ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

JOS System for Numerical Multi Party Computation

Author:
Bin Lu

Supervisors:
Prof. Serge Vaudenay
Dr. Johannes Schneider

Master Thesis Project

in the

Security and Cryptography Laboratory
School of Computer and Communication Sciences
and
Software Systems Group
ABB Corporate Research Center in Switzerland

March 10, 2016

Abstract

Secure multi-party computation (MPC) is one of the most promising approaches to ensure confidentiality of data used in computations in an untrusted environment. In MPC a client outsources computations on confidential data to a network of servers. The client does not trust a single server, but believes that multiple servers do not collude. In this project our main focus was on efficient numerical computations. For that purpose we derived new protocols and extended existing protocols for various operations based on the JOS scheme, which is based on linear secret sharing. The main contribution is an implementation of the JOS system with our extensions covering several novel aspects that have not been addressed in prior MPC systems, e.g. dependence graph analysis and code parallelization, code generation at runtime and sub-domain privacy.

Contents

Chapter 1

Introduction

1.1 Motivation

Cloud computing has been a fast growing industry in the last few years and it is anticipated to continue growing at a similar rate. Today, almost all major IT companies have realized this trend and a multitude of solutions are available.

Cloud computing research has delivered many concepts focusing on reliability, scalability and distributed system design, while security issues have not been addressed sufficiently. First, personal data uploaded by users may not be encrypted if it is used for computations. Since the full power of cloud computing is only released when conducting computations, confidentiality of data is at risk in all practical systems. In 2014, iCloud was attacked by hackers, having a collection of private pictures of various celebrities leaked. Dropbox was breached in the same year and had over 7 million of its users' passwords stolen by hackers. Additionally governments can compel cloud computing vendors to help decrypt the data from any suspicious user.

Incidents like iCloud 2014 leak, Dropbox security breach as well as other attacks lead us to consider security mechanisms and requirements of cloud computing. Ideally, users encrypt data using provably secure encryption schemes(not relying on any assumption) and they only upload encrypted data to the cloud in order to prevent unauthorized access(by hackers or vendors). Furthermore, it should be possible to carry out efficient computations on the encrypted data only (without having access to keys). As we shall discuss in Section 1.2, none of the existing schemes comes even somewhat close to fulfill all requirements for arbitrary computations. However, secure multi-party computation seems to be the most suitable candidate.

In this thesis we put particular emphasis on numerical operations. For many applications such as automation systems numerical operations are essential. For example in a car manufacturing plant, sensors throughout the production line collect data that is sent to a local trusted database and after preaggregation a reduced data set is sent into an untrusted cloud environment for computation. Typical operations on such sensor data range from simple statistics like averages, standard deviation etc. to more complex analysis like Fourier analysis involving trigonometric functions. Overall, numerical operations are often the biggest part of all computations and thus deserve special attention. In this thesis we implement and extend a variety of numerical operations.

1.2 Secure Computation Methods

Approaches to carry out computation on cipher-text include homomorphic encryption, secure cryptoprocessor, differential privacy and secure multi-party computation(MPC). We briefly define them and discuss their strengths and weaknesses. Our main focus lies on MPC since we consider it to be the most suitable for practical secure computations.

1.2.1 Homomorphic Encryption

The idea of homomorphic encryption comes from homomorphism of algebraic structures e.g. groups and rings. A homomorphism maps two structures while ensuring structure-preserving, e.g in RSA, the encryption of a message s is defined as $ENC(s) = s^k \mod m$, where k and m composes public key and thus the property of homomorphism is $ENC(s_1) \cdot ENC(s_2) = (s_1^k \mod m) \cdot (s_2^k \mod m) = (s_1 \cdot s_2)^k \mod m = ENC(s_1 \cdot s_2)$.

Partially homomorphic encryption systems only provide limited number of operations on cipher-text. A fully homomorphic encryption(FHE) scheme is introduced in [4] supporting evaluation of arbitrary boolean circuits. However, FHE is too slow for most practical applications despite rapid improvements.

1.2.2 Secure Cryptoprocessor

We can use a dedicated computer on a chip or microprocessor called secure cryptoprocessor to carry out computation on cipher-texts. It is integrated into a system with multiple physical security modules and gives the system a degree of tamper

resistance. A secure cryptoprocessor will not output decrypted data or decrypted program instructions in an environment where security cannot always be maintained.

Though secure cryptoprocessor is useful, it is unable to ensure adequate security. It is vulnerable to side channel attacks as well as physical attacks. Furthermore, secure cryptoprocessors are costly.

1.2.3 Differential Privacy

Differential privacy makes it possible to provide global, statistical information about the data while preserving the privacy of users' records. However, with differential privacy, perfect security and efficient, accurate computation cannot be guaranteed at the same time.

1.2.4 Secure Multi-party Computation

Secure multi-party computation (MPC) is a subfield of cryptography with the goal of designing protocols for two or more parties to perform joint computation, without revealing to each other their private inputs and outputs. In a MPC model, we have a function $f(x_1, x_2, ...x_k) = (y_1, y_2, ...x_k)$ and k parties $p_1, p_2, ...p_k$ which receive $x_1, x_2, ...x_k$ respectively as input. Each party wants to compute the correct result $y_1, y_2, ...x_k$ [9]. JOS scheme is a MPC scheme which requires at least three parties. It is possible to evaluate arbitrary operations efficiently in JOS system. Moreover, it provides perfect or statistical security.

Security Model

Semi-honest Adversary and Malicious Adversary The two main adversary models in MPC are semi-honest adversaries and malicious adversaries. Semi-honest adversaries follow the prescribed protocol but try to glean more information than allowed from the protocol transcript. Malicious adversaries can run any efficient strategy in order to carry out their attack.

Perfect Security and Statistical Security The goal of secure computation systems is to provide perfect security for arbitrary operations. Perfect security means that the cipher-texts reveal no information about the content of the plain-texts. In another word, attackers are not able to guess what the plain-texts and keys are, no matter how many cipher-texts they acquire. Perfect security also means that the probability

distribution of cipher-texts does not depend on the plain-texts or keys. However, most cryptographic methods are only statistically secure. Statistical security means that it is impossible to break a cipher-text with the current computer technology within a period short enough to be practicable.

1.3 Contribution

In this project we derived new protocols and extended existing protocols for numerical operations based on JOS scheme. New protocols include OR operation and fixed-point integer multiplication. The protocol for comparison with zero(i.e. EqualZero in [2]) is slightly adjusted and an auxiliary, very efficient protocol AddModToAdd is introduced in order to assist fixed-point integer operations. Algorithms for arithmetic operations with one non-confidential and one confidential operand are given.

Additionally, a prototype of the JOS system was implemented with extensions covering several novel aspects that have not been addressed in prior systems:

- Communication rounds are reduced by analyzing the dependency graph of a program and executing instructions in parallel. The test results in Section 6.2 shows that the performance is improved by a factor of 70 for certain cases.
- We trade off privacy and performance by introducing the concept of sub-domain privacy. It allows to treat variables non-confidential depending on the value they store. The pattern matching example in Section 5.3 shows that improvements of a factor of 100 are feasible by introducing sub-domain privacy on variables.
- JOS system can generate code at runtime. If-else branching is translated into different code at runtime depending on whether variables are treated confidentially or not, which is only known at runtime. Runtime code generation helps to reduce the amount of instructions and to choose more efficient protocols depending on the needed degree of security.

1.4 Overview

Chapter 1 gives a general introduction of this project. Chapter 2 presents previous works of MPC schemes and MPC systems as well as the concept of sub-domain privacy.

Chapter 3 gives an overview of JOS system. The system can be divided into two subsystems: compiler and runtime. In order to compose a program, we define a domain specific language and implement the compiler in Section 3.3. Compiler generates

a syntax tree which will be translated into intermediate code. Intermediate code translation is discussed in Section 3.4. The architecture of JOS runtime is presented in Section 3.5, including protocol module and scheduler module. In Section 3.6, we describe the network module which assists the execution of protocols at runtime.

Protocols extension and improvement are introduced in Section 4.1. In addition to primitive operations , we also support compound operations, namely if-else branching and while loop, which is discussed in Section 4.2.

In Chapter 5, we introduce approaches to improve the performance of JOS system. Parallelization allows instructions to be executed concurrently. Asynchronous protocol overcomes busy waiting problem. Sub-domain privacy trades off privacy and performance.

Finally results of individual protocol test and complicated program tests are presented in Chapter 6.

Chapter 2

Related Work

In this chapter, we will first introduce previous works of MPC and then present JOS scheme and briefly compare it with other MPC schemes. Moreover, two primitive protocols - Addition and AND - are presented to demonstrate JOS scheme. Finally, we discuss several MPC systems and the concept of sub-domain privacy.

2.1 MPC Schemes

Bar-Ilan et al's paper [7] introduced a method to compute unbounded fan-in multiplicative (or AND) gates using constant rounds. By using this method, it is possible to evaluate any function with constant rounds, while size of messages is proportional to the size of a constant-depth, unbounded-fan-in circuit describing the function.

In Yao's paper [10] , a solution for evaluating a boolean circuit is proposed. In the computation, two parties are required, i.e. party A containing private inputs from itself and another party B. A does not learn anything about the inputs of B and B does not learn anything about the circuit or the input of A. Based on Yao's work, various protocols has been proposed to improve certain aspects of Yao's protocol, e.g. Goldreich-Micali-Widgerson (GMW)[11] uses oblivious transfer to compute Boolean circuits. Values are encrypted such that each party holds parts of the non-encrypted value.

SHAREMIND [8] allows secure computation over the ring of 32-bit integers for three parties. It uses an additive secret sharing scheme over the ring $\mathbb{Z}_{2^{32}}$, i.e. for a secret x each party P_i obtains a share x_i such that $\sum x_i \mod 2^{32} = x$. In the SHAREMIND framework, all modular reductions occur modulo 2^{32} such that results always coincide

with the standard 32-bit integer arithmetic. As secrets are shared with additive sharing in the SHAREMIND framework, implementing addition and multiplication with a public constant c is straightforward by doing only local operations. For example, to compute $a+b$ with shares (a_i, b_i), each party simply computes $a_i + b_i$.

2.1.1 JOS

JOS scheme is a MPC scheme, which employs client-server model to evaluate the function. More specifically, the client is a trusted party which generates all the inputs x_i and sends them to the corresponding servers p_i. The servers jointly compute the results y_i without revealing the secrets. y_i will be eventually sent back to the client, differing from the classical MPC models, where each party holds a secret and the output should be known by (at least) one party.

In JOS scheme, at least three parties are required to implement all protocols. One party plays the role as encrypted value holder(EVH), one as key holder(KH). The remaining parties assist KH and EVH to carry out the execution. On principle, all parties should be able to communicate with the client. However, helpers can solely be invoked by EVH or KH and do not return results to the client. Communication between client and helpers is thus not necessary. [1] and [2] show that three parties are sufficient for JOS protocols.

Security

JOS scheme provides perfect or statistical security in the semi-honest model[1]. To ensure security, KH only stores keys but has no access to the cipher-texts while EVH only stores encrypted values but has no keys. The helper is not allowed to receive any paired key and encrypted value during one running of a program, preventing helper from decrypting the encrypted values. When one party has to share secret with others, double encryption is required and new keys and encrypted values will be delivered to the corresponding parties. JOS system involves 3 parties, which means at most one party could be corrupted and parties should not collude with each other to reveal the secret.

Given that all parties will follow protocols and do not collude, we only need to make sure no information is leaked during secret sharing. JOS uses liner secret sharing among two out of three parties, i.e. two shares will be generated. These two shares

[1] For integer operations, perfect security is guaranteed. However, fix-point integer protocols are statistically secure.

are the encrypted values and the corresponding keys. This is analogous to linear secret sharing, but the distinction of keys and encrypted values becomes relevant for certain operations like trigonometric functions. Different operations require different encryption schemes for their operands. For boolean operations (logical boolean operations and bitwise boolean operations), XOR encryption is required. For arithmetic operations and comparison operations, operands shoculd be AddMod encrypted.

Definition 1 *XOR Encryption: Given secret* $s \in \{0,1\}^n$*, we choose a random number* $k \in \{0,1\}^n$ *and compute* $ENC_k(s)$ *by applying bitwise XOR between s and k, i.e.* $ENC_k(s) = s \oplus k$*. The decryption* $DEC_k(s)$ *of cipher-text c is defined as* $DEC_k(c) = c \oplus k$

Definition 2 *AddMod Encryption: Given secret s in field* F_{2^n} *, we choose a random number* $k \in F_{2^n}$ *and compute* $ENC_k(s)$ *by adding s and k on* F_{2^n}*, i.e.* $ENC_k(s) = s + k \pmod{2^n}$*. The decryption* $DEC_k(s)$ *of cipher-text c is defined as* $DEC_k(c) = c + 2^n - k \pmod{2)^n}$

The above two encryption schemes provide perfect security. However, operands in fixed-point integer operations (Sin, Inverse and Fixed-point Integer Multiplication) are supposed to be encrypted with addition encryption without modulo, i.e. $ENC_k(s) = s + k$, $DEC_k(c) = c - k$. In this case, we can only guarantee statistical security by choosing k in a range significantly larger than that of s. In Section 4.1.3, we will show how this is implemented in a system that only provides XOR and AddMod encryption schemes by default.

Other security issues like authentication and confidential communication between parties are not the concern of JOS system.

Example: Addition

Addition only involves two parties - KH and EVH. Protocol is easy to understand and the complete process ranging from encryption of plain-text to decryption of result can thus be demonstrated clearly. Addition requires AddMod encryption for both operands. First, the client generates two pairs of shares: $(ENC_{k_a}(a), k_a)$ and $(ENC_{k_b}(b), k_b)$, where $ENC_{k_a}(a) = a + k_a \pmod{2^n}$ and $ENC_{k_b}(b) = b + k_b \pmod{2^n}$. $ENC_{k_a}(a)$ and $ENC_{k_b}(b)$ will be sent to EVH and k_a and k_b will be sent to KH. Next, EVH computes $ENC_{k_f}(a+b) = ENC_{k_a}(a) + ENC_{k_b}(b) \pmod{2^n}$ and KH computes $k_f = k_a + k_b \pmod{2^n}$. EVH and KH then send the results to the client. Finally,

the client decrypts the result from EVH with the key from KH by using the formula $DEC_{k_f}(c) = ENC_{k_f}(a+b) + 2^n - k_f \pmod{2^n}$.

Given two operands are independent from each other, all keys are one-time pad for encryption, EVH only has encrypted values, KH only has keys and Helper is not involved in the protocol, no party can derive a or b.

Example: AND

The first step of AND protocol is similar to addition protocol except that the two operands are encrypted with XOR encryption scheme, i.e. $ENC_{k_a}(a) = a \oplus k_a$, $ENC_{k_b}(b) = b \oplus k_b$. Given EVH has $ENC_{k_a}(a)$ and $ENC_{k_b}(b)$ and KH stores k_a and k_b, the next task is for EVH to compute $ENC_{k_f}(a \wedge b)$ while KH holds k_f. Following equation will help to derive $ENC_{k_f}(a \wedge b)$ and k_f.

$$a \wedge b = (ENC_{k_a}(a) \wedge ENC_{k_b}(b)) \oplus (k_a \wedge ENC_{k_b}(b)) \oplus (k_b \wedge ENC_{k_a}(a)) \oplus (k_a \wedge k_b)$$

The first term and the forth term can be computed by EVH and KH respectively. To compute the remaining two terms, Helper has to assist because KH cannot send k_a and k_b to EVH and EVH is not allowed to transfer $ENC_{k_a}(a)$ and $ENC_{k_b}(b)$ to KH. Values hold by EVH and KH need to be double encrypted before being send to Helper. Figure 2.1 shows the protocol of multiplication.

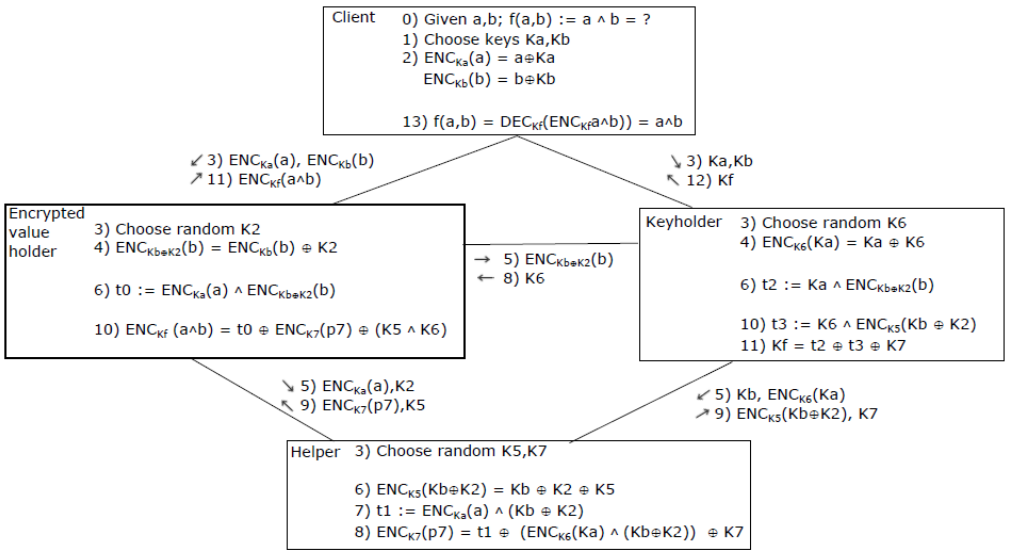

Figure 2.1 – Algorithm for an AND operation (figure from [1])

All parties generate one-time pad keys and encryption scheme is AddMod, which provides perfect security. EVH receives k_6 from KH which is an independent key. It also gets $NEC_{k_7}(p_7)$ and k_5 form Helper. k_5 is a random key. Since EVH learns nothing about k_7, it is impossible to derive p_7, which contains information about k_a and k_b. KH obtains $ENC_{k_b \oplus k_2}(b)$ from EVH and $ENC_{k_5}(k_b \oplus k_2)$ and k_7 from Helper. $ENC_{k_b \oplus k_2}(b)$ is a double encrypted value of b. Since $k_b \oplus k_2$ is encrypted with k_5, KH cannot derive k_2, such that b keeps confidential for KH. Helper receives $ENC_{k_a}(a)$ from EVH, but since k_a is encrypted by k_6, a remains secret for Helper. No information about b can be acquired by Helper. To sum up, none of the three parties has enough information to derive a and b.

2.1.2 Comparison of JOS and MPC Schemes

JOS is superior for some metrics to classic works such as Yao, GMW, SHAREMIND and Bar-Ilan et al. JOS is more efficient for boolean and arithmetic circuits. To evaluate an unbounded fan-in gate, Bar-Ilan et al [7] requires the message size to be proportional to the size of the constant depth of that gate. JOS also requires a constant number of rounds for computation of a fan-in gate but more communication. However, JOS has better performance for small fan-in gates.

To perform a multiplication (similar to AND), Sharemind requires 3 rounds and 27 messages each containing a 32-bit value. JOS require at most 2 rounds and 4 messages.

The round complexity of GMW is linear in the circuit depth. For circuits in disjunctive normal form such that the number of variables per clause is less than the logarithm of the security parameter, JOS requires less communication and local computation than Yao's scheme or its improvements.

2.2 MPC Systems

We will introduce several MPC systems based on different MPC schemes in this section. MPC systems usually provide a toolchain for automating, i.e. tools for compiling source code(defined in a domain specific language) and tools for code execution.

2.2.1 TASTY

TASTY is a two-party system that can generate protocols based on garbled circuits, homomorphic encryption and combinations of both. Users describe the computations to be performed on encrypted data in a domain-specific language, which will be

automatically transformed into a protocol by TASTY[16].

2.2.2 Fairplay

A two-party system called Fairplay is introduced in [18]. The system implements generic secure function evaluation and defines a high level procedural language called SFDL. Fairplay system also comprises a compiler of SFDL, which translates source code into a one-pass boolean circuit presented in a language called SHDL.

2.2.3 ABY

A mixed-protocol framework called ABY is implemented in [19] providing a feasible solution for secure two-party computation. In ABY, computation schemes based on Arithmetic sharing, Boolean sharing and Yao's garbled circuits are efficiently combined. Most cryptographic operations in ABY will be preprocessed such that the system can perform proper conversions between secure computation schemes. ABY supports several standard operations and protocols, i.e. computations on encrypted and non-encrypted data.

2.2.4 PICCO

PICCO is a multi-party system for translating a general-purpose program written in an extension of C into secured representation and executing the program in a distributed environment. Variables in PICCO can be declared as private [17].

2.2.5 SHAREMIND

Laud and Randmets' paper [15] introduced a MPC compiler for SHAREMIND scheme. In SHAREMIND framework, a set of primitive protocols are designed and more complicated operation are built on top of it. This feature allows efficient implementation of new protocols by reusing existing protocols. However, the set of primitive protocols still need to be maintained, e.g. adding new protocols to the set or propagating possible optimizations for a particular sub-protocol into larger protocols. In order to make protocol maintenance easier, a domain-specific language is defined and its compiler is implemented.

2.3 Compiler

Figure 2.2 shows a typical decomposition of a compiler into phases [12]. We have implemented the lexical analyzer, syntax analyzer, intermediate code generator and machine-independent code optimizer in JOS system. Semantic analyzer is not implemented because JOS language is simple and it does not support function declaration and complicated data types. Machine code generator and machine-dependent code optimizer are discarded because the execution of programs is carried out on a network of servers instead of local machines. JOS runtime takes intermediate code as input.

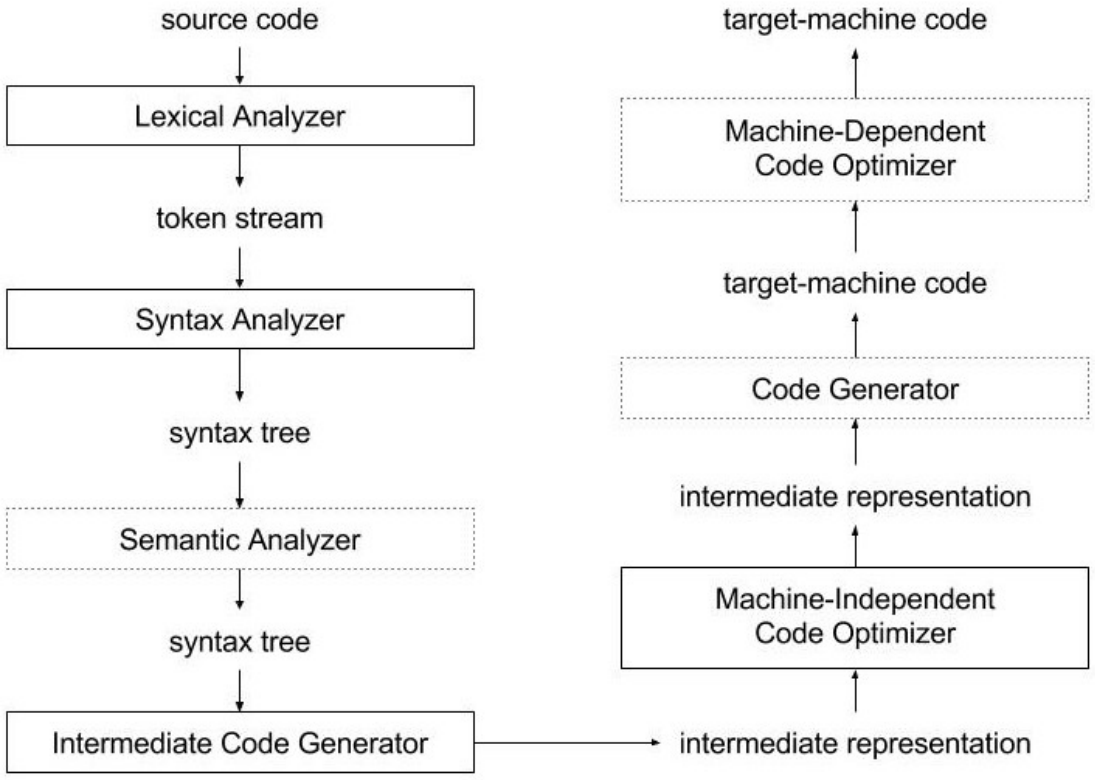

Figure 2.2 – Phases of a Compiler[12]. Dotted modules are not implemented in JOS system.

2.4 Sub-domain Privacy

Several secure computation schemes including JOS and feasible system implementations have been discussed in previous sections. Though many of them offer promising results with respect to performance for some applications, overall their applicability is still heavily limited due to communication latency between parties. JOS system has

minimized communication rounds but execution time is still enormous compared with native code execution.

To solve this problem, [3] introduces a novel approach called sub-domain privacy(SDP), which defines privacy depending on the value of a variable. When a variable is declared with SDP, the underlining value is not necessarily kept secret during the whole execution of the program but might be revealed depending on the concrete value it holds at runtime. Through this method, one can define an efficient program by trading off privacy and performance.

2.5 Summary

In this chapter, we have introduced JOS scheme as well as security in JOS system. Two examples - Addition and AND - show the design procedure of JOS protocols. More protocols are discussed in [1] and [2], which are Multiplication, Subtraction, XOR, OR, NOT, EqualZero, LessZero, Inverse(Log) and Sin. We will show that it is sufficient to derive and implement other operations based on these primitive protocols and thus compose a program.

Chapter 3

System Overview

In the project, we implemented a prototype of JOS system. It allows to execute complex sequences of operations. Only necessary interfaces of each module are exposed and the dependencies between modules are minimized. We can thus develop and maintain modules independently. In such a way, it is also easier to add components into the system, e.g. a new operation protocol or an intermediate code optimizer.

JOS system evaluates programs by outsourcing the code to third parties. Thus, JOS system should have the same functionalities as other source code interpreting and executing systems, which normally consist of two parts: compiler and runtime. Figure 3.1 shows the structure of JOS system. Interpreter is part of the compiler conceptually, as in most compilers intermediate code is generated directly from source code instead of the syntax tree for efficiency reasons. However, since compiler implementation is not the focus of our project and defining a complete programming language and implementing the corresponding compiler are obviously not trivial, we adopt the classic and comprehensible compiler architecture from [12], where intermediate code is generated from syntax tree. Interpreter module can thus be implemented separately from the other parts of the compiler. To sum up, JOS system contains four independent modules: compiler, interpreter, runtime and network. The network module assists communication between parties.

Figure 3.2 describes the work flow of program execution. First, the client translates source code into intermediate code, performs code encryption and sends the encrypted code to EVH and KH. Second, EVH and KH execute instructions in a synchronized order by invoking corresponding protocols. Third, the client waits for the results from EVH and KH and performs decryption.

In Section 3.1, we will introduce the main data structures, i.e. numeric, syntax tree and intermediate code. Implementation details about compiler will be discussed in Section 3.3. Section 3.4 focuses on intermediate code generation, optimization and encryption. In Section 3.5, we will show the most significant part of JOS system, the runtime module, which includes the JOS protocols and instruction scheduler. In the last section we will talk about the network module.

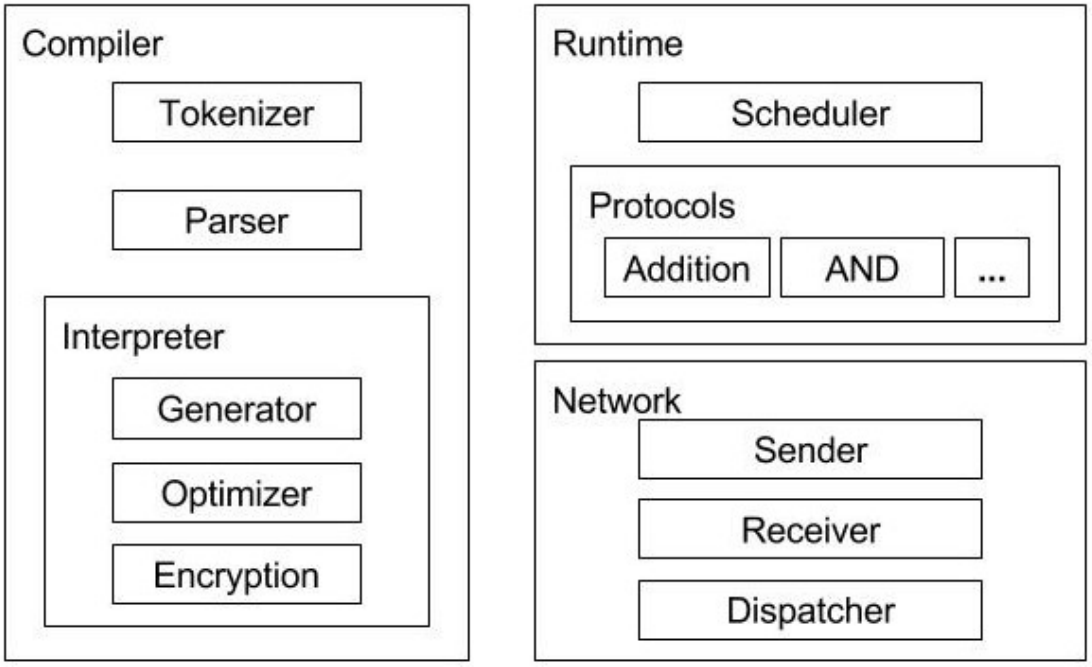

Figure 3.1 – JOS System Overview

3.1 Data Structures

In this section, we will introduce three data structures that are defined in JOS system. **Numeric** encodes all numeric data types, i.e. integer, decimals and booleans. **Syntax Tree** is generated by parser and serves as input for interpreter. **Intermediate Code** is the output of interpreter and it will be passed to runtime.

3.1.1 Numeric

A numeric object has three fundamental attributes(or fields in C#), which are internal storage for an integer, scaling information(scale bits) and encryption type. The first field stores all the bits of a value. The second field helps to represent a decimal data type(or fixed-point integer). The third field is necessary for coordinating operand

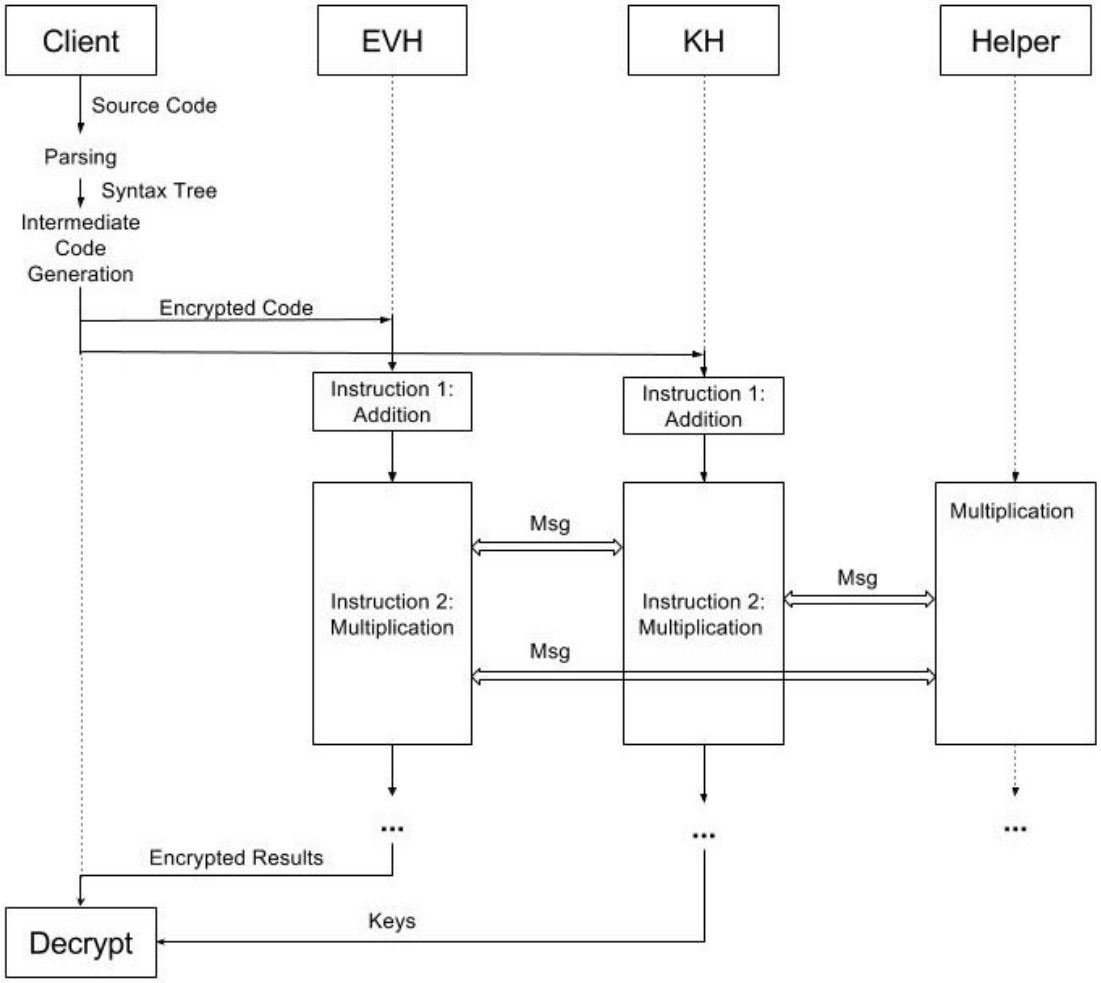

Figure 3.2 – Work Flow of Program Execution

values between instructions. Different operations in JOS system require different encryption schemes for operands and result. Each instruction decides whether to transform the encryption type of operands or not before carrying out the following execution.

Integer The size of integer is k where $k \in \{2^i | i \in \mathbb{N} \wedge i \geq 3\}$ [1]. Negative numbers are encoded with two's complement.

Fixed-point Integer The storage of bits and negative number encoding are exactly the same as integers. We can use the following formula to calculate the value, where vDecimal stand for the value of the fixed-point Integer, vInteger is the

[1] Several leading bits will be reserved as sign bits, i.e. the length of bits to encode an integer(say it is n) is shorter than k. The range of an integer is then $[-2^{(n-1)}, 2^{(n-1)} - 1]$.

integer value represented by the n bits(interpreted as signed integer)

$$vDecimal = vInteger/2^{\text{scale bits}}$$

For example, assume $n = 8$, decimal 0.5 is represented as 00000001 with scale bits set to 1.

Boolean True and False are equivalent to integer 1 and 0 respectively.

A set of arithmetic and bitwise boolean operations are defined for Numeric. We use big integer data type to perform the underling calculation and only the tailing k bits will be stored as the final result. For addition and multiplication, result truncating is equivalent to taking the module, which facilitates calculations on the field F_{2^k}.

For addition, subtraction, XOR, AND, OR operations, if both operands have the same scale bits, it is sufficient to perform operation on the underlying integers and keep their common scale bits. The result can be exactly represented with the same scale bits, as long as no overflow occurs, i.e. the sum of the two integers fits in the underlying integer type. If the numbers have different scale bits then one of them must be converted to the other before the operation. For example, to add 1 scaled by 1(0.5) and 1 scaled by 2(0.25), the first operand needs to be converted to 2 scaled by 2(0.5). Adding 2 scaled 2(0.5) and 1 scaled by 2(0.25) yields 3 scaled by 2(0.75).

To multiply two numerics, it suffices to multiply the two underlying integers and assume that the scale bits of the result is the c of their scale bits. This operation involves no rounding. For example, multiplying 2 scaled by 2 (0.5) and 3 scaled by 2(0.75) yields the integer 6 scaled by 4, that is $6/2^4 = 0.375$. Since underlying integer of numeric type has fixed bit length, result of multiplication can easily overflow if there are several successive operations. To solve this problem, the result has to be rounded. In this case, the result of the previous example must be divided by 2^{4-2} to yield either 1 (0.25) or 2 (0.5).

3.1.2 Syntax Tree

A syntax tree is a tree representation of the abstract syntactic structure of source code. It serves as an intermediate representation of the program. A simple program contains a sequence of statements. Statements in JOS language are assignment, if-else branching, while loop, block statement and return statement. Data structures for expressions are numeric literal, variable, unary expression and binary expression. Numeric literal and variable are terminals. Assignment is an atomic statement which

has a variable expression as result and an expression as value. IF-Else contains a condition field(expression), if-branch(block statement) and else-branch(block statement). While loop consists of a condition field(expression) and a list of statements(block statement). Block statement wraps statements sequence. A program is essentially a block statement. Return statement carries the names of variables that we want to compute. Unlike normal programming languages, JOS program can return several values at the same time. Figure 3.3 shows the syntax tree of Algorithm 1.

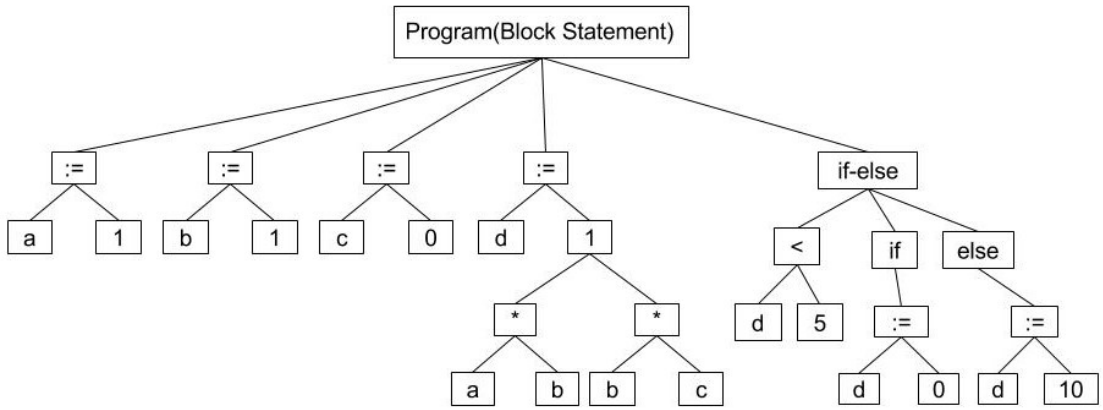

Figure 3.3 – Syntax Trees of Algorithm 1

3.1.3 Intermediate Code

There are three types of intermediate code(IC for short): assignment IC, If-Else IC and While IC.

Assignment IC is three-address code where result is a variable expression and two operands are terminal expressions.

If-Else IC has three lists of intermediate codes and a condition field. One IC list contains translated instructions from the condition expression if it is not a terminal. The result, which should be a terminal expression, is stored in condition field. The other two lists are used to hold intermediate codes translated from if branch and else branch.

While IC contains a list holding instructions translated from condition, a list of instructions for loop body and a field for condition.

3.2 JOS Language

We define a domain specific language including basic programming language components: arithmetic operations, boolean circuits, comparisons, assignment statements, if-else statements, while loops etc. The type of variables in JOS language is inferred from the value being assigned, e.g. in simple Algorithm 1, a is an integer and b is a fixed-point integer. We also define sub-domain privacy attributes for variables and if-else statements. Details about sub-domain privacy can be found in Section 5.3.

Following is the Grammar of JOS language in EBNF-like form.

$$
\begin{aligned}
&<program> ::= <block> \\
&<block> ::= \text{"\{"}, \{<declaration> | <assignment> | <if-else> \\
&\qquad\qquad | <while> | <return>\}, \text{"\}"}; \\
&<declaration> ::= \text{"var"}, <ident> \\
&\qquad\qquad [, \text{" = "}, <numeric\_expression> [, \text{"(none)"}]] \\
&\qquad\qquad [\text{"["}, <number>, \text{","}, <number>, [\text{","}, <number>], \text{"]"}], \\
&\qquad\qquad \{\text{","}, <ident>, \\
&\qquad\qquad [\text{" = "}, <numeric\_expression> [, \text{"(none)"}]] \\
&\qquad\qquad [\text{"["}, <number>, \text{","}, <number>, [\text{","}, <number>], \text{"]"}]\}, \text{";"} \\
&<assignment> ::= <ident>, \text{" = "}, <numeric\_expression>, \text{";"} \\
&<if-else> ::= \text{"if"}, \text{"("}, <boolean\_expression>, \text{")"}, \\
&\qquad\qquad [\text{"["}, (\text{"true"} | \text{"false"}), [\text{","}, <number>], \text{"]"}] <block>, \\
&\qquad\qquad [\text{"else"}, <block>] \\
&<while> ::= \text{"while"}, \text{"("}, <boolean\_expression>, \text{")"}, <block> \\
&<return> ::= \text{"return"}, <numeric\_expression>, \\
&\qquad\qquad \{\text{","}, <numeric\_expression>\}, \text{";"} \\
&<numeric\_expression> ::= [\text{"+"} | \text{"-"}], <numeric\_term>, \\
&\qquad\qquad \{(\text{"+"} | \text{"-"}), <numeric\_term>\} \\
&<numeric\_term> ::= <numeric\_factor>, \{(\text{"*"} | \text{"/"}), <numeric\_factor>\} \\
&<numeric\_factor> ::= <number> | <ident> \\
&\qquad\qquad | \text{"("}, <numeric\_expression>, \text{")"} | \text{"sin"}, <numeric\_expression> \\
&<boolean\_expression> ::= <boolean\_term>, \{\text{"||"}, <boolean\_term>\} \\
&<boolean\_term> ::= <boolean\_factor>, \{\text{"\&\&"}, <boolean\_factor>\} \\
&<boolean\_factor> ::= \text{"true"} | \text{"false"} | \text{"("}, <boolean\_expression>, \text{")"}
\end{aligned}
$$

$$| < numeric\_expression > , < logical\_operator > , < numeric\_expression >$$
$$< logical\_operator > ::= \text{ " <= "|" >= "|" < "|" > "|" == "|"! = "}$$

For regular expressions and statements, the grammar is exactly the same as C#. Additionally, it is possible to declare a constant to be unencrypted by appending a tag "(none)", e.g. "a = 3(none)" means number 3 will not be encrypted during code encryption phase. JOS language also allows us to attach sub-domain privacy information when a variable is declared, e.g. "var a [1, 100]" or "var b = 0 [1, 100, 0.9]".

3.3 Compiler

Though compiler implementation is not the focus of our project, it plays an important role in our project. First of all, in order to demonstrate a complete JOS system, which is the purpose of a prototype, compiler is necessary. Moreover, it simplifies the testing process. To evaluate a program, we need a syntax tree of the program, which can be generated in two ways. One approach is to manually assemble the program using data structures defined in the system. This approach does not require compiler, but it is error prone because users have to take over the work of a compiler. The other approach is to generate the syntax tree with the help of a compiler. Thus, users can concentrate on realizing the functionality of the program and leave the reaming work to the compiler.

3.3.1 Tokenizer

Source code is a stream of characters. The compiler first needs to correctly identify meaningful groups of characters which is called lexemes, e.g. a single "=" is an assignment symbol while "==" should be recognized as a logical operator.

We use regular expressions to match lexemes from the beginning of source code stream. A list of matching rules is defined beforehand. Each rule consists of a regular expression and symbol information. The order of rules has an effect on the output token sequence because rules are checked in turn until one matches. For example, if rule to match "=" goes before rule for "==", "==" will be identified as two successive "=". After each match, the corresponding characters have to be removed from the stream and a token will be created. A token holds the original characters and symbol name.

3.3.2 Parser

JOS grammar is in LL(1) form and thus it is possible to design a predictive parser which is a recursive descent parser that does not require backtracking. Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. One procedure is associated with each nonterminal of a grammar. Predictive parsing is a simple form of recursive-descent parsing in which the look ahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal. The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree[12]. In addition to generate a syntax tree, our parser is able to yield error messages to help users find out syntax errors.

During parsing, all variables declare in the program will be inserted into a table called VTable. Underlining data structure of VTable is a dictionary supporting concurrent access and it uses the names of variables as keys. The names of variables must be unique. Throughout the whole program, only one Vtable exists, which means variables declared in sub-scope, e.g. if branch of the if-else statement, will be visible outside the scope and thus the names should differ from the existing variables.

In common programming languages, division is regarded as a primitive operation. However, JOS system only contains multiplication and inverse, thus our parser will decompose expression "a / b" into Mul(a, Inv(b)) which contains two operations.

We can perform less-then-zero and equal-to-zero tests on a value. Boolean NOT operation is also implemented as a primitive operation. Based on these three operations and subtraction, all comparison operations can be implemented.

- $a < b$: LessZero(Substract(a, b))
- $a \leq b$: NOT(LessZero(Substract(b , a)))
- $a > b$: LessZero(Substract(b, a))
- $a \geq b$: NOT(LessZero(Substract(a , b)))
- $a == b$: EqualZero(Substract(a, b))
- $a \neq b$: NOT(EqualZero(Substract(a, b)))

3.4 Interpreter

Syntax tree is sufficient for program representation, but not convenient for code optimization. Intermediate code is thus introduced to assist this procedure. In native languages like C/C++ and Java the source code or the syntax tree of a program is translated into intermediate code before being used to generate object or machine code for a target machine[13]. To some extent, JOS language is a high-level programming language which does not output object or machine code themselves, but outputs the intermediate code only. Moreover, JOS intermediate code still contains control flow information, i.e. if-else statement or while loop will remain as an instruction and they will be further translated at runtime. For instance, Intermediate Code 2 is translated from Algorithm 1.

Algorithm 1 Simple Algorithm

```
1: var a := 1
2: var b := 1.2
3: var c := 0
4: d := a * b + b * c
5: if d < 5 then
6:     d := 10
7: else
8:     d := 0
```

Intermediate Code 2 Intermediate Code of Simple Algorithm 1

```
1: a := 1
2: b := 1
3: c := 0
4: $1 = a * b
5: $2 = b * c
6: d := $1 + $2
7: $3 := d − 5
8: $4 = LessZero($3)
9: if $4 then
10:     d := 10
11: else
12:     d := 0
```

From the example, we observe several features of three-address code. First, each three-address assignment instruction has at most one operator on the right side. Second, the order in which instructions are to be executed is fixed, e.g. the multiplication precedes the addition in Algorithm 1. Third, the compiler must generate a temporary variable to hold the value computed by a three-address instruction. Forth, some "three-address

instructions" have fewer than three operands, e.g. Less-than-zero operation has only one operand. Fifth, the control information remains in intermediate code.

3.4.1 Generation

Our simple procedural language includes arithmetic expressions, boolean expressions, conditional branching (if-else statement) and while loop.We work directly on the syntax tree and each category of components will be translated via a translation function. During the translation, temporary variables will be created and inserted into the variable table (VTable) along with other variables declared in the program. This requires the names of temporary variables to be unique.

Translating Expressions

Expressions in JOS Source Language consists of variables and numeric literals, unary and binary operations. Translation function

$$Trans_{Exp} ::= (Exp, VTable, Location) -> [ICode]$$

returns a list of intermediate code instructions(ICode) that upon execution, result of **Exp** will be stored in variable **Location**. Variables and numbers are terminal expressions which requires no further translation. Table 3.1 shows the steps to translate unary and binary expressions in the function call $Trans_{Exp}(Exp, VTable,$ **place**$)$. $Trans_{Exp}$ will be invoked recursively until reaching a terminal expression.

It is worth to notice that the translation function works for both arithmetic operations and boolean operations. $place_1$ and $place_2$ are temporary variables, which are instantiated by calling function $newVar()$. Uniqueness of variable names has been taken care of in $newVar()$.

Expression Type	Translation Algorithm
Unary Operation **unop** Exp_1	$place_1 = newVar()$ $code_1 = Trans_{Exp}(Exp_1, VTable, place_1)$ return $code_1$ | [place := **unop** $place_1$]
Binary Operation Exp_1 **binop** Exp_2	$place_1 = newVar()$ $place_2 = newVar()$ $code_1 = Trans_{Exp}(Exp_1, VTable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, VTable, place_2)$ return $code_1$ | $code_2$ | [place := $place_1$ **unop** $place_2$]

Table 3.1 – Expression Translation

Translating Statements

Translation function for statements is defined as follow:

$$Trans_{Stat} ::= (Stat, VTable) -> [ICode]$$

For each type of statement in the syntax tree, the translation procedures are listed in Table 3.2.

Statement Type	Translation Algorithm
Sequence $Stat_1; Stat_2$	$code_1 = Trans_{Stat}(Stat_1, VTable)$ $code_2 = Trans_{Stat}(Stat_2, VTable)$ return $code_1 \mid code_2$
Assignment $Ident = Exp$	$code = Trans_{Exp}(Exp, VTable, Ident)$ return $code$
If $Cond$ **then** $Stat_1$ **then** $Stat_2$	$code_{if-else} = newICIfElse()$ $palce_1 = newVar()$ $code_{if-else}.codes_{cond} = Trans_{Exp}(Cond, VTable, palce_1)$ $code_{if-else}.cond = palce_1$ $code_{if-else}.codes_{if} = Trans_{Stat}(Stat_1, VTable)$ $code_{if-else}.codes_{else} = Trans_{Stat}(Stat_2, VTable)$ return $code_{if-else}$
While $Cond$ **Do** $Stat_1$	$code_{while} = newICWhile()$ $palce_1 = newVar()$ $code_{while}.codes_{cond} = Trans_{Exp}(Cond, VTable, palce_1)$ $code_{while}.cond = palce_1$ $code_{while}.codes_{body} = Trans_{Stat}(Stat_1, VTable)$ return $code_{while}$
Return $Exp_1, ..., Exp_n$	$palce_1 = newVar()$ $code_1 = Trans_{Exp}(Exp_1, VTable, palce_1)$ ... $palce_n = newVar()$ $code_n = Trans_{Exp}(Exp_n, VTable, palce_n)$ add $palce_1, ..., place_n$ to return variable table return $code_1, ..., code_n$

Table 3.2 – Statement Translation

3.4.2 Optimization

The goal of code optimization is to minimize program execution time. In JOS system, code optimization is an optional pass which takes in and outputs intermediate code. Optimizers should be independent from each other such that it is flexible to add or remove any one. The bottleneck of performance in JOS system is the communication delay between parties. We will introduce an approach to reduce communication rounds in Section 5.1.2.

3.4.3 Encryption

In code encryption phase, all constants in the program will be found and encrypted. After this process, we acquire two similar programs P_{enc} and P_{key} . P_{enc} contains only encrypted constants while in P_{key} constants will be replaced with the corresponding keys. However, if one constant is declared as unencrypted, i.e. attached with tag "**(none)**", it will remain as it is in both P_{enc} and P_{key}. P_{enc} and P_{key} will then be sent to EVH and KH respectively.

3.5 Runtime

Protocol module and scheduler module constitute JOS runtime. The former plays an important role in JOS system while scheduler is responsible for invoking protocols upon execution.

3.5.1 Protocol Module

In this section, we will introduce the common implementation principles among protocols and protocol details will be discussed in Chapter 4.

Figure 3.4 shows the inheritance hierarchy of operation classes. The root class **Operation** contains common data such as operation type, caller and so on. **OperationOnEVH**, **OperationOnKH** and **OperationOnHelper** inherit from **Operation** and override **Run** function. **Run** is the only public function of operations, which simplifies protocol invocation. The only work for **Run** is to call **OnEVH**, **OnKH** and **OnHelper** on each subclass. The actual protocol implementation is accomplished in bottom level classes, e.g. AdditionOnEVH, ANDOnKH. These classes override **OnEVH**, **OnKH**

and **OnHelper** respectively. All operations have to implement two classes inherited form **OperationOnEVH** and **OperationOnKH**. Some operations like AND and Multiplication require assistance from Helper and thus they will have to implement **OperationOnHelper** class as well.

In terms of coding, there are two ways to implement operations for EVH and KH. The first way is to implement protocols in separate classes, like we did in this project. The second approach is to share code between KH and EVH and use branching where protocols differ, i.e. when we call the function, we need to pass party type as a parameter, such that the expected branch will be executed. The first approach allows us to concentrate on the implementation for one party at a time, which makes coding easier, especially when the protocol is complicated. The disadvantage of this approach is code duplication, since protocols for EVH and KH are quite similar. The second approach reduces code quantity but potentially makes coding more difficult.

Execution of operations between parties needs to be synchronized, i.e. when one party requires message from another, it has to wait. The naive solution is to let each operation repeatedly check if the message is in the buffer(also called busy-waiting or spinning). This will not cause severe performance issue when instructions are scheduled sequentially. However, in Section 5.1.2, we introduce a scheduling scheme which allows instructions to be executed in parallel. Each instruction runs one thread or is handled by one task. Potentially, it causes multiple threads to access message buffer concurrently, which results in performance penalty. Moreover, spinning itself is not an efficient approach to solve message receiving problem. In Section 5.2, we will discuss a scheme, by which operations can be woken up asynchronously.

When a party evaluates an instruction, it first instantiates the corresponding operation class and then calls **Run**. This is the first case of operation invocation. The second case is to evaluate codes for If-Else IC or While IC. The third case is to use the operation as a sub-process, e.g. LessZero recursively invokes EqualZero operation. The third case shows the possibility to design and implement new protocols based on existing ones.

The operation protocols can be classified into three categories, namely primitive operation, compound operation and auxiliary operation.

Primitive operations include arithmetic, boolean and comparison operations. All these operations can be invoked by party, by compound operations or be used as a sub-process.

Compound operations are If-Else operation or While operation. They can be invoked by party or compound operations.

Auxiliary operations can only be invoked by other operations. Some operations like AddModToXOR and XORToAddMod are implemented separately for code reuse. The other operations like HammingDistance are separated from the master operation in order to make the protocol comprehensible.

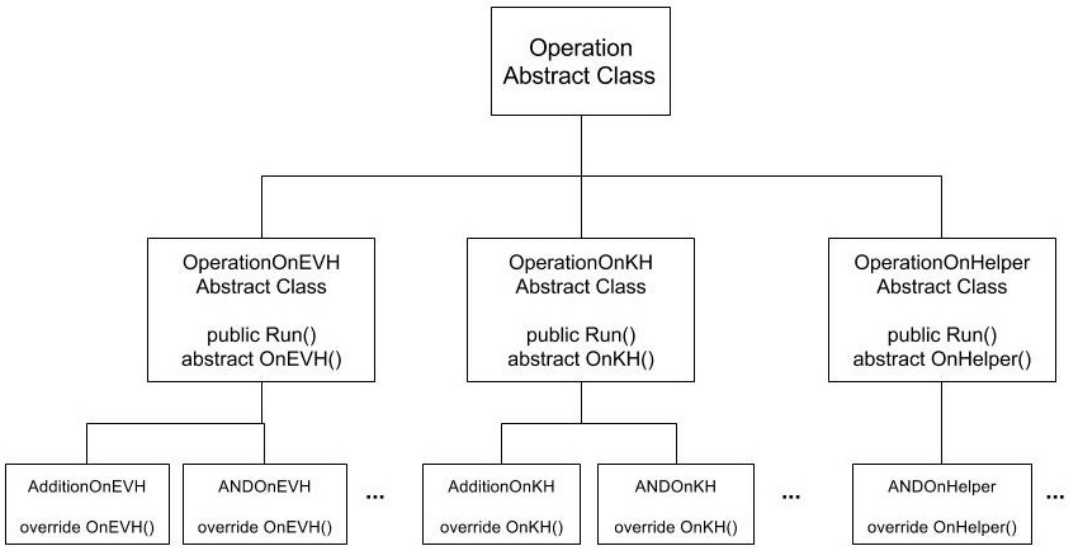

Figure 3.4 – Inheritance Hierarchy of Operation Classes

For primitive operations, encryption type of operands needs to be transformed properly at the beginning of the protocol[2]. Upon termination of the protocol, the result also needs to be processed carefully. In case the operation is invoked by parties or compound operations, result will be inserted into VTable. Otherwise, it is stored in a designated location and later be retrieved by the master operation. If the operation is invoked by a party, the corresponding code will be marked as evaluated. If it is initiated by another operation, caller will be invoked to continue its execution. Through this mechanism, instructions can be executed asynchronously. Asynchronous protocol implementation will be discussed in Section 5.2.

3.5.2 Scheduler

EVH, KH and Helper work in coordination to carry out the execution of intermediate code. Depending on whether the code is optimized according to Section 5.1.2 or not,

[2]Encryption type transformation is only required when the operation is invoked by parties or by compound operations. If the operation serves as a sub-process, the transformation should be taken care of by the master operation.

the order of instruction execution may vary. We need a mechanism to coordinate parties. In intermediate code generation phase, a unique identifier(instruction index) will be assigned to each instruction. Parties can then use index to synchronize execution.

Upon execution, EVH and KH will actively invoke operation protocol based on the next instruction of the program. Protocol execution on Helper however is initiated passively. EVH or KH sends to Helper a message containing explicit invoking message. Helper can only be invoked at most once for each instruction.

3.5.3 Code Generation at Runtime

With sub-domain privacy, the condition of if-else branching may be revealed at runtime and thus code will be translated differently. If $cond_2$ in Intermediate Code 18 can be revealed and the value is **True**, the code will be translated to Intermediate Code 3. Otherwise Intermediate Code 19 will be executed at runtime. Code translation will be discussed in detail in Section 4.2. We can observe that instructions have been significantly reduced by revealing the condition, which shows the potential of sub-domain privacy to improve the performance.

Intermediate Code 3 Translated Nested If-Else Intermediate Code

1: $\$1 = cond1$
2: $\$2 = \$1 * 1$
3: $\$3 = NOT(\$1)$
4: $\$4 = \$3 * d$
5: $d = \$2 + \$4$
6: $\$5 = NOT(cond1)$
7: $\$6 = \$5 * 3$
8: $\$7 = NOT(\$6)$
9: $\$8 = \$7 * d$
10: $d = \$6 + \$8$

3.6 Network

The network module is responsible for laborious message transmission and handling tasks. Ideally, protocols run on four machines(including the client) and messages are transferred through the real network. However, it is not convenient to do testing locally. Our solution is to simulate the network with multiple threads, i.e. one party runs on one thread. Communication between threads can be simulated in two ways. First, though local machined only has one IP address, threads can exploit port number to identify themselves, which allows threads to communicate through real network

protocols. Second, threads exchange messages through shared memory. By this mean, network latency needs to be simulated additionally because accessing memory is undoubtedly faster than real network transmission.

Each party has a sender and a receiver. Besides normal transmission tasks, sender and receiver are also responsible for assembling and disassembling messages.

Instead of sending a message immediately, sender will accumulate several messages. Sender has a buffer for each remote party and it constantly scans all buffers. Messages will be sent only when the number of messages in one buffer exceeds a threshold or timeout occurs. Because the size of one message is usually small and transmission time is negligible compared to the connection overhead, reducing the occurrences of transmission can help to reduce the execution time in total. Communication latency is the performance bottle-neck in JOS system.

Upon receiving a message, receiver disassembles it and inserts the results into a buffer for received messages. In Section 5.2, we will see that dispatcher scans this buffer and awakes the operations.

3.7 Summary

In the chapter we have introduced the structure of the prototype of JOS system. The system is sufficient to evaluate a simple program which satisfies JOS language definition. The purpose of this prototype is to demonstrate a possible system design satisfying JOS scheme and to provide a platform testing. We defined a simple language and implemented a compiler for it. Most of our effort was devoted to JOS protocols implementation and testing.

Chapter 4

Improvements and Extensions of JOS Protocols

Most of JOS protocols have been discussed in [1] and [2], including algorithms and security proofs. In Chapter 3, we have briefly discussed the structure and general implementation patterns of JOS Runtime. In this chapter, our focus will be on improving the existing protocols and introducing new protocols.

4.1 Primitive Operations

In this section, new protocols for OR computation and fixed-point integer multiplication will be presented. Additionally EqualZero protocol is slightly modified and an auxiliary protocol AddModToAdd is introduced. Finally, for the purpose of integrating sub-domain privacy feature into JOS system, we introduce the algorithms for binary operations with one unencrypted operand.

4.1.1 OR Computation

OR protocol is built on top of AND and XOR operations. Based on the equation

$$a \vee b = (a \wedge b) \oplus (a \oplus b) \tag{4.1}$$

OR can be decomposed into an AND and a XOR, which have been implemented in our project as primitive protocols. Algorithm 4 gives the steps of OR protocol on EVH and KH. Since XOR computation is carried out locally, the communication rounds of OR are exactly the same as AND. Both operands of OR should be XOR encrypted and

the encryption type of the result will be XOR encryption.

Algorithm 4 OR (encrypted values $ENC_{k_a}(a), ENC_{k_b}(b)$, keys k_a, k_b)

1: $(ENC_{k_{AND}}(a \wedge b), k_{AND}) = AND((ENC_{k_a}(a), k_a), (ENC_{k_b}(b), k_b))$
2: $EVH : ENC_{k_{OR}}(a \vee b) = ENC_{k_{AND}}(a \wedge b) \oplus ENC_{k_a}(a) \oplus ENC_{k_b}(b)$
3: $KH : k_{OR} = k_{AND} \oplus k_a \oplus k_b$
4: $return \quad (ENC_{k_{OR}}(a \vee b), k_{OR})$

Theorem 1 *The OR protocol in Algorithm 4 is perfectly secure and its computation is correct.*

Proof 1 *Correctness:*

$$
\begin{aligned}
& ENC_{k_{OR}}(a \vee b) \oplus k_{OR} \\
& \quad = (a \wedge b) \oplus k_{AND} \oplus a \oplus k_a \oplus b \oplus k_b \oplus k_{AND} \oplus k_a \oplus k_b \\
& \quad = (a \wedge b) \oplus a \oplus b \\
& \quad = a \vee b (\textit{Using Equation } 4.1)
\end{aligned}
$$

Security: Since the security of AND protocol has been proved in [1] and XOR protocol does not involve message transmission, the operands and results keep confidential for all parties.

4.1.2 EuqalZero Protocol

EqualZero protocol is introduced in [2], which is based on iteratively computing hamming distance between encrypted value and key. Hamming distance gives the different bits between two values, e.g. hamming distance between 101 and 100 is 1. If the secret is 0, encrypted value and key should be equal and the hamming distance is 0.

Hamming distance computation is implemented as an independent protocol. It will not reveal secret to any party by applying double encryption during message transmission. The inputs of this protocol is an encrypted value on EVH and the corresponding key on KH. After execution, EVH stores the encrypted value of hamming distance while KH holds the key.

If the secret is not zero and we iteratively compute hamming distance of the previous output, the underlying secret after each iteration will never be zero. Vice versa, if the secret is zero, the secret will remain zero. We also observe that fewer bits are required

to represent the result of hamming distance computation. For a value with n bits, there are at most n different bit, i.e. hamming distance is at most n and thus we only need $\lceil \log(n+1) \rceil$ bits to encode the result.

The goal of EqualZero is to return a boolean value(i.e. one bit) indicating whether the secret is zero. In EqualZero protocol, hamming distance computation is iteratively invoked and the to-be-compared bits are reduced, i.e. $n = \lceil \log(n+1) \rceil$. Ideally, after constant iterations, n will be equal to 1 and we are able to return a desired result. The algorithm derived from this idea is presented in [2]. However, n can never converge to 1 because $\lceil \log(2+1) \rceil = 2$. Our solution is to apply an OR operation on the last two bits when n reaches 2. In JOS system, we implemented EqualZero protocol according to Algorithm 5.

Algorithm 5 EqualZero (encypted value $ENC_k(a)$, key k, numeric length l_{num})

1: $l = l_{num}$
2: $(ENC_{k_H}(H), k_H) = HammingDistance(ENC_k(a), k, l)$
3: $l = \lceil \log(l+1) \rceil$
4: **while** $l > 2$ **do**
5: $\quad (ENC_{k_H}(H), k_H) = HammingDistance(ENC_k(a), k, l)$
6: $\quad l = \lceil \log(l+1) \rceil$
7: $(ENC_{k_H}(H)^0, k_H^0) = (ENC_{k_H}(H) \wedge \texttt{0x1}, k_H \wedge \texttt{0x1})$(last bit)
8: $(ENC_{k_H}(H)^1, k_H^1) = (ENC_{k_H}(H) \wedge \texttt{0x2}, k_H \wedge \texttt{0x2})$(second to last bit)
9: $(ENC_{k_H}(H), k_H) = OR((ENC_{k_H}(H)^0, k_H^0), (ENC_{k_H}(H)^1, k_H^1))$
10: return $(ENC_{k_H}(H), k_H)$

4.1.3 Conversion among Encryptions

We discuss the encryption conversion from AddMod to Addition in this section. Addition Encryption is required by operations involving scaled values(fixed-point integer), namely Inverse, Sin and Fixed-pint Integer Multiplication. All values in our system are encrypted with either XOR or AddMod encryption scheme by default. Algorithm 6 converts the encryption type of a value from AddMod to Addition[1].

We choose l_{eKey} smaller than l_{key} such that adding a number with l_{num} bits to another with l_{eKey} bits will not cause overflow. Since overflow bits will be truncated, the effect is the same as taking a modulo. After applying Algorithm 6, overflow will not happen, which makes the resulting encryption type is pure addition. Given $l_{num} < l_{eKey}$, l_{eKey} cannot be larger than $l_{Key} - 3$ in order to avoid overflow.

[1] addition and subtraction are defined on the field $F_{2^{l_{key}}}$, i.e. implicitly with modulo

However, there is possibility to reveal the sign of a secret(or the distribution of a secret). For example, given $l_{key} = 8, l_{eKey} = 5, l_{num} = 4$, k_{new} is in $[32, 63]$ and secret value is in $[-8, 7]$. If the encrypted value is in $[64, 70]$, we assert the secret is a positive number. In the extreme case, if it is 70, the secret must be 7. Or when we observe 69 on EVH, the possible values of the secret are 6 and 7. This problem occurs when the key is too close to the range boundary. If we set l_{eKey} large enough, the values in the middle have better chance to be chosen. On the other hand, l_{num} must be significantly smaller than l_{eKey}, such that given the encrypted value, the distribution of secret value is nearly uniform distribution, revealing no information about the csecret.

An alternative protocol is introduced in [2]. However, the conversion is very slow due to carry bit. Algorithm 6 gives a fast conversion procedure that requires using less bits for Addition encryption(l_{eKey}) than for AddMod(l_{key}) which seems abnormal since Addition encryption usually needs more bits than AddMod. But if l_{key} is chosen large enough, Addition encryption with l_{ekey} can still ensure statistical security. Though long key length slows down computations, fast conversion between AddMod and Addition might pay off overall.

To sum up, Algorithm 6 provides an efficient solution for conversion from AddMod encryption to Addition encryption. The algorithm ensures statistical security as long as effective key length(l_{eKey}) is large enough and it is significantly greater than the bit length of secret. In our tests, we set l_{key} to be 128, and l_{eKey} to be 124. The bit length of secret is 64.

Algorithm 6 AddModToAddition (encrypted value $ENC_{k_a}(a)$, key k_a)

1: KH:generate a random unsigned integer k_{new} with length l_{eKey} (k_{new} is uniformly distributed in $[2^{l_{eKey}}, 2^{l_{eKey}+1} - 1]$)
2: KH: $ENC_{-k_{new}}(k_a) = k_a - k_{new}$, send $ENC_{-k_{new}}(k_a)$ to EVH
3: EVH: $ENC_{k_{new}}(a) = ENC_{k_a}(a) - ENC_{-k_{new}}(k_a)$
4: $return \quad (ENC_{k_{new}}(a), k_{new})$

4.1.4 Multiplication

In Section 3.1.1, we have discussed multiplication with fixed-point integer operands(fixed-point integers). Integer multiplication protocol has been introduced in [1]. Algorithm 7 gives the protocol for fixed-point integer multiplication.

In Algorithm 7, we first ignore the scaling bits of both operands and invoke integer multiplication. Then, we need to set scaling bits properly. Scaling bits attribute of **Numeric** class essentially represents the fraction length of a binary number. One way

is simply adding the scaling bits of operands, which preserves precision. However, since bit length to encode a value is fixed, if the length of fractional part is increased, integral part has to be reduced, which shrinks the range of number that can be represented. It will cause overflow in JOS system, because in a program a sequence of fixed-point Integer multiplications may exist and eventually scaling bit length will exceed numeric bit length. Our solution is thus to divide the result with the scaling factor of operands[2], such that the result is scaled in the same way as operands.

If a value is AddMod encrypted, we cannot divide the secret with a constant by simply diving encrypted value and key with the constant. For example, given key length is 4, secret is 4 and key is 14, encrypted value is then $4+14 \mod 2^4 = 2$. We apply division by 2 on the encrypted value and the key, which gives 1 and 7. The decrypted value will be 10 but the expected result is 2. However, if the encryption type is Addition without modulo, this problem is solved. In Algorithm 7, AddMod to addition conversion is invoked before division.

Algorithm 7 Fixed-point Integer Multiplication(encrypted values $ENC_{k_a}(a), ENC_{k_b}(b)$, keys k_a, k_b)

1: $(ENC_{k_{mul}}(a*b), k_{mul}) = Mul((ENC_{k_a}(a), k_a), (ENC_{k_b}(b), k_b))$
2: $(ENC_{k_{new}}(a*b), k_{new}) = AddModToAddition(ENC_{k_{mul}}(a*b), k_{mul})$
3: $EVH: ENC_{k_f}(a*b) = ENC_{k_{new}}(a*b)/ScalingFactor$
4: $KH: k_f = k_{new}/ScalingFactor$
5: set scaling bits of $ENC_{k_f}(a*b)$ and k_f
6: $return \quad (ENC_{k_f}(a*b), k_f)$

4.1.5 Operations with Unencrypted Operands

In order to embed sub-domain privacy feature into JOS system and to allow efficient processing of unencrypted variables, we need protocols for operations with unencrypted operands. Only binary operations with one unencrypted operand will be discussed here. If a value is declared unencrypted, both EVH and KH hold the plain-text.

Addition

Algorithm 8 Public-Private Addition(($ENC_{k_a}(a)$, k_a), b)

1: EVH:$return \quad ENC_{k_a}(a) + b$
2: KH: $return \quad k_a$

[2]Both operands are supposed to have the same scaling factor

Multiplication

We have to handle integer operands and fixed-point integer operands differently. Following two algorithms show the difference.

Algorithm 9 Public-Private Multiplication (($ENC_{k_a}(a)$, k_a), b)

1: EVH:$return \quad ENC_{k_a}(a) * b$
2: KH: $return \quad k_a * b$

Algorithm 10 Public-Private Fixed-point Integer Multiplication (($ENC_{k_a}(a)$, k_a), b)

1: $(ENC_{k_f}(a*b), k_f) = Fixed{-}point\_Intege\_Multiplication((ENC_{k_a}(a), k_a), (b, 0))$
2: $return \quad (ENC_{k_f}(a * b), k_f)$

Subtraction

Algorithm 11 Public-Private Subtraction(($ENC_{k_a}(a)$, k_a), b)

1: EVH:$return \quad ENC_{k_a}(a) - b$
2: KH: $return \quad k_a$ (if minuend b is unencrypted, KH will return $-k_b$)

AND

Algorithm 12 Public-Private AND (($ENC_{k_a}(a)$, k_a), b)

1: EVH:$return \quad ENC_{k_a}(a) \wedge b$
2: KH: $return \quad k_a \wedge b$

OR

Algorithm 13 Public-Private OR (($ENC_{k_a}(a)$, k_a), b)

1: EVH:$return \quad ENC_{k_a}(a) \vee b$
2: KH: $return \quad k_a \wedge \neg b$(if a is unencrypted, return $return \quad \neg a \wedge k_b$)

Theorem 2 *The result of Public-Private OR protocol in Algorithm 13 is correct.*

Proof 2

$$ENC_{k_f}(a \vee b) \oplus k_f$$

$$= (ENC_{k_a}(a) \vee b) \oplus (k_a \wedge \neg b)$$
$$= (ENC_{k_a}(a) \wedge b) \oplus (ENC_{k_a}(a) \oplus b) \oplus (k_a \oplus \neg b)$$
$$= k_a \wedge b \oplus a \wedge b \oplus k_a \oplus a \oplus b \oplus k_a \wedge \neg b$$
$$= a \wedge b \oplus a \oplus b$$
$$= a \vee b (\textit{Using Equation } 4.1)$$

4.2 Compound Operations

Compound operations contain a sequence of primitive operations. At runtime, they will be invoked in the same way as primitive operations, which requires the protocols to comply with the general implementation patterns. In JOS system, there are two types of compound operations: If-Else branching and While loop. Code for compound operations will be generated at runtime.

4.2.1 If-Else Branching

Since the condition of if-else branching cannot be revealed, both branches have to be executed. For example, Intermediate Code 14 is equivalent to the following equations which will be further translated to Intermediate Code 15.

$$d = cond * (a + b) + NOT(cond) * d$$
$$d = NOT(cond) * (a - b) + NOT(NOT(cond)) * d$$

Intermediate Code 14 If-Else Intermediate Code

```
1: if cond then
2:     d = a + b
3: else
4:     d = a − b
```

4.2.2 While Loop

An approach to secure loop with fixed intervals is introduced in [3]. By applying this approach, Intermediate Code 16 will be translated into Intermediate Code 17 in JOS system.

Intermediate Code 15 Translated If-Else Intermediate Code

1: $\$1 = a + b$
2: $\$2 = cond * \$1$
3: $\$3 = NOT(cond)$
4: $\$4 = \$3 * d$
5: $d = \$2 + \$4$
6: $\$5 = NOT(cond)$
7: $\$6 = a - b$
8: $\$7 = \$5 * \$6$
9: $\$8 = NOT(\$5)$
10: $\$9 = \$8 * d$
11: $d = \$7 + \$9$

Intermediate Code 16 While Intermediate Code

1: **while** $d < 5$ **do**
2: $\quad d = d + 1$

4.2.3 Nested Compound Operations

When translating a nested compound operation, we need to consider the condition of outer scope. The translation process can be decomposed into two steps. First, we generate the condition combination that leads the execution to a certain branch, e.g. in Intermediate Code 18, to reach the **Else** branch of the inner **If-Else**, $cond_1 \wedge \neg cond_2$ must be true. Second, if there is a further nested compound operation, the condition combination(its result) will be passed as parameter to the translation function. Intermediate Code 18 is equivalent to the following codes which will be translated into Intermediate Code 19.

$$tempCond1 = cond1 \wedge cond2$$
$$d = tempCond1 * 1 + NOT(tempCond1) * d$$
$$tempCond2 = cond1 \wedge NOT(cond_2)$$
$$d = tempCond2 * 2 + NOT(tempCond2) * d$$
$$tempCond3 = NOT(cond1)$$
$$d = tempCond3 * 3 + NOT(tempCond3) * d$$

Intermediate Code 17 Translated While Intermediate Code

```
1: (nextReveal = c_0, iter = 0)
2: repeat
3:     repeat
4:         $1 = d − 5
5:         $2 = LessZero($1) ($2 is the condition, which will be stored in While IC)
6:         $3 = d + 1
7:         $4 = $2 ∗ $3
8:         $5 = NOT($2)
9:         $6 = $5 ∗ d
10:        d = $4 + $6
11:    until iter == nextReveal
12:    (nextReveal = nextReveal ∗ c_1)
13:    reveal $2
14: until $2 == false
```

Intermediate Code 18 Nested If-Else Intermediate Code

```
1: if cond1 then
2:     if cond2 then
3:         d := 1
4:     else
5:         d := 2
6: else
7:     d := 3
```

Intermediate Code 19 Translated Nested If-Else Intermediate Code

```
1: $1 = cond1 ∧ cond2
2: $2 = $1 ∗ 1
3: $3 = NOT($1)
4: $4 = $3 ∗ d
5: d = $2 + $4
6: $5 = NOT(cond2)
7: $6 = cond1 ∧ $5
8: $7 = $6 ∗ 2
9: $8 = NOT($6)
10: $9 = $8 ∗ d
11: d = $7 + $9
12: $10 = NOT(cond1)
13: $11 = $10 ∗ 3
14: $12 = NOT($10)
15: $13 = $12 ∗ d
16: d = $11 + $13
```

Chapter 5

Performance Improvement

5.1 Parallelization

This section will introduce two parallelization strategies, namely algorithm level parallelization and code parallelization. Parallelization can help to reduce communication rounds significantly.

5.1.1 Within Protocols

All protocols in JOS system are able to handle multiple inputs in one execution, i.e. no matter how many inputs there will be, the communication round is always equal to that of evaluating one input. Thus, it is possible to optimize the algorithm during protocol design phase by aggregating successive but independence sub-operations into one. Protocol designers should try to optimize sub-operations. For example, in LessZero protocol, we need to recursively invoke LessZero operation on right and left half of the value. Since both halves are independent, we can pass them together to LessZero. At the last level of the recursion, there will be l_{key} inputs.

5.1.2 Running Protocols in Parallel

In addition to sub-operations in each individual protocol, instructions in a program also have the potential to be evaluated in parallel. For example, there is no dependence among all commands in Intermediate Code 20, which allows parallel execution.

However, when users writes programs with JOS language, they will focus on realizing functionalities instead of optimization. Instruction schedulers(EVH and KH) should

Intermediate Code 20 Independent Intermediate Code

1: $a_1 = b_1 * c_1$
2: $a_2 = b_2 + c_2$
3: $a_3 = b_3 - c_3$

then take on the task. Schedulers first detect the dependence graph of a program and then execute instructions accordingly.

Dependence Graph

Dependence graph analysis will be carried out on intermediate codes instead of syntax tree, which simplifies the process as intermediate code only contains result variable and at most two operands.

Between two **Assignment** intermediate codes C_i, C_j, there exists three types of dependence [20]. We assume C_i precedes C_j in a program but they are not necessarily successive.

Output dependence output variables of C_i and C_j are the same.
anti dependence output variable of C_j is one of the operands of C_i.
flow dependence output variable of C_i is one of the operands of C_j and there is no instruction C_k between C_i and C_j such that the output of C_k is the same as C_i.

To determine the dependence between an **If-Else** intermediate code $C_{if-else}$ and an **Assignment** C_{assign}, we compare C_{assign} with each code in condition, if-branch and else-branch codes lists of $C_{if-else}$. If dependence exists between C_{assign} and any $C_{if-else}$'s code, C_{assign} is dependent on $C_{if-else}$. Dependence between **While** intermediate code C_{while} and C_{assign} is determined similarly. In order to determine the dependence between compound intermediate code which are $C_{if-else}$ and C_{while}, all codes in their code lists need to be compared pairwise. Figure 5.1 shows the dependence graph of Intermediate Code 21.

Instruction Scheduling

Schedulers check all instructions in rotation and create one new task to handle one instruction if it does not depend on previous codes or dependent codes have been evaluated. Since tasks are carried out concurrently, instructions are evaluated in parallel. Though message rounds are not strictly synchronized, it is likely that several

Intermediate Code 21 Sample Code Optimization

Statement 1: $a_1 = b_1 * c_1$
Statement 2: $a_2 = b_2 + c_2$
Statement 3: $a_3 = a_1 - a_2$
Statement 4: $a_4 = a_1 * a_2$
Statement 5: $a_1 = a_3 - a_4$
Statement 6: $cond = LessZero(a_1)$
Statement 7: $if(cond)$
$x = 1$
$else$
$x = 10$

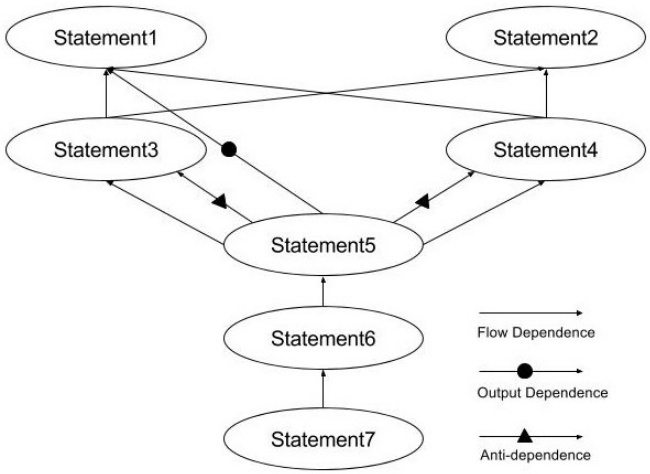

Figure 5.1 – Dependence Graph of Intermediate Code 21

messages will be aggregated and sent together. For instance, Intermediate Code 21 will be executed in order (Statement1, Statement2),(Statement3, Statement4), Statement5, Statement6, Statement7. (Statement1, Statement2) means Statement1 and Statement2 will be evaluated simultaneously and so are Statement3 and Statement4.

5.2 Asynchronous Protocol

We use different tasks to handle instructions concurrently. In the first version of prototype, tasks are blocked if they wait for messages from a another party, i.e. tasks spin on the message queue to check if messages have been received. Busy waiting will cause competition among tasks which is not only computationally expensive but also limits the performance gain for code parallelization.

Our solution is to delegate message receiving process to a dispatcher, such that there

will be only one process reading from message queue. When one instruction requires a message from another party, it sends dispatcher a request which contains the context of current instruction, the party it wishes to receive message from and the location to store the message. The task itself will be terminated.

Dispatcher maintains a queue of requests and constantly examines each request. It dequeues a entry from the request queue. If the corresponding message is found in the message queue, dispatcher will store the result in a specified location and invoke the instruction to continue the execution, which is essentially a callback mechanism.

Callback is also required when one instruction invokes sub-operations. When instruction calls another operation, current task will be terminated and the control is handed over to the task of sub-operation. After the execution of sub-operation, the invoker will be woken up.

To implement callback mechanism in our system, protocols are decomposed into multiple steps. Each step contains one and only one of the operations: invoke another operation; receive a message; invoke itself. Invoking instruction itself is necessary when no special operation is required under certain condition, e.g. at the beginning of each protocol, encryption type of operands need to converted properly, but if the operands have already been encrypted correctly, there is no need to invoke conversion operation.

Figure 5.2 demonstrates sub-operation invocation and callback mechanism in EualZero Protocol on EVH. The figure also shows the control transfer between operation and dispatcher, operation and sub-operation.

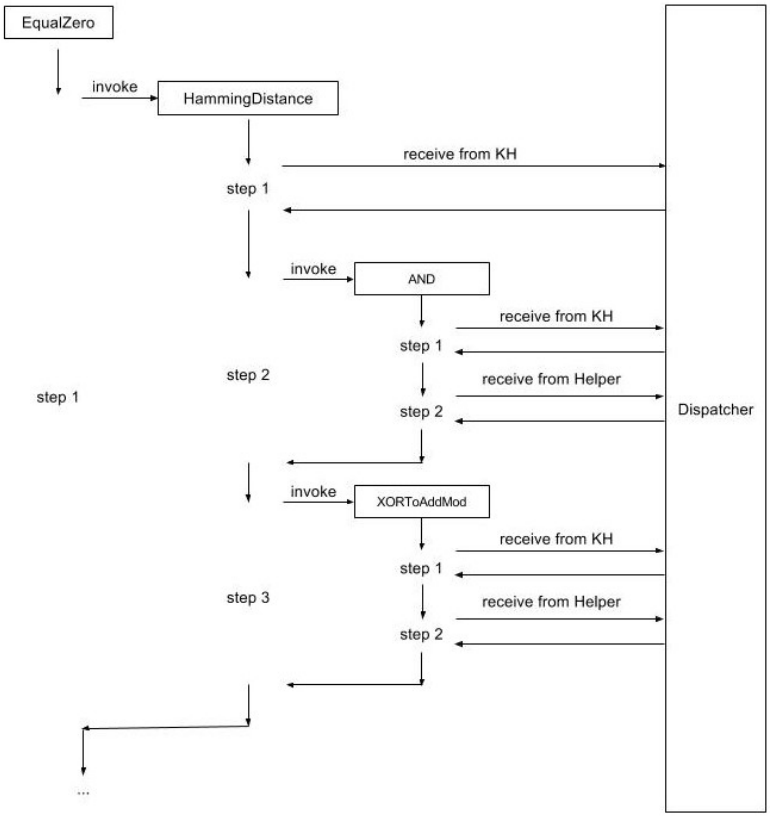

Figure 5.2 – Invocation and Callback of EualZero Protocol(Partial)

5.3 Sub-domain Privacy

The paper [3] introduces the concept of sub-domain privacy and its potential to optimize performance by fine tuning security definitions and, potentially, trading security for performance. Statistical sub-domain privacy is a variant of sub-domain privacy. Sub-domain privacy and statistical sub-domain privacy are defined as follow[3]:

Definition 3 *(Sub-domain Privacy (with respect to C)). A variable v with $val(v) \in D$ is sub-domain private with respect to a set of values $C \subseteq D$ if given that $val(v) \in C$ an attacker cannot do better than guessing $val(v)$ with probability $1/|C|$. We write $DOM(v,C)$.*

Definition 4 *(Statistical Sub-domain Privacy (with respect to membership in C)). A variable v with $val(v) \in D$ is statistically sub-domain private with respect to membership in C if v is sub-domain private with respect to C and one can infer whether $val(v) \in C$ with probability at most p. We write STATDOM(v,C, p).*

We have implemented DOM/STATDOM for variable and if-else branching in the project.

5.3.1 Random Number Generation

STATDOM will only reveal normal values with certain probability p. We thus need to generate a random number and compare it with p at runtime. Algorithm 22 provides a solution to keep the random number encrypted. The protocol is robust against a malicious adversary.

Algorithm 22 Random Number Generation()

1: EVH: generate a random number $r_1 \in [0,1]$ and a key k_1, compute $ENC_{k_1}(r_1) = r_1 \oplus k_1$ and send k_1 to KH
2: KH: generate a random number $r_2 \in [0,1]$ and a key k_2, compute $ENC_{k_2}(r_2) = r_2 \oplus k_2$ and send $ENC_{k_2}(r_2)$ to EVH
3: EVH: $ENC_{k_f}(r_1 \oplus r_2) = ENC_{k_1}(r_1) \oplus ENC_{k_2}(r_2)$
4: KH: $k_f = k_1 \oplus k_2$
5: $(ENC_k(r), k) = XORToAddMod(ENC_{k_f}(r_1 \oplus r_2), k_f)$
6: return $(ENC_k(r), k)$

Theorem 3 *Algorithm 22 is correct and secure.*

Proof 3 *Correctness: Given* r_1 *and* r_2 *are uniformly distributed in* $[0,1]$ *and they are fixed-point integers,* $r = r1 \oplus r2$ *is uniformly distributed in* $[0,1]$. *Security: assume with loss of generality that the EVH is corrupt. We have that* $r = r1 \oplus r2$. *The EVH has* $ENC_{k_1}(r_1)$, $ENC_{k_2}(r_2)$ *and* k_1. *To be able to gain information about the random number r he needs* k_2. *However,* k_2 *is retained by the KH. Thus, he cannot do better than guess any bit in r and thus cannot change r in a meaningful way.*

5.3.2 DOM/STATDOM for Variables

DOM/STATDOM is part of type information of a variable, which should be clarified when the variable is declared, e.g.

$var \quad a = 0[100,200];$
$var \quad b[1,100,0.7];$

In JOS language, DOM/STATDOM attribute comes after variable declaration and initialization. The first example declares variable a with DOM attribute. $[100,200]$

means if $a \in [100,200]$, it must be kept confidential. The second example declares variable b with STATDOM attribute, where $[1,100,0.7]$ means if $b \notin [1,100]$, the value will be revealed with probability 0.7.

The DOM/STATDOM attribute will be referred to when the available is assigned a value. At the end of each operation, an auxiliary operation called **Conceal** will be invoked to deal with assignment. If the available does not have DOM/STATDOM attribute, the value will be assigned directly regardless whether it is encrypted or not. Otherwise, we will have to conceal or reveal the value according to DOM/STATDOM attribute. For DOM/STATDOM, if the value is not encrypted and abnormal, it will be concealed upon assignment. In case the value is encrypted but normal, we will reveal it for DOM or with certain probability for STATDOM.

5.3.3 DOM/STATDOM for If-Else Branching

Securing the condition of if-else branching requires executing both branches. Using DOM/STATDOM, confidentiality can be preserved by revealing the evaluated condition in some cases, enabling the system to only evaluate one of the two branches. As shown in previous chapter, condition revealing helps to reduce the amount of codes at runtime. The performance gain is more significant in case one branch is computationally expensive, e.g.

$var \quad iter = 1, a = 0, b = 300;$
$if(iter > 0)[true, 0.8]$
    $a = a + b;$
$else$
    $while(iter < 10000)$
    {
        $iter = iter + 1;$
        $a = a + b;$
    }

"[true, 0.8]" means if condition is *true*, it will be revealed with probability 0.8. Once the condition is revealed, only *if* branch will be executed. In this case, the running time should be reduced significantly as *else* branch contains a while loop which repeats the same operation at least 10000 times.

Chapter 6

Evaluation

We will present evaluation results of JOS system in this chapter. Section 6.1 shows the setup of tests. Results of individual protocol test will be given in Section 6.2. Program execution and performance gain by applying sub-domain privacy will be discussed in Section 6.3.

6.1 Setup

We run tests on a desktop PC equipped with 32 GB RAM and two AMD 2000Mhz, 8 cores processors(in total 16 cores). Parties are simulated by different threads and communication between parties is simulated by message transmission between threads. Sender(running on a separate thread) will sleep for 100 milliseconds before sending a message, simulating the network delay that occurs in real networks. Network delay in real networks can vary dramatically, but is typically about 30 ms in developed countries to reach other reasonably well-connected servers. All tests are run once but each of them contains enough repetitions of the same operation or program. In individual protocol test, each program contains 100 instructions of the same operation. In tests for sub-domain privacy, the program is executed 10000 times per test.

6.2 Individual Protocol Test

We test the accuracy and running time of each protocol by executing a program consisting of successive and identical operations. Figure 6.1 presents the average execution time of each operation. The first three columns give the time for concurrent execution. The remaining three columns show the time for sequential execution.

	Average Execution Time(Optimized) [ms]			Average Execution Time(Not Optimized) [ms]		
	Encrypted	Not Encrypted	Partialy Encryped	Encrypted	Not Encrypted	Partialy Encryped
Addition	255	235	225	1713	1698	1692
Substraction	146	131	148	1690	1692	1692
Multiplication	530	133	535	39267	1691	39492
Sin	882	134		64561	1692	
XOR	149	132	132	1693	1690	1695
AND	405	134	132	26791	1710	1691
OR	389	148	131	26821	1691	1714
NOT	145	138		1692	1700	
EqualZero	15351	148		111903	1692	
LessZero	24969	136		215478	1670	

Figure 6.1 – Results of individual protocol test, $l_{key} = 128, l_{num} = 80, l_{fraction} = 21$

Two conclusions can be summarized from Table 6.1.

- For the operations requiring communication between parties(Multiplication, AND, OR, Sin, EqualZero, LessZero), if one operand or both operands are not encrypted, the average execution time is expected to be shorter than that of fully secured protocol. The only exception is multiplication with fixed-pointer integer. Even if one operand is not encrypted, the execution time is equal to that of fully secured protocol. The protocol can be optimized in the future. For the operations with no communication, the execution time is not improved by revealing any operand. We could also make improvement on that.
- Executing code in parallel helps to improve performance. Instructions in the program are independent from each other, which allows them to be evaluated concurrently. Protocols with heavy message transmission benefit form code parallelization the most, e.g. performance is improved by a factor of 70 for Multiplication, AND, OR and Sin.

Execution time can also be further reduced by re-implementing the whole system with C++ or optimizing message parsing and encoding.

6.3 Sub-domain Privacy

The test cases and results in this section are also presented in [3].

6.3.1 Control Algorithm

Algorithm 23 contains a simple branch for normal data ($v \leq 50$) and a more complex branch for abnormal data ($v \in [50, 1000]$).

Algorithm 23 Control Algorithm for Benchmarking

```
1: if v > 50 then
2:     {Abnormal data, handle with x, v confidential}
3:     if v > 700 then
4:         x = a_8 * Sin(v)
5:         ...
6:     else if v > 100 then
7:         x = a_2 * Sin(v)
8:     else
9:         x = a_1 * Sin(v)
10: else
11:     x = a_0 * Sin(v)
```

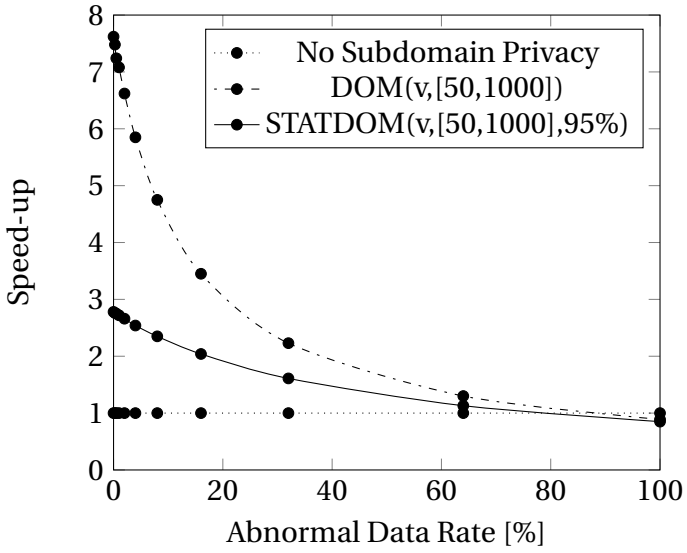

Figure 6.2 – Speed-up for Control Algorithm[3]

Figure 6.2 shows the speedup for 10000 executions depending on the probability that a value v is normal for three scenarios: i) without sub-domain privacy, i.e. treating all values confidentially; ii) with domain privacy DOM(v,[50,1000]) and iii) statistical sub-domain privacy STATDOM(v,[50,1000],95%). DOM requires two LessZero comparisons of the value v with the lower and upper bound. If all data is normal, using DOM yields a factor of about 800% speedup. STATDOM behaves worse, because normal data will be processed confidentially with certain probability. Besides, in addition to the two comparisons, it requires a generation of a (secret) random number and comparison of the number with the percentage given in the definition of statistical subdomain privacy.

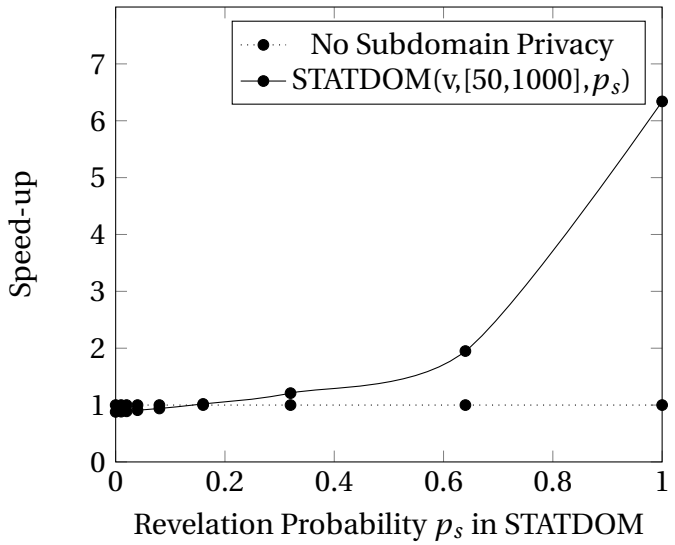

Figure 6.3 – Speed-up for Control Algorithm for varying parameter for statistical sub-domain privacy [3]

Figure 6.3 shows the speedup when using statistical sub-domain privacy, i.e. STATDOM(v,[50,1000], p_s), for varying revelation probability p_s. The benchmark assumes 1% of values are abnormal. If we do not reveal any values, statistical sub-domain privacy increases run-time by about 10% . Good speedups are achieved for large values of p_s exceeding a factor of 6.

6.3.2 Pattern Matching

We use synthetic clinical data of patients having some disease and healthy persons. We look for patterns, e.g. we want to determine whether specific patterns like genes in the DNA correlate with a disease. We model the data by a binary sequence of 512 bits and try to find 10 bit sequences of 32 bits each. For healthy patients we are allowed to decrypt the data and compute on plain-texts. For a specific pattern our algorithm simply checks from each position i whether the next 32 bits match the pattern. We use the sign bit to denote whether a person is patient or not, i.e. data of patients has sign bit 0, yielding DOM(data,$[0, 2^{511} - 1]$)

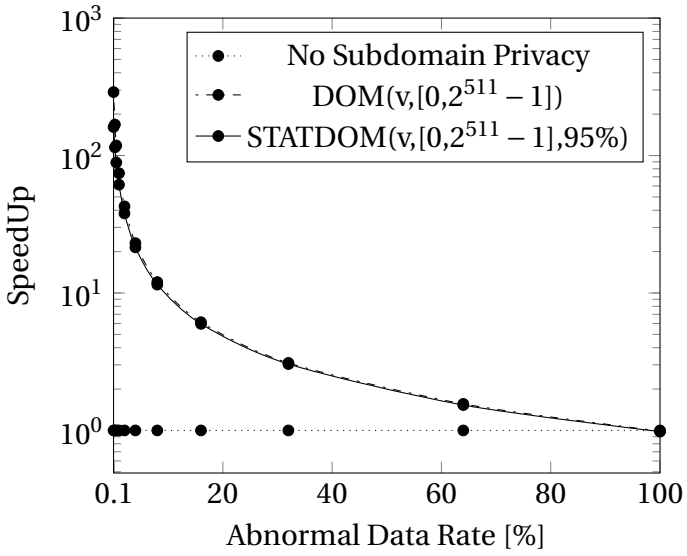

Figure 6.4 – Speed-up for Pattern Matching [3]

Figure 6.4 shows the speedup for 10000 persons depending on the probability that a person is healthy. The maximum possible improvement is a factor of 100 for one per cent of sick people.

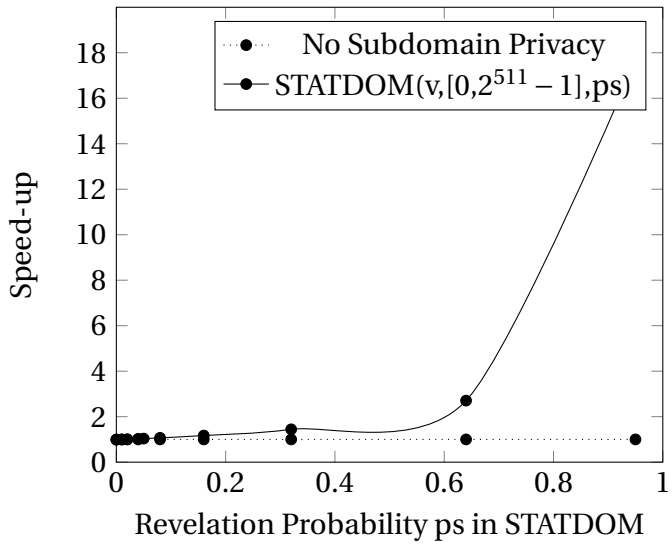

Figure 6.5 – Speed-up for Pattern Matching for Statistical Privacy[3]

In Figure 6.5 we assumed that 95% of persons are healthy. The maximum possible gain is a factor of 20 for a revelation probability of 100%, which we closely achieve.

Chapter 7

Future Work

We only defined a simple language containing basic operations and statements. Plenty of basic features of a practicable programming language like Function, Array, Class are missing in JOS language. In order to add these features, new protocols might be introduced. For example, we need to create a new protocol for array access if the index is encrypted.

Dependence graph of intermediate codes is generated by optimizer to assist concurrent code execution at runtime. However intermediate code optimization is a well-studied field and various approaches have been proposed, e.g. remove redundant codes, optimize a while loop. We can implement new optimizers and integration into JOS system is trivial.

Some protocols need improvement. i) **Sin** only supports 64 key bit length(or less) because acceptable inputs of **Sin** in C# range from approximately -9223372036854775295 to approximately 9223372036854775295. When carrying out **Sin** operation on encrypted values or keys, the value can easily exceed the range if key length is too large. ii) LessZero is much slower than other operations. iii)Result of Inverse is not 100% accurate and the operand of Inverse can only be positive integer. iv) protocol for public-private fixed-point integer multiplication could be redesigned such that it does not require communication.

Executing time of operations with unencrypted operands are not negligible because of the overhead of instruction scheduling. For operations that do not need communication, i.e. Addition, Subtraction, XOR and NOT, run-time is not even reduced by revealing one or both operands. If instructions with unencrypted operands can be converted into native code, the performance can be further improved.

Performance of JOS system can also be improved in other aspects, e.g. i) re-implement the whole system with C++. ii) optimize message parsing and encoding.

Currently all parties run the same machine and network is simulated. We need to deploy the code on servers and test JOS system on real network.

Bibliography

[1] Johannes Schneider. *A General Multi-Party Protocol for Minimal Communication and Local Computation.* arXiv preprint arXiv:1508.07690 (2015).

[2] Johannes Schneider. *Secure Numerical and Logical Multi Party Operations.* arXiv preprint arXiv:1511.03829 (2015).

[3] Johannes Schneider, Bin Lu, TL, SO and MH. *Subdomain and Access Pattern Privacy: Trading off Confidentiality and Performance.* ABB Research,Baden-Daettwil,Switzerland. In Submission.

[4] Craig Gentry. *Fully Homomorphic Encryption Using Ideal Lattices.* In STOC, vol. 9, pp. 169-178. 2009.

[5] Craig Gentry and Shai Halevi. *Implementing Gentry's fully-homomorphic encryption scheme.* In Advances in Cryptology–EUROCRYPT 2011, pp. 129-148. Springer Berlin Heidelberg, 2011.

[6] M. Ben-Or, S. Goldwasser and A. Wigderson. *Completeness theorems for non-cryptographic fault-tolerant distributed computation.* In Proceedings of the twentieth annual ACM symposium on Theory of computing, pages 1-10, 1988.

[7] J. Bar-Ilan and D. Beaver. *Non-cryptographic fault-tolerant computing in constant number of rounds of interaction.* In Proceedings of the eighth annual ACM Symposium on Principles of distributed computing, 1989.

[8] D. Bogdanov, S. Laur and J. Willemson. *Sharemind: A framework for fast privacy-preserving computations.*In Computer Security-ESORICS 2008, pages 192-206.Springer, 2008.

[9] Ran Canetti. *Security and composition of multiparty cryptographic protocols.* Journal of CRYPTOLOGY 13, no. 1 (2000): 143-202.

[10] Andrew Chi-Chih Yao. *How to generate and exchange secrets*. In Foundations of Computer Science, 1986., 27th Annual Symposium on, pp. 162-167. IEEE, 1986.

[11] O. Goldreich. *Towards a theory of software protection and simulation by oblivious rams*. In Proceedings of the nineteenth annual ACM symposium on Theory of computing, pages 182-194. ACM, 1987.

[12] Alfred V. Abo, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools(Second Edition)*. Published on September 10, 2006.

[13] Michael L. Scott. *Programming Language Pragmatics*. Published on January 1, 2000.

[14] D. Boneh, E. Goh and K. Nissim. *Evaluating 2-DNF Formulas on Ciphertexts*. In Theory of cryptography, pp. 325-341. Springer Berlin Heidelberg, 2005.

[15] Peeter Laud and Jaak Randmets. *A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols*. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1492-1503. ACM, 2015.

[16] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider and Immo Wehrenberg *TASTY: tool for automating secure two-party computations*. In Proceedings of the 17th ACM conference on Computer and communications security, pp. 451-462. ACM, 2010.

[17] Yihua Zhang, Aaron Steele and Marina Blanton *PICCO: A General-Purpose Compiler for Private Distributed Computation*. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 813-826. ACM, 2013.

[18] Dahlia Malkhi, Noam Nisan, Benny Pinkas and Yaron Sella. *Fairplay – A Secure Two-Party Computation System*. In USENIX Security Symposium, vol. 4. 2004.

[19] Daniel Demmler, Thomas Schneider and Michael Zohner. *ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation*. In NDSS. 2015.

[20] D.J.Kuck, R.H.Kuhn, D.A.Padua, B.Leasure and M.Wolfe *Dependence Graphs and Compiler Optimization*. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 207-218. ACM, 1981.