen and src subfolder of the src folder contain the ANTLR generated code and hand-written code respectively. The generated code consists of the lexer, the parser and the tree rewrite grammar. The hand-written source code is, among other things, responsible for handling input-output, checking types and inserting standard library definitions.

Appendix 2: The SECREC compiler software

Introduction

This thesis is accompanied by software that can be used to demonstrate the results. In this Appendix we explain what software is provided and how to use it. Note that the source code is not provided for all components as the binary versions can be tested more easily. While the compiler is written in Java and can be run on several platforms, only Microsoft Windows versions of the Sharemind virtual machine are provided. These have been tested on various, but not all versions of Windows so the author cannot guarantee the successful execution of all the software components.

Installing the software

Extract the files from `SecreC-SDK-1.95-beta.zip` to any folder. Three folder are created:

1) `secrec-compiler` contains a prebuilt version of the SecreC compiler with source code
2) `SecreCIDE` contains a prebuilt version of the SecreC IDE
3) `sharemind-vm` contains a prebuilt version of the Sharemind virtual machine with some applications
4) `algorithms` contains the SecreC programs that were used in the case studies

Note: The provided software is a debug build for software that is still being developed. For that reason the binaries and required dynamic libraries are very large.

Compiling SecreC programs on the command line

Make sure you have the Java runtime (version 1.5 or newer) installed and it is in your path.

Take the following steps to compile the SecreC algorithms on the command line.

1) On the command line, enter the '`algorithms`' folder.
2) To compile a SecreC code file (.sc), enter the command
   `compile code.sc code.sa`
   This compiles the SecreC source file into Sharemind assembly (.sa).
3) The file `code.sa` will be created that contains the Sharemind assembly version of the algorithm.

Running the assembly code on the Sharemind virtual machine

Copy the compiled `.sa` files into the `sharemind-vm` folder. Copy the compiled histogram algorithm into the '`scripts`' subfolder of each miner folder (`miner1`, `miner2` and `miner3`).

Take the following steps to run the Sharemind virtual machine

1) In Windows Explorer, enter the `sharemind-vm` folder. Run `miners.bat` in the `sharemind-vm` folder. This will run three standalone miners that will try to connect to each other. They are ready to run code when the "Start your applications." message is displayed.
2) The miners can be stopped by closing the window or pressing the Ctrl+C key combination.

You can now upload and run the assembly code on the virtual machine. The Apriori, Eclat and Hybrid Apriori implementations are self-contained and can be run using the `ScriptBenchmark` tool. Make sure that the Sharemind virtual machine is running and take the following steps:

1) On a command line, enter the `sharemind-vm` folder and run the following command:
   `ScriptBenchmark --upload apriori_optimized.sa`
   This will upload the file `apriori_optimized.sa` and run it.
2) Feel free to use different filenames to test other FIM algorithms.

Note, that the histogram code requires parameters to the main function and cannot be run using `ScriptBenchmark`. Instead, run the `HistogramCompute` application. This application requires testing data which has been provided in the archive. It does not require command line parameters.

Setting up SecreCIDE

Start the SecreCIDE application in the `SecreCIDE` folder. Take the following steps to configure it.

1) Open the Options menu from `Tools->Options`.
2) Enter or select the location of the `secrec-compiler` folder for the variable Sharemind Path.
3) Select the location of the `secrec-compiler` folder for the variable Additional Classpath.

You should now be able to compile and run programs from SecreCIDE.

Compiling and running SecreC programs with SecreCIDE

Take the following steps to compile a SecreC program in SecreCIDE.
1) Open a SecreC code file (`File->Open`).
2) Click the Build button on the toolbar. It looks like a brick wall.
3) The filename for the compiled version will appear in the left-side list.

To run programs from SecreCIDE, make sure that the Sharemind virtual machine is running. Take these steps:

1) Select a compiled file with a `.sa` extension in the file list. Use one of the algorithms that does not require parameters (Apriori, Eclat, Hybrid-Apriori).
2) Click the Run button on the toolbar. It looks like a white running man in a green circle. The program should run. You can verify it by looking at the Sharemind miner windows.
3) The FIM algorithms results are shown in the Controller log.

To run the histogram example, you need to provide the algorithm with some required parameters.

1) Right-click the histogram assembly code (the file with the extension `.sa`) in the file list on the left. Choose Set Runtime Arguments. A new window will open.
2) Enter three parameters:
   Name: table *Type*: PUBLIC_STRING *Value*: enter one from the following: *answers100, answers1000, answers10000, answers100000*
   Name: column *Type*: PUBLIC STRING *Value*: enter one of the following *twoanswers, threeanswers, fiveanswers* and *tenanswers*.
   Name: choices *Type*: PUBLIC_INT *Value*: enter the number you chose for the column (eg if you chose *threeanswers*, enter 3)
3) Click the run button on the toolbar. The algorithm should complete and the results should be shown in the Results tab.

Additional notes

SecreCIDE has support for debugging assembly execution and observing data in the registers and stack of the Sharemind machine. While this feature is not complete, it may provide insight into the working of the compiler and the IDE.
This development version of IDE does not have the integrated documentation with it.