Programme
Seventh Framework Programme
Strategic Objective
Information Security in Supply Chain Management
Integrated Project / Programme Title
Secure Supply Chain Management
ACRONYM
secureSCM
Project No.
FP7-213531
WP No. / Work Package Title
WP9 Cryptographic Aspects
Deliverable No. / Deliverable Title
D9.2 Security Analysis

Leading Partner: IU

Security Classification: CONFIDENTIAL

Delivery: July 2009

Version: 1.0

Versioning and contribution history

Version	Description	Responsibility	Date	Comments
0.1	New	IU	01 Feb. 2009	Deliverable document start
0.3	Peer review	UNIMI, TUE	13 Jul. 2009	Quality assurance by internal peers
0.6	Final editing	IU	27 Jul. 2009	Revision based on peer review comments, proof-reading, formatting
0.7	Final approval	PCC / Scientific Coordinator delegate.	27 Jul. 2009	Formal approval within the project
1.0	Submission to EC	PMO		Collection, formatting, printing, and distribution of deliverables

Legal Disclaimer

Table of Contents

List of Abbreviations

LP	Linear Programming
LSB	Least Significant Bit
PRF	Pseudo-Random Function
RSS	Replicated Secret Sharing
PRSS	Pseudo-random Replicated Secret sharing
RISS	Replicated Integer Secret Sharing
SMC	Secure Multiparty Computation
ST	Small-Tableau
LT	Large-Tableau
IP	Integer Pivoting
RP	Rational Pivoting

List of Symbols

$\mathbb{F}$	An arbitrary finite field.
$\mathbb{Z}$	Set of integers $[-\infty..\infty]$.
$\mathbb{Z}_m$	Set of integers $[0..m-1]$.
$\mathbb{Z}_q$	Reserved for a prime field (q is prime).
$[r]^{\mathbb{F}}$	A Shamir sharing in $\mathbb{F}$ of $r \in \mathbb{F}$.
$[r]^{RZ}$	A RISS sharing of r.
$[r]^{\mathbb{Z}_q}$	Shamir sharing of $r \in \mathbb{Z}_q$ for prime $q > 2$.
$[r_0]^{\mathbb{F}_{2^m}}$	Shamir sharing of LSB of r represented in $\mathbb{F}_{2^m}$.
$[r]$	The same as $[r]^{\mathbb{Z}_q}$ when q is implicit.
$[r]_B$	Shamir sharing in $\mathbb{Z}_q$ of bits of r in 2s complement.
$a \rightarrow i$	a is sent to i over a private authentic channel.
$(a_1, \ldots, a_n) \rightarrow (i_1, \ldots, i_n)$	a_j is sent to i_j over a private authentic channel.
$a \Rightarrow i$	a is sent to i over a public broadcast channel.
$a \Rightarrow$	a is broadcast.
$a \leftarrow b$	a is assigned the value b.
$a \overset{R}{\leftarrow} X$	a is chosen uniformly from set X.
$a \in_R X$	a is distributed uniformly over set X.

List of Protocols

Executive Summary

Collaborative supply chain management and optimization requires participants to share necessary but sensitive data. Often the risks associated with revealing this data far exceed the benefits gained. Consequently some parties may be unwilling or unable to share certain inputs with the other parties. The aim of the SecureSCM project is to develop cryptographic solutions using Secure Multiparty Computation (SMC) to address this problem of data sharing in supply chain optimization. The goal of SMC is to perform some computation in a secure manner - i.e., without revealing inputs and outputs to unauthorized parties.

Work-Package 9 (WP9) of the project deals with cryptographic aspects of secure computation and recent work in this work-package focused on the analysis of the protocols developed in the first phase of the project. The secure Simplex protocols described in Deliverable 3.1 (D3.1) are based on standard cryptographic techniques from the literature and provide adequate security in the semi-honest model. On the other hand, they rely on general building blocks (discussed in Deliverable 9.1 (D9.1)) and do not take into account any special features/requirements of our application scenario. Consequently, some of the building blocks are serious bottlenecks. After reviewing D3.1, we concluded that WP9 has to address a combination of several aspects: (1) security; (2) complexity and performance; (3) functional aspects (such as accuracy); and (4) tradeoffs between the above. This deliverable (D9.2) summarizes the following work done in WP9:

1. Development of more efficient methods and building blocks, adapted to our applications (e.g., for data encoding, shared random values, secure arithmetic and comparison for fixed-point numbers).

2. Theoretical foundation for our protocols with statistical security.

3. Formulation/selection of precise complexity metrics. The metrics abstract away factors that depend on configuration/implementation/execution environment and offer meaningful information suitable for studying tradeoffs and comparing design solutions.

4. Complete and consistent specification of all the building blocks along with an appliation example (secure ST-RP Simplex) with a rigorous complexity/security analysis.

5. Tests and performance measurements with prototypes of the main building blocks and application protocols.

6. Analysis of tradeoffs between security, efficiency, and functional aspects (perfect versus statistical privacy, efficient encoding of different data types versus the overhead of conversions, pre-computation).

7. Identification of several building blocks that remain bottlenecks and need more efficient solutions taking in consideration of our application context.

In conclusion, the deliverable established a foundation for the protocols developed in the project, by providing complete specifications and analysis for all the building blocks and for the current application protocols

1 Introduction

The SecureSCM project aims to develop cryptographic solutions to the problem of data sharing in Supply Chain Optimization (SCO). The SCO problem has a precise mathematical structure. It is an instance of the general Linear Programming (LP) problem. However, standard algorithms for LP problems are not suitable for this purpose because they require participants to reveal private data needed as input to the algorithm. The risk of revealing this information far exceeds the benefits gained. Therefore, the aim of the project is to develop efficient techniques for securely solving LP problems. We refer the reader to Deliverable[26] for details of the SecureSCM problem and to Deliverable 3.1[27] and [23] for details of the Simplex algorithm to solve an LP problem. In our context, we consider the following variations of the Simplex algorithm: Small-Tableau (ST), Large-Tableau (LT), Integer-Pivoting (IP), Rational-Pivoting (RP), and Revised-Tableau (RT), along with their combinations. An overview of the algorithms not provided in D3.1 is given in Chapter 6.

In this deliverable, we analyze the different components for constructing secure protocols implementing the above algorithms. In particular we focus on efficiency, security and accuracy of the underlying components and discuss different trade-offs.

1.1 Privacy-Preserving Computation

Consider the dining cryptographers' problem [8]: three cryptographers are sitting down to dinner at their favorite restaurant when their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they would like to be sure of which side is paying.

They decide to solve their problem as follows[1]: each cryptographer mentally decides a secret bit - 0 if (s)he paid and 1 if (s)he did not. Then they follow a protocol to compute the AND of these bits without revealing their secret bits and reveal the result (from which they can solve their problem).

The above strategy is 'secure' assuming they use a protocol that computes the (public) AND of secret bits and at the same time reveals absolutely no information about the secret bits. Such a protocol is said to be *privacy-preserving* or *secure*.

Secure multi-party computation (SMC) is a distributed computation using a protocol constructed from such privacy-preserving building blocks. A classical result in cryptography [7, 5, 6] states that the resulting protocol is perfectly privacy-preserving in the sense of universal composability (UC) if every building block is perfectly privacy-preserving in the sense of UC. Another result [32, 31] states that there exists a privacy-preserving protocol for every computation that we can imagine. Although the technique of [32, 31] works for any computation, it is extremely inefficient in practice since it operates at the circuit level and uses bits as the basic data type. Recently, however, efficient privacy-preserving protocols for several operations on signed integers have been proposed [12, 18, 4]. These are based on linear secret sharing and do not operate at the bit level. Using these building blocks, it is possible to design complex protocols accomplishing several interesting tasks that were not practically possible using earlier methods.

[1] Note that the solution proposed in the original paper was different.

1.2 Problem Definition

Our secure computation problem involves n parties with $n \geq 3$. A LP problem in *standard* or *canonical* form (depending on the algorithm used) is given. The different values defining the constraints and the objective function are the data to be protected. The way in which the different parties supply the data is called *partitioning*. Although this partitioning will depend on the actual real-world context, in this deliverable we assume a random partitioning of data, which is the most general case. A mathematical description of an LP problem is given in Section 6.2.2.

At this stage we will not discuss how to encode a given LP problem instance in our secure processor. Rather we will assume that at the beginning of the protocol, the parties have obtained a secret-sharing of some given problem instance. Details of how to input the problem and read the final solution will be discussed in Deliverable D3.2.

The goal of the project is to design efficient protocols for solving LP problems of a certain form. The most common method for solving LP problems is the Simplex algorithm (see Section 6.1). Our secure protocols will be based on this algorithm.

1.3 Goals of WP9

Work Package Goals: Although the theory of SMC is quite simple, designing a practical secure protocol is not that easy. A primary issue is to ensure that the resulting protocol indeed achieves the desired security. When several building blocks are composed (e.g., with statistical and perfect privacy, or public and secret outputs) the resulting protocol becomes quite complicated to analyze. The challenge is to build protocols that offer sufficient security guarantees and at the same time meet the (minimal) functional and performance requirements of the applications. Most of the protocols proposed in research papers have not been implemented, and experience with real-life applications is very limited. The goal of WP9 is the investigation of cryptographic aspects of the proposed solution. Some of the issues addressed in this work package are:

1. Survey of state-of-the-art in cryptographic tools.
2. Identify the cryptographic tools and SMC framework to use.
3. Design of the underlying building blocks.
4. Analysis of the above building blocks: (a) security, (b) complexity/performance, (c) functionality/usability, (d) tradeoffs.

Deliverable 9.2 Objectives: The aim of WP9.2 is to identity bottlenecks, describe security guarantees and performance metrics, and to identify open issues. The eventual goal is to serve as a guideline for implementing the final prototype. The recommendations of this work package will form the basis of the design decisions in WP3.2 and WP4, and the feedback will be used for further work in WP9.3.

The goal of this report is to summarize the work of WP9.2 - investigation of the performance, security and functional aspects of the building blocks to be used in designing secure linear programming protocols. In this document, we focus only on building blocks for secure ST-Simplex. Other variants can be obtained as extensions.

1.4 Summary

Cryptographic Primitives Used: The following is a summary of the cryptographic primitives used in our protocols: (1) Secret sharing schemes: Shamir's secret sharing [28]; Additive secret sharing over finite fields and integers; Replicated Secret Sharing (RSS) [11]; Replicated Integer Secret Sharing (RISS) [15]; and Pseudo-random Replicated Secret Sharing (PRSS) [11], (2) Pseudo-Random Functions (PRFs), (3) Diffie-Hellman key agreement, and (3) one-time pad encryption using addition or multiplication in a finite field.

Analysis Presented: Our analysis comprises of the following. Complexity analysis consists of three abstracts metrics: (1) rounds, (2) invocations, and (3) exponentiations. These metrics describe respectively the quantities data transmitted, communication time, and local computation. Each metric is also associated with a corresponding finite field, which is also explicitly given. For protocols that are new or optimized variants of standard protocols, we also discuss two other aspects: (1) correctness and (2) security. For standard protocols, we refer to the original papers and skip this analysis. Finally, we give some recommendations for future work.

Organization: This report is designed to be comprehensive with every needed building block explicitly described. The analysis is done based on earlier deliverables (D9.1, D3,1) and any security/correctness analysis done therein is not repeated here.

Chapter 2 introduces the security models, the metrics and the cryptographic framework to be used in the SecureSCM project and for the protocols in the remaining chapters. Chapter 3 discusses basic primitives used for several protocols in remaining chapters. These are protocols for random number generation and related operations. Chapter 4 discusses k-ary associative operations such as multiplication, AND, OR, XOR. Chapter 5 discusses protocols for arithmetic and any remaining operations. Protocols for secure Simplex are given in Chapter 6. Finally, conclusions, assessment and recommendations are presented for each chapter (or section, as suitable) and summarized in Chapter 7.

2 Models and Methods

In this chapter, we introduce the models, methods and notation used in the remaining part of this document. The aspects we discuss here are: the communication and security models, complexity metrics, tradeoffs, SMC framework, and data representation.

2.1 Communication Model

Secure computation is an distributed computation. Therefore, in order to make a reliable performance and security analysis, it is necessary to make some assumptions of the underlying communication model.

Type of communication channels: One of the first assumptions we make is about type of available channels. Although several of our protocols require broadcast of public information, for simplicity we assume only one-to-one reliable full-mesh channels. Broadcast is achieved by sending a copy of the message to every party. Reliability implies only guaranteed delivery and not any security properties such as authenticity or privacy.

Security of communication channels: Cryptographic protocols should withstand an adversary having full control of the underlying communication infrastructure. This implies that the parties conceptually communicate by exchanging messages via the adversary. The second assumption we will make is that in addition to guaranteed delivery, the above unicast channels also implicitly provide source authentication and confidentiality.

2.2 Security Models

Security in SMC is defined using several parameters - adversary structure, type of attacks, computation power of adversary and information leakage. The adversary in our model has the ability to *corrupt* parties. The set of all possible sets of corrupted parties is called the *adversary structure*. We follow the general model of a *threshold* adversary structure, where the adversary can corrupt up to t parties. The value t is called the *corruption threshold* (or simply the threshold). A corruption implies that the adversary obtains read (and sometimes write) access to that party's memory and communication channel. An adversary with read-only access (i.e., the ability to only observe) is called a *passive* adversary, while one with read-write access (i.e., the ability to not only observe but also modify memory contents) is called *active*. Assuming that parties behave correctly by default, a passive adversary cannot alter the behavior of corrupted parties. However, an active adversary has the ability to alter the messages transmitted by corrupted parties, and thus, their behavior.

We will follow the standard notation given in deliverable D9.1 [26] (and the references therein) without any further elaboration on these concepts. For efficiency reasons, we make the following assumptions:

1. We will assume a computationally bounded adversary, even though most of our protocols offer protection even against unbounded adversaries.

2. We will assume *static* corruptions where the adversary must corrupt parties at the start of the computation and cannot corrupt parties during or after the computation.

2.2.1 Active v/s Passive Security

The analysis of the building blocks is done assuming only passive attacks. However, most of these building blocks can be adapted to active attacks using standard transformation at the cost of increased complexity.

In Chapter 7 we discuss some approaches for converting passively secure protocols to actively secure ones.

2.2.2 Statistical Security

Security in our protocols is of two types: *perfect* and *statistical*. The difference between the two notions is discussed below. Some of our building blocks provide only statistical security in order to achieve better efficiency.

Statistical Distance: We use the notion of statistical distance to give a quantitative notion of security. Let X and Y be two random variables. Then the statistical distance $\Delta(X, Y)$ is defined below:

Definition 2.1. *Let X and Y be two random variables, both taking values in some finite set V. The statistical distance between X and Y is defined as*

$$\Delta(X;Y) = \frac{1}{2}\sum_{v\in V} |\mathbb{P}(X = v) - \mathbb{P}(Y = v)|. \tag{1}$$

Intuitively, if $\Delta(X;Y)$ is small (or zero), then the distributions of X and Y are statistically (resp. perfectly) indistinguishable.

All our building blocks are based on the following high-level idea. Let x be a random variable denoting a secret integer. We first generate a random secret integer r in some range and reveal $f(x, r)$ for some function f.[2] Let $\delta = \Delta(r, f(x, r))$. The type of security offered by the protocol depends on δ as follows: $\delta = 0$ implies perfect security and $\delta \leq c/2^{\kappa}$ (for some constant c) implies statistical security in security parameter κ.

Following are some basic results about statistical distance. Their proofs can be found in Chapter 8 of [29].

We first show that if U is uniform on some finite set then the statistical distance between $X + U$ and U can be bounded by the size of the domain of U.

Lemma 1. *Let M and K be positive integers with $M \leq K$. Let X, U be random variables in $[0..M-1]$, $[0..K-1]$ respectively such that U is uniform. Then $\Delta(U; X+U) \leq (M-1)/K$ and this bound is tight.*

Proof. This is Lemma 1 in [25, Appendix A]. □

Remark 2.1. The result of Lemma 1 implies that $\Delta(U; X+U)$ is small if $M \ll K$. For instance, if one sets $K = M2^k$, we see that the statistical distance between U and $X+U$ is less than $1/2^k$, hence approaches 0 exponentially fast as a function of k. In other words, one can mask an integer value X from a bounded range $\{0, \ldots, M-1\}$ by adding a uniform random integer U from an enlarged range $\{0, \ldots, K-1\}$. This way one can do one-time pad encryption with integers, where X is the message, U is the one-time pad, and $X + U$ is the ciphertext.

[2] This includes methods based on secret sharing and additive or multiplicative hiding - $f(x, r) \in \{x+r, xr\}$ for field elements x, r. This is similar to one-time pad encryption where r is the pad (i.e., key) and x is the plaintext.

In Theorem 4, we show that this holds even if U is not uniform, but a sum of uniform distributions. For this we will use the following lemmas.

Lemma 2. *Let X and Y be random variable taking values in some finite set V and let $f : V \to V'$ be some function mapping to some finite set V'. It holds that*

$$\Delta(f(X); f(Y)) \leq \Delta(X; Y). \tag{2}$$

Proof. This is Theorem 8.32 of [29]. □

Lemma 3. *Let X,Y and Z be random values, where X and Z are independent and Y and Z are independent, then*

$$\Delta((X, Z); (Y, Z)) = \Delta(X; Y). \tag{3}$$

Proof. This is Theorem 8.33 of [29]. □

Theorem 4. *Let $X \in [0..M-1]$ and U be random variables and let $U = \sum_{i=1}^{n} U_i$ for some finite n, where each U_i is independent and uniform in $[0..K-1]$. Then:*

$$\Delta(X + U; U) \leq \frac{M-1}{K}. \tag{4}$$

Proof. Let $U_i \in_R [0..K-1]$ for $i = 1, \ldots, n$ such that U_i is selected uniformly and let and $X \in [0..M-1]$ be with unknown distribution. Let $= \sum_{i=1}^{n} U_i$. Lastly, let $f : [0..(n-1)K-n+1] \times [0..M+K-2] \to [0..M+nK-n-1]$ be defined as $f(x, y) := x+y$. It follows that

$$\begin{aligned}
\Delta(X + U; U) &= \Delta(X + \sum_{i=1}^{n} U_i; \sum_{i=1}^{n} U_i) \\
&= \Delta(X + \sum_{i=1}^{n-1} U_i + U_n; \sum_{i=1}^{n-1} U_i + U_n) \\
&= \Delta(f(\sum_{i=1}^{n-1} U_i, X + U_n); f(\sum_{i=1}^{n-1} U_i, U_n)) \\
\text{Lemma 2} &\leq \Delta((\sum_{i=1}^{n-1} U_i, X + U_n); (\sum_{i=1}^{n-1} U_i, U_n)) \\
\text{Lemma 3} &= \Delta(X + U_n; U_n) \\
\text{Lemma 1} &\leq \frac{M-1}{K}
\end{aligned}$$

□

From Theorem 4 it follows that if $K = M2^k$, then $\Delta(X + U; U)$ decreases exponentially fast in k. On the other hand, $U = \sum_{i=1}^{n} U_i \in [0..nK-n]$ such that in order to get the same bound on the statistical distance as when U would be uniformly random the domain of U is increased with a factor n.

Theorem 5 is an extension of Theorem 4 where U is constructed in a slightly different manner. This is used for instance in the security proof of Protocol 5.2.

Theorem 5. *Let $X \in [0..M-1]$ and U be random variables and let $U = U' + K' \sum_{i=1}^{n} U'_i$, where $U' \in_R [0..K'-1]$ and each U'_i is uniform and independent in $[0..K''-1]$. Then:*

$$\Delta(X+U;U) \leq \frac{M-1}{K'K''}. \tag{5}$$

Proof. Let $U_n = U' + K'U'_n$ and $U_i = K'U'_i$ for $i = 1, \ldots, n-1$. Observe that U_n is uniform in $[0..K'K''-1]$, and U_i are independent. Also, let

$$f : [0..(n-1)K'(K''-1)] \times [0..M + K'K'' - 2] \rightarrow [0..M + nK'(K''-1) + K' - 2]$$

be defined as

$$f(x,y) := x + y$$

Using the same method as in the proof of Theorem 4 we obtain:

$$\begin{aligned}
\Delta(X+U;U) &= \Delta(X + \sum_{i=1}^{n} U_i; \sum_{i=1}^{n} U_i) \\
&= \Delta(X + \sum_{i=1}^{n-1} U_i + U_n; \sum_{i=1}^{n-1} U_i + U_n) \\
&= \Delta(f(\sum_{i=1}^{n-1} U_i, X + U_n); f(\sum_{i=1}^{n-1} U_i, U_n)) \\
\text{Lemma 2} &\leq \Delta((\sum_{i=1}^{n-1} U_i, X + U_n); (\sum_{i=1}^{n-1} U_i, U_n)) \\
\text{Lemma 3} &= \Delta(X + U_n; U_n) \\
&= \Delta(X + U' + K'U'_n; U' + K'U'_n) \\
\text{Lemma 1} &\leq \frac{M-1}{K'K''}
\end{aligned}$$

□

The particular case used in our protocols is $M = 2^k$, $K' = 2^k$, and $K'' = 2^\kappa$. It follows that $\Delta(X+U;U) \leq \frac{2^k-1}{2^{k+\kappa}} < 2^{-\kappa}$. The range of U is $[0..n2^k(2^\kappa - 1) + 2^k - 1]$.

The security of every protocol with statistical security presented in this document follows from one of the above theorems.

2.2.3 Universal Composability

In order to talk about security analysis, it is necessary to discuss a concept named *Universal Composability* (UC). Without going into too much technical details (for which we refer the reader to [7, 6, 26]), we discuss the main aspects relevant to this document.

Roughly speaking, UC implies some sort of security when two building blocks are composed to form a larger building block. The main UC theorem [6] says that if protocols P_1, P_2 are UC, then any resulting protocol formed using (only) P_1, P_2 as building blocks in any imaginable manner is also UC.[3] Furthermore, being UC implies security under standard requirements (such as privacy of inputs, etc).

To prove the security of a protocol P, we can utilize one of the following methods:

[3] Note that only the weakest security notion is preserved. For instance if P_1 provides perfect security, while P_2 provides only statistical security, then the combined protocol will provide only statistical security.

1. Construct a simulator such that any information extracted by an adversary interacting with the parties in the real execution of P can also be extracted by the simulator interacting with a trusted party implementing the *ideal* functionality of P.

2. Show that P is UC by showing that every sub-protocol P_i of P is UC in the following sense. For every P_i, construct a simulator that interacts with a trusted party implementing the ideal functionality of P_i and outputs something that cannot be distinguished (by any external entity) from the output of the adversary interacting with the parties in a real execution of P_i.

Although the first approach allows us to prove the security of P, it does not guarantee security of a larger protocol in which P may be a sub-protocol. On the other hand, using the second approach allows us to claim that P itself is UC, which allows us to use it in a larger protocol without any loss of security.

Most of the protocols used in this work are standard in the literature with well accepted guarantees of security and we will not prove security of those protocols. The remaining protocols are optimized variants with non-standard techniques. For these, we will follow the second approach by sketching out a simulator for each of their sub-protocols.

2.3 Complexity Analysis

In this section we describe the metrics we use in our complexity analysis. The purpose of this analysis is to give quantitative results about the overall running time of the protocols, using abstract metrics, independent of implementation and execution environment.

As we shall see, the following four types of analysis are sufficient to obtain an estimate of the running time of the protocols without fully implementing them: (1) data transmitted, (2) communication time, (3) local computation, and (4) empirical results. We discuss the concepts in detail below.

Computation model. Secure computation requires frequent communications between parties. Consider, for example, a secure computation system based on the Shamir's secret sharing scheme over a finite field. The system is based on the following operations with secret field elements: input sharing (each party obtains a secret share of each private input); addition/multiplication of shared values; addition/multiplication of public and shared values; output recovery (each party obtains the output of a computation, reconstructed from shares). These primitives are used to build protocols for any other secure computation task. Multiplication of shared values, input, and output require interaction between parties, while the other operations can be locally computed.

The parties run local instances of a protocol and communicate whenever they execute an interactive operation. Causality relations define a partial order of the start and finish times of the operations: a party can start an operation as soon as its input is available, and may start several operations in parallel; on the other hand, a party can finish an interactive operation ϕ only after receiving the necessary data from the other parties; any operation whose input depends on ϕ can start only after finishing ϕ.

A simple computation model based on these remarks is sufficient for the analysis of our protocols. A secure computation is structured into rounds according to the causality relations determined by the protocol (algorithm and secure computation primitives). Round r groups the interactive operations that can start after finishing round $r-1$. We

assume that the set of operations executed in a round is defined when the round begins, and they are all finished before starting other operations (next round).

We distinguish the special case of operations whose start time does not depend on the completion of other operations (e.g., joint generation of secret random values). These operations can be precomputed, in the sense that if their output is needed in round r they may be executed in any round $r' \leq r$. By convention we assign them to round r, but they may be grouped in a precomputation round.

An implementation can schedule the interactive operations according to the partial order relation, such that to minimize the effects of communication and synchronization delays. We cannot analyze all executions paths that would be allowed by scheduling any operation as soon as its input is available. However, the structure of rounds defined above offers a good approximation of the parallelism allowed by the protocol, in order to reduce the communication delays. Moreover, important performance gains can be obtained by precomputation.

2.3.1 Communication Complexity

This quantity defines how much data is transmitted in a given protocol. In quantitative terms, we are interested in the total network bandwidth consumption. Instead of several possible metrics to define this overhead (such as octets per party, total octets, etc), we decided to use the metrics called *invocations*. This was done because all the protocols discussed in this work have some common features: (1) they are symmetric with respect to the parties (i.e., all parties have almost identical communication overhead), (2) the number of parties and corruption threshold is same in all protocols, and (3) our communication model precludes true broadcast channels (broadcast is emulated by unicast). In this setting, one invocation turns out to be the minimum communication overhead in protocols requiring interaction. Furthermore, it turns out that the actual communication overhead in every protocol is always an integer multiple of one invocation.

Invocation: Assume that the only means to communicate between parties is via secure one-to-one channels. An invocation is a secure computation primitive achieved with a single interaction between parties. During this interaction every party sends a message to every other party. In our protocols the payload of a message is a secret share of a value, and a share is an element of a finite field $\mathbb{F}$.

The unit of communication complexity is the amount of data transmitted by a party during an invocation, i.e., when every party transmits to every other party an element of $\mathbb{F}$. We will always consider an invocation with a corresponding field. Note that it is easy to translate invocations to octets by considering the field elements to be transmitted in some encoding and the number of parties.

Therefore, the metrics for communication complexity is the number of invocations performed during a protocol run. This is proportional to the actual amount of data sent or received, and conveniently abstracts away common factors like configuration parameters (number of parties, type and size of field) and implementation dependent parameters (encoding of field elements, message format).

Remark 2.2. It is important to note that we use invocations as the metrics for the amount of data transmitted. In particular, invocations do not consider the overhead involved in

preparing/processing the transmitted/received data. This overhead is measured using a different quantity called computation complexity (see Section 2.3.3).

2.3.2 Round Complexity

Roughly speaking, this quantity defines how much time is used for communication between parties without considering the amount of data. As before, we are interested in abstract metrics of this quantity. It turns out that invocations allows us to formulate similar metrics for time. Recall that one invocation is the fundamental interaction between parties. Assume that the secure computation is time-optimized by executing in parallel all invocations in a round. Then we can take the amount of time needed to complete one invocation as unit for interaction time. We call this unit a *round*.

Round: A *round* is the time needed for one invocation. If a protocol step requires two or more invocations that can be done in parallel, then that step still counts as only 1 round of time. On the other hand, if two or more invocations in a step cannot be done in parallel, then each such invocation adds one more round to the protocol. It turns out that in all our protocols, whenever two or more invocation can be done in parallel, they are always of the same type. Thus, even if invocations in two different fields take different times, it is possible to use the total number of rounds and the type of invocation(s) in each round to obtain a realistic estimate of the actual communication time.

The round complexity of a protocol is the number of rounds necessary to complete a protocol run (according to the computation model given above). This metric takes into account the inherent network delays, which are independent of the amount of data sent.

An estimate of the overall communication time (for an ideal implementation of the computation model) can be obtained by combining the number of rounds and the number of invocations.

2.3.3 Computation Complexity

Roughly speaking, this quantity defines the amount of computation time locally needed by each party during a protocol run. In typical configurations, the overall protocol running time is dominated by the communication time. However, if a protocol is executed a large number of times in parallel, the local computation becomes prominent (e.g., during precomputation or in Simplex iterations). Therefore, it becomes useful to get a theoretical estimate of the computation complexity.

One way to obtain an accurate metric is to count the number and type of field operations done by each party before and after each sequential invocation(s). However, in practice, addition and multiplication are fast enough to be discarded from analysis. The only expensive operation is field exponentiation. Therefore, our basic unit of measuring local computation is an *exponentiation*.

2.3.4 Empirical Results

The above theoretical analysis is almost sufficient to obtain an estimate of the running time. Assuming that the total number of rounds, the number and type of invocations in each round, and total number of exponentiations are known. Then the only missing information is the time for one invocation of each type and the time for one exponentiation.

These two parameters can be obtained using experiments. For simple protocols with few invocations, the time obtained from this analysis should be accurate.

On the other hand, for complex protocols with several tens of rounds and several thousand invocations per round, several other factors come into play. The main problem is efficient scheduling and synchronization of network messages in several parallel invocations. The second problem is of optimally parallelizing local computation with the round computation.

In order to gain better understanding of the performance of secure Simplex, it is necessary to make some empirical tests. Some of the tests suggested are: running time with several rounds and several thousand parallel invocations in each round.

2.3.5 Tradeoffs

In this section we consider several possible tradeoffs that may result in improved performance at the expense of some other property.

Precomputation: One of the tradeoffs we can make is to use precomputation, which trades storage with time. Many protocols use large amounts of secret random values (unknown to any party) and spend an important amount time to generate them. These random values can be precomputed, either before starting the secure computation, or before starting the protocol that needs them. Precomputation can reduce the running time of the protocols, but requires additional resources.

For example, each iteration of secure Simplex has to generate a large number of secret random bits. Each bit requires interaction between all parties and one exponentiation by each party. We cannot precompute the random bits needed during the entire protocol run, because the number of iterations is not known in advance (and the total amount is huge for a large linear program). However, we can substantially reduce the running time by precomputing the random bits needed by an iteration. We can reduce the number of rounds by generating all the bits in parallel, instead of small batches, sequentially. Furthermore, a multi-processor computer can run the precomputation for iteration i in parallel with the iteration $i-1$. The results of precomputation experiments with secure Simplex are presented in Section 6.3.

Round versus communication complexity: When designing a protocol we often have to choose between solutions that reduce the communication/computation complexity but need more rounds, and solutions that save rounds but increase the communication/computation complexity. We want the solution with the best performance, but the performance depends on several factors, including protocol implementation, execution environment, and application.

This tradeoff applies to many of the secure computation tasks discussed in the next chapters. A particular issue is the choice between encoding all data types in the same field, which saves rounds but can increase the communication complexity, and efficient data encoding in different fields, which reduces the communication complexity but requires additional rounds for conversions (Section 5.5).

Functionality: A possible tradeoff in functional aspects is to reduce accuracy in order to gain performance. This tradeoff applies to secure multiplication and division of fixed-

point numbers and their building blocks (Sections 5.1 and 5.4): we can choose a more efficient protocol with lower accuracy, or a more complex protocol with better accuracy.

Another functional tradeoff is with respect to correctness, where we allow a small probability of getting incorrect results in order to gain performance. Several protocols proposed in the literature (e.g., for comparison) offer probabilistic correctness, with error probability that can be made arbitrarily small.

As another example, the variant of secure Simplex presented in Chapter 6 does not use Bland's rule in selecting the pivot. Although this does not cause any undesirable effects in general, there is always a very small probability of entering an infinite loop.

Security: Finally, as another tradeoff, we can gain performance by relaxing the security requirements. Many of the protocols presented in Chapter 5 are designed for statistical privacy, instead of perfect privacy, in order to improve their efficiency. Based on the same ideas one can construct protocols that offer perfect privacy, but their complexity is substantially higher. As another example, Chapter 3 presents several protocols with statistical privacy (based on RISS) for secret random values, in addition to slower protocols with perfect privacy.

The protocols mentioned above are based on standard security notions. They are general building blocks that can be used in any application. On the other hand, since performance is a critical issue in our applications, we consider further relaxing the security requirements, in particular cases. The security of those protocols is not rigorously quantifiable (as an example, TruncPrN). However, we give an intuitive argument for their security when they are used as a sub-protocol in certain contexts (such as fixed-point multiplication during Simplex iterations).

2.4 SMC Framework

Notation: The symbol $a \rightarrow i$ denotes that value a is sent to party (or set of parties) represented by i over a secure channel. The symbol $(a_1, a_2, \ldots a_m) \rightarrow (i_1, i_2, \ldots, i_m)$ is shorthand for $(a_j \rightarrow i_j)$ for $1 \leq j \leq m$. The symbol $a \Rightarrow i$ indicates that a is sent to i over a public broadcast channel.

Our secure computation problem involves some n parties with $n \geq 3$ working on secret data. The basic framework we use is that of Shamir secret sharing [28]. In this framework, secret values are represented as share vectors of the Shamir sharing scheme. Each party holds one share of the secret and a threshold number of parties must pool their shares in order to obtain the secret. The protocol uses the Lagrange form of polynomial interpolation and is described in Section 2.4.1.

All the protocols discussed in this section are standard in the literature. The provide perfect privacy and are UC-secure against a passive adversary. Consequently, we will not present a security analysis of these protocols. The reader is referred to D9.1 [26] and the references therein for the security proofs.

2.4.1 Shamir's Secret Sharing

The scheme is defined using the following parameters: an integer $n \geq 2$ (denoting the number of parties), an integer $t < n$ denoting a threshold (maximum number of parties the adversary can corrupt) and a finite field $\mathbb{F}$ denoting the secret-space. The *Dealer* is the party who holds a secret and might be one of the n parties.

Goal: Construct n shares of the secret such that any $t+1$ or more shares can be used to reconstruct the secret and any t or fewer shares reveal zero information about the secret. In typical situations, the dealer secretly distributes one share to each of the n parties. The scheme has three phases described below:

1. *Share generation:* The secret s is an element of some finite field $\mathbb{F}$.

   The dealer uniformly selects t field elements $a_1, a_2, \ldots, a_t \in \mathbb{F}$. He then constructs the degree-t polynomial $p(x) = s + a_1 x + a_2 x^2 + \ldots a_t x^t \in \mathbb{F}[x]$ and computes $s_i = p(i)$ for $1 \leq i \leq n$. The n-vector $(s_1, s_2, \ldots, s_n)$ is called a *sharing* of s. We denote by $\mathsf{RandShare}(s, n, t)$ the function that computes a sharing of s with parameters n, t.

2. *Share distribution:* $s_i \rightarrow i$ for $1 \leq i \leq n$.

3. *Secret reconstruction:* Choose any subset $J \subset [1..n]$ s.t. $|J| = t+1$ and pool their shares to reconstruct the secret s as:

$$s = \sum_{j \in J} \left(s_j \prod_{i \in J, i \neq j} \frac{-i}{j-i} \right)$$

Computation: In algorithm RandShare, all except the first term needed in the computation of s_i can be precomputed without knowing the secret. Then this algorithm needs n additions in $\mathbb{F}$. Similarly, the vector of products needed for reconstruction can be precomputed. Then this step needs $t+1$ multiplications and t additions in $\mathbb{F}$.

Security: If the number of available shares is $\leq t$, they reveal no information about s. Thus, the scheme provides *information theoretic* or *perfect* privacy. For details see [27].

Notation: We use the following shorthand notation. If $s \in \mathbb{F}$ is the secret then for any n, t, the symbol $[s]$ denotes a sharing of s. In other words, $[s]$ is the vector $(s_1, s_2, \ldots, s_n)$. In the context of this document, the word "secret" refers to any value that is shared using a secret-sharing scheme (such as above).

2.4.2 Arithmetic with Secret Field Elements

We give some building blocks used throughout the document. These are protocols for addition, multiplication and related operation with secret-shared field elements.

Addition of secrets: Each party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{F}$. Party i computes $c_i = a_i + b_i \in \mathbb{F}$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[a+b]$.

Addition of a secret with a public field element: Each party $i \in [1..n]$ holds a_i, the share of $a \in \mathbb{F}$. Let $\alpha \in \mathbb{F}$ be a public value. Party i computes $c_i = \alpha + a_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha + a]$.

Multiplication of a secret with a public integer or field element: Each party $i \in [1..n]$ holds a_i, the share of $a \in \mathbb{F}$. Let $\alpha \in \mathbb{Z} \cup \mathbb{F}$ be a public value. Party i computes $c_i = \alpha a_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha a]$.[4]

[4] When multiplying by a public integer (instead of a field element), $\mathbb{F}$ is considered as a $\mathbb{Z}$-module.

Linear Combination of secrets: Each party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{F}$. Let $\alpha, \beta \in \mathbb{Z} \cup \mathbb{F}$ be public values. Party i computes $c_i = \alpha a_i + \beta b_i$. Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\alpha a + \beta b]$.

Multiplication of secrets: Each party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{F}$. They follow the following protocol. Note that this requires $t < n/2$.

Protocol 2.1: $[c] \leftarrow \mathsf{Mul}([a], [b])$

1 **foreach party** $i \in [1..2t+1]$ **do parallel**
2 $\quad d_i \leftarrow a_i b_i$;
3 $\quad (d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \leftarrow \mathsf{RandShare}(d_i, n, t)$;
4 $\quad (d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \rightarrow (1, 2, \ldots, n)$; `// 1 rnd, ≈1 inv (𝔽)`
5 **foreach party** $j \in [1..n]$ **do**
6 $\quad c_j \leftarrow \sum_{i=1}^{2t+1} \left(d_{(i,j)} \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
7 $\quad$ **return** c_j;

Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[ab]$.

Inner Product of Secrets: Let $[a], [b], [a'], [b']$ be sharings. The parties now want to jointly compute the sharing $[h] = [ab + a'b']$. Using Protocol 2.1 the naive way would result in each party $j \in [1..n]$ computing c_j, c'_j and adding them locally to get the final result. In other words, party j would compute

$$\sum_{i=1}^{2t+1} \left(d_{(i,j)} \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right) + \sum_{i=1}^{2t+1} \left(d'_{(i,j)} \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$$

$$= \sum_{i=1}^{2t+1} \left((d_{(i,j)} + d'_{(i,j)}) \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right),$$

where $d_{(i,j)}$ and $d'_{(i,j)}$ are j's shares in the sharings $[a_i b_i]$ and $[a'_i b'_i]$ respectively. From the addition protocol, it is clear that $d_{(i,j)} + d'_{(i,j)}$ is j's share in the sharing $[a_i b_i + a'_i b'_i]$. Thus, it is more efficient to first compute $a_i b_i + a'_i b'_i$ and then obtain j's share from the sharing $[a_i b_i + a'_i b'_i]$. Based on this, we obtain a generalization for computing the inner product of two m-vectors.

Notation: Let $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m) \in \mathbb{F}^m$ be an m-vector of field elements. For $i \in [1, m]$, let $[\mathbf{a}_i] = (a_{(i,1)}, a_{(i,2)}, \ldots, a_{(i,n)})$ be a sharing of $\mathbf{a}_i$. For $j \in [1, n]$, define $\mathbf{a}^*_j \stackrel{\text{def}}{=} (a_{(1,j)}, a_{(2,j)}, \ldots, a_{(m,j)})$. By $[\mathbf{a}]$, we denote the n-vector $(\mathbf{a}^*_1, \mathbf{a}^*_2, \ldots, \mathbf{a}^*_n)$

Let $\mathbf{a}, \mathbf{b}$ be two m-vectors. Each party $i \in [1..n]$ holds two m-vectors $\mathbf{a}^*_i, \mathbf{b}^*_i$ as defined above. To compute the sharing $[\mathbf{a} \cdot \mathbf{b}]$, they use Protocol 2.2.

Protocol 2.2: $[c] \leftarrow \mathsf{Inner}([\mathbf{a}], [\mathbf{b}])$

1 **foreach party** $i \in [1..2t+1]$ **do parallel**
2 $\quad d_i \leftarrow \mathbf{a}_i^* \cdot \mathbf{b}_i^*$; // d_i = inner product of $\mathbf{a}_i^*$ and $\mathbf{b}_i^*$
3 $\quad (d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \leftarrow \mathsf{RandShare}(d_i, n, t)$;
4 $\quad (d_{(i,1)}, d_{(i,2)}, \ldots, d_{(i,n)}) \rightarrow (1, 2, \ldots, n)$; // 1 rnd, ≈1 inv (𝔽)
5 **foreach party** $j \in [1..n]$ **do**
6 $\quad c_j \leftarrow \sum_{i=1}^{2t+1} \left(d_{(i,j)} \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
7 $\quad$ **return** c_j;

Then $[c] = (c_1, c_2, \ldots, c_n)$ is the sharing $[\mathbf{a} \cdot \mathbf{b}]$.

In Protocols 2.1, 2.2, we assumed that only the first $2t+1$ parties are involved in the initial phase. However, to optimize communication and computation load, it is possible to select any arbitrary subset of $2t+1$ parties. Details are given in [26].

In most of our computation, the field $\mathbb{F}$ will be the field $\mathbb{Z}_q$ for some prime $q > 2$. In some cases, it will be the field $\mathbb{F}_{2^m}$ for some integer m.

Shorthand Notation: For convenience, we will use the infix notation to denote the addition, subtraction, multiplication and linear combination protocols. For instance, $[c] \leftarrow [a][b]$ indicates that $[c]$ is the (secret) output of the protocol for multiplication with $[a], [b]$ as inputs. Similarly $[c] \leftarrow a[b]$ indicates that $[c]$ is the output of the protocol for multiplication with public input a and secret input $[b]$.

2.4.3 Input and Output

For convenience, we define the following two protocols:

Protocol 2.3: $[a] \leftarrow \mathsf{Input}(a, j)$

1 **for party** j **do**
2 $\quad (a_1, a_2, \ldots, a_n) \leftarrow \mathsf{RandShare}(a, n, t)$;
3 $\quad (a_1, a_2, \ldots, a_n) \rightarrow (1, 2, \ldots, n)$; // 1 rnd
4 **foreach party** $i \in [1..n]$ **do**
5 $\quad$ **return** a_i;

Protocol 2.4: $a \leftarrow \mathsf{Output}([a])$

1 **foreach party** $i \in [1..t+1]$ **do parallel**
2 $\quad a_i \rightarrow [1..n]$; // 1 rnd, ≈1 inv (𝔽)
3 **foreach party** $j \in [1..n]$ **do**
4 $\quad a \leftarrow \sum_{i=1}^{t+1} \left(a_i \prod_{\ell=1, \ell \neq i}^{t+1} \frac{-\ell}{i-\ell} \right)$; // local computation
5 $\quad$ **return** a;

2.5 Data Representation

Although we use the above SMC framework with arithmetic on secret field elements, the data in our application domain consists of signed integers, signed fixed-point numbers and boolean values. In this section we discuss how to map application data to field elements in

order to do secure computation with them. The reverse mapping is performed to extract the application data after the computation.

Integers in the range $[-2^\ell..2^\ell]$ will be represented as elements of a field $\mathbb{Z}_q$ with prime q such that $q > 2^{\ell+1}$. For some integer α in the range, the corresponding field element representation is $\alpha \bmod q$.

Fixed point numbers in the range $(-2^e - 1, 2^e + 1)$ with resolution (number of fractional bits) f will be represented as integers in the range $[-2^{e+f}..2^{e+f}]$, which are then represented as field elements of $\mathbb{Z}_q$ as described above. For the fixed point number β in the range, the corresponding integer representation is $2^f\beta$.

Notation: Let $\mathbf{Fld}_q : [-2^\ell..2^\ell] \mapsto \mathbb{Z}_q$ be the function that maps integers to their field element representation and let $\mathbf{Int}_f : (-2^e - 1, 2^e + 1) \mapsto [-2^{e+f}..2^{e+f}]$ be the function that maps fixed-point numbers with resolution f to their integer representation. In other words, $\mathbf{Fld}_q(x) = x \bmod q$ and $\mathbf{Int}_f(x) = 2^f x$.

2.5.1 Boolean Operations

Every field $\mathbb{F}$ consists of the additive and multiplicative identities, 0 and 1 respectively. These can be used to encode the boolean variables $True$ and $False$ respectively. The arithmetic operations with secret field elements can be used to perform boolean operations on the sharings of these bits. Table 2 in Section 2.6 shows how to evaluate the basic boolean functions using the protocols of Section 2.4.2.

2.5.2 Integer Arithmetic

Using the above representation, the arithmetic with integers can be emulated using modular arithmetic in $\mathbb{Z}_q$. In particular, for any integers $a, b \in [-2^\ell..2^\ell]$, the operation $a \odot b$ for $\odot \in \{+, -, \times\}$ can be done as:

$$a \odot b = \mathbf{Fld}_q^{-1}(\mathbf{Fld}_q(a) \odot \mathbf{Fld}_q(b))$$

The above can be done using protocols presented in Section 2.4.2. The protocol for computing inner product of two integer vectors is a trivial extension of Protocol 2.2 for field elements and we will not elaborate further on these.

Furthermore, if $b|a$ then the division a/b can be done as

$$a/b = \mathbf{Fld}_q^{-1}(\mathbf{Fld}_q(a) \times \mathbf{Fld}_q(b)^{-1}).$$

Assuming that $b \neq 0$, Protocol 3.12 (Inv) of Section 3.2.4 can be used to compute inverse of secret field elements. Protocols for comparison of integers are given in Section 5.2.

2.5.3 Fixed-Point Arithmetic

Using the above representation, the operations of addition, subtraction and comparison with fixed point numbers are similar to those with integers. To do an operation on fixed point numbers a, b, we do the same operation on their integer representations $\mathbf{Int}_f(a), \mathbf{Int}_f(b)$. Depending on the operation, the output then is either the correct integer representation of the fixed-point result (for addition, subtraction) or a boolean value (for comparison).

Multiplication, on the other hand is not so straightforward and division is even more complicated. In the multiplication protocol, we have two main steps: (1) multiply the integer representations $\mathbf{Int}_f(a), \mathbf{Int}_f(b)$ of two fixed-point numbers with resolution f to obtain the integer representation $\mathbf{Int}_{2f}(ab)$ of a fixed point number with resolution $2f$, and (2) truncate this resulting value by f bits to obtain the integer representation of a number with resolution f.[5]

Truncation of an integer (or an integer representation) α by f bits follows this general procedure: (1) compute $\alpha' = \alpha \bmod 2^f$, and (2) compute $\alpha'' = (\alpha - \alpha')/2^f$, the truncated value. Division by 2^f is carried out by multiplication with $2^{-f} \in \mathbb{Z}_q$.

For the fixed-point division α/β, we perform the (fixed-point) multiplication $\alpha\beta'$, where β' is the (fixed-point) reciprocal of β, and is computed using the Newton-Raphson method. Details are given in Deliverable 3.1 [27] and Section 5.4.

2.6 Summary

Notation: The remaining document is based on the notation and assumptions given here. Unless otherwise specified, all secure computation is done using a (t, n) Shamir secret-sharing scheme in a prime field $\mathbb{Z}_q$ (the "default" field) with q sufficiently large and $q > 2^{2e+2f+\kappa}$ for some security parameter κ, where e, f denote the magnitude and resolution of fixed-point numbers, and t, n denote the corruption threshold and the number of parties. Some of the protocols work in other fields such as $\mathbb{F}_{2^8}$ during intermediate steps.

From this point onwards, by $[x]$ we denote a Shamir sharing of $x \in \mathbb{Z}_q$, the default field. Depending on the context, x may represent any of the following: (1) a signed integer, (b) an unsigned integer, (c) a signed fixed-point number, or (d) a boolean value using the mappings described in Section 2.5. For convenience, we will use the infix notation for Mul and boolean operations.

When dealing with sharings in a non-default field, or when there is any ambiguity, we will superscript the name of the field with the secret. For instance, the Shamir sharing of s in $\mathbb{F}_{2^m}$ will be denoted by $[s]^{\mathbb{F}_{2^m}}$.

Organization: Due to various restriction imposed by the structure of the protocols (i.e., their dependencies), the building blocks are grouped in broad categories (with some minor overlap). This classification is done to ensure that every sub-protocol used at any point has already been described before (i.e., we follow the 'bottom-up' approach). Our high-level organization is as follows:

1. Random number generation (Chapter 3)
2. Unbounded Fan-in (i.e., k-ary) and binary operations (Chapter 4)
3. Arithmetic operations and comparison (Chapter 5)
4. Secure Simplex (Chapter 6)

Complexity of basic protocols: Table 1 gives the complexity of basic protocols.

[5] Note that due to the underlying algorithm, the value of q should be chosen so as to contain integers in the range $[-2^{e+2f}..2^{e+2f}]$.

Operation	Protocol	Rounds	Invocations ($\mathbb{F}$)	Local Computation
$[c] \leftarrow [a] + [b]$		0	0	1 add.
$[c] \leftarrow a + [b]$		0	0	1 add.
$[c] \leftarrow a[b]$		0	0	1 mul.
$[c] \leftarrow \alpha[a] + \beta[b]$		0	0	1 add., 2 mul.
$[c] \leftarrow [a][b]$	Mul	1	1	1 mul., 1 RandShare
$[c] \leftarrow [\mathbf{a}] \cdot [\mathbf{b}]$	Inner	1	1	$k-1$ add., k mul. 1 RandShare
$\forall j : [a_j] \leftarrow \mathsf{Input}(a_j, j)$	Input	1	1	1 RandShare
$a \leftarrow \mathsf{Output}([a])$	Output	1	1	t add., $t+1$ mul.

Table 1: Complexity of basic building blocks.

Field used	Operation	Protocol	How to compute?	Rounds	Invocations
Any ($\mathbb{F}$)	$[c] \leftarrow [a] \wedge [b]$	AND	$[c] \leftarrow [a][b]$	1	1 ($\mathbb{F}$)
Any ($\mathbb{F}$)	$[c] \leftarrow [a] \vee [b]$	OR	$[c] \leftarrow [a] + [b] - [a][b]$	1	1 ($\mathbb{F}$)
Any ($\mathbb{F}$)	$[c] \leftarrow \neg[a]$	NOT	$[c] \leftarrow 1 - [a]$	0	0
Any ($\mathbb{F}$)	$[c] \leftarrow [a] \oplus [b]$	XOR	$[c] \leftarrow [a] + [b] - 2[a][b]$	1	1 ($\mathbb{F}$)
$\mathbb{F}_{2^m}$	$[c] \leftarrow [a] \oplus [b]$	XOR	$[c] \leftarrow [a] + [b]$	0	0

Table 2: Evaluation of boolean functions.

3 Secret Random Number Generation

3.1 Interactive Protocols For Randoms

Generation of secret-shared random values is a fundamental building block in our SMC protocols. In summary, our main goal is to generate a sharing $[r]$ for some unknown $r \in \mathbb{F}$ such that each party holds one share of r and any of the following hold:

1. r is uniform in $\mathbb{F}$ (random field element)
2. r is uniform in $\{0_{\mathbb{F}}, 1_{\mathbb{F}}\}$ (random bit)
3. If $\mathbb{F} = \mathbb{Z}_q$ then $r \in \mathbb{Z}_{2^m} \subset \mathbb{Z}_q$ for some public integer $m < \log_2(q)$.

We describe four basic protocols (Protocols 3.1-3.4). While Protocol 3.1 works in any field, the remaining protocols (Protocols 3.2-3.4) work only in a prime field $\mathbb{Z}_q$.

(A) Random Field Element: Protocol 3.1 is a fundamental primitive used in several protocols. It generates a random Shamir-shared field element in any finite field $\mathbb{F}$.

Protocol 3.1: $[r]^{\mathbb{F}} \leftarrow \mathsf{RandFld}(\mathbb{F})$

1 **foreach party** $i \in [1..n]$ **do parallel**
2 $\quad r_i \xleftarrow{R} \mathbb{F}$;
3 $\quad [r_i]^{\mathbb{F}} \leftarrow \mathsf{Input}(r_i)$; `// 1 rnd, 1 inv (F)`
4 $[r]^{\mathbb{F}} \leftarrow \sum_{i=1}^{n} [r_i]^{\mathbb{F}}$;
5 **return** $[r]^{\mathbb{F}}$;

Security: Protocol 3.1 provides perfect privacy [26].

(B) Random Bits: Protocol 3.2 (RandBit) generates a secret-shared random bit. This protocol, its variants, and all other protocols that use this as a sub-protocol require $\mathbb{F} = \mathbb{Z}_q$ and $q \equiv 3 \pmod 4$.

Protocol 3.2: $[b] \leftarrow \mathsf{RandBit}(q)$

1 $[r] \leftarrow \mathsf{RandFld}(\mathbb{Z}_q)$; `// 1 rnd, 1 inv (Zq)`
2 $[u] \leftarrow [r][r]$; `// 1 rnd, 1 inv (Zq)`
3 $u \leftarrow \mathsf{Output}([u])$; `// 1 rnd, 1 inv (Zq)`
4 $v \leftarrow u^{-(\frac{q+1}{4})} \bmod q$; `// 1 exp (Zq)`
`// ` $uv^2 \equiv 1 \pmod q$
5 $[b] \leftarrow (2^{-1} \bmod q)(v[r] + 1)$;
6 **return** $[b]$;

Security: Protocol 3.2 provides perfect privacy [26].

(C) Random Elements In Range: We present two protocols for generating randoms in range. Both protocols generate a Shamir sharing $[r]$ in a prime field $\mathbb{Z}_q$ of random $r \in [0..2^m - 1]$ with $m < \log_2(q)$. In the first protocol, Rand2mU, the value r is uniformly distributed, while in the second one, Rand2mN, this value is a sum of random uniform integers. Furthermore, Rand2mU additionally outputs a bit-wise sharing of the random.

Protocol 3.3: $[r], [r]_B \leftarrow \mathsf{Rand2mU}(q, m)$

1 **foreach** $i \in [0..m-1]$ **do parallel**
2 $\quad [r_i] \leftarrow \mathsf{RandBit}(q)$; // 3 rnd, $3m$ inv, m exp ($\mathbb{Z}_q$)
3 $[r] \leftarrow \sum_{i=0}^{m-1} 2^i [r_i]$;
4 $[r]_B \leftarrow ([r_0], [r_1], \ldots, [r_{m-1}])$;
5 **return** $[r], [r]_B$;

Security: Protocol 3.3 provides perfect privacy.

As always, the number of parties is n. Let $\tau : \mathbb{Z}_n \times [1..n] \mapsto \{0,1\}$ be a public function s.t. $\forall x \in \mathbb{Z}_n : \sum_{i=1}^{n} \tau(x,i) = x$. Protocol Rand2mN is as follows.

Protocol 3.4: $[r] \leftarrow \mathsf{Rand2mN}(q, m)$

1 **foreach party** $i \in [0..n]$ **do parallel**
2 $\quad \beta_i \leftarrow \lfloor (2^m - 1)/n \rfloor + \tau(2^m - 1 \bmod n, i)$;
3 $\quad r_i \xleftarrow{R} [0..\beta_i]$;
4 $\quad [r_i] \leftarrow \mathsf{Input}(r_i)$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
5 $[r] \leftarrow \sum_{i=1}^{n} [r_i]$;
6 **return** $[r]$;

3.2 Protocols Based on PRSS

First we give some notation. Let there be n parties. We assume a threshold adversary structure with threshold $t < n$. Then the set $A = \{X | X \subset [1..n] \wedge |X| = t\}$ is the set of all maximal unqualified subsets of parties. Note that $|A| = \binom{n}{t} = \frac{n!}{(n-t)!t!}$.

3.2.1 Replicated Secret Sharing (RSS)

RSS [11] is a secret sharing scheme over some finite field $\mathbb{F}$ as described below.

1. *Share Generation:* To share a secret $s \in \mathbb{F}$, the dealer generates $r_i \xleftarrow{R} \mathbb{F}$ for $1 \leq i \leq |A| - 1$ and sets $r_{|A|} = s - \sum_{i=1}^{|A|-1} r_i$. Then $(r_1, r_2, \ldots, r_{|A|})$ is the share vector of s. We use the symbol $[s]^{RQ}$ to denote this share vector.

2. *Share Distribution:* The dealer assigns some arbitrary (public) labeling to elements of A and writes A as $\{X_1, X_2, \ldots, X_{|A|}\}$. Then for $1 \leq i \leq |A|$, he distributes the shares as follows:
$$r_i \rightarrow [1..n] \backslash X_i$$

3. *Secret Reconstruction:* Let $B \in [1..n]$ such that $|B| = t + 1$. Then members of B jointly share the entire vector $(r_1, r_2, \ldots, r_{|A|})$. They compute $s = \sum_{i=1}^{|A|} r_i$.

Security: The scheme provides perfect privacy. See D9.1 [26] and [11] for details.

3.2.2 Conversion of RSS Shares to Shamir Shares

Let $[s]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$ be an RSS sharing of some $s \in \mathbb{F}$ as described above. Then the share vector of each party $i \in [1..n]$ is $(r_j)_{i \notin X_j}$ with $\binom{n-1}{t}$ elements. To convert the

RSS shares of s to Shamir shares of s for the same access structure the parties follow the following protocol [11].

Protocol 3.5: $[s]^{\mathbb{F}} \leftarrow \mathsf{RSStoShamir}([s]^{RQ})$

1 **foreach party** $i \in [1..n]$ **do**

$$s_i \leftarrow \sum_{j=1, i \notin X_j}^{|A|} \left(r_j \prod_{\ell \in X_j} \frac{(\ell - i)}{\ell} \right) ;$$

2 `/* Local computation */`
`/* All arithmetic is done in` $\mathbb{F}$ `*/`

3 **return** s_i;

Then $(s_1, s_2, \ldots, s_n)$ is the Shamir sharing $[s]^{\mathbb{F}}$.

Correctness: Consider the polynomial $f(x) = \sum_{i=1}^{|A|} r_i f_i(x)$, where $f_i(x) = \prod_{j \in X_i} \frac{(j-x)}{j}$. By construction, the polynomials $f_i(x)$ have degree t over $\mathbb{F}$; $f_i(0) = 1$; and $f_i(j) = 0$ for all j such that $j \in X_i$. Therefore, $f(x)$ is a polynomial of degree t and $f(0) = \sum_{i=1}^{|A|} r_i f_i(0) = s$. Thus, $f(x)$ forms a valid polynomial for the Shamir sharing of s. Also observe that $s_i = f(i)$ for all $i \in [1..n]$. Hence the vector $(s_1, s_2, \ldots, s_n)$ is the Shamir sharing $[s]^{\mathbb{F}}$. See [26, 15] for details.

Security: Security follows from the locality of conversion. The converted shares cannot leak more information than the original shares. If the original shares are random uniform then the resulting Shamir shares are also random uniform.

3.2.3 Non-Interactive Generation of RSS Shares

Observe that once the parties obtain the RSS sharing of a random field element, they can locally convert it to a Shamir sharing of the same value. Also note that if the parties mutually generate random consistent RSS shares, the resulting secret is also random. Furthermore, note that a consistent RSS sharing implies that several subsets of members are able to agree on a secret common field element (similar to a secret group key).

Since we want to generate several random field elements, we need to an equal number of independent and consistent random RSS sharings. Our approach is to somehow obtain one consistent random RSS sharing and reuse this sharing an arbitrary number of times. We call the initial sharing the *master sharing*. The idea is to use each RSS share r_i of the master sharing as the key to a pseudorandom function (PRF) H and then to generate the corresponding RSS share $r_{(i,j)}$ of jth subsequent instance as $H_{r_i}(j)$ [11, 15].

Although exponentially many random secret field elements can be generated based on a single master sharing, obtaining the master sharing could (and probably would) still require interaction. However, even if the cost of obtaining this master sharing may be high, it is amortized over time as it can be used to generate practically infinite subsequent random sharings without interaction.

(A) Generating The Master Sharing: There are several ways to generate the master sharing. One model is to use a trusted party to distribute the master sharing. Another method suggested in [15] does not use trusted parties. In this, each party acts as the dealer and distributes an RSS sharing of a secret random field element. Then the master sharing is the sum of these secrets (computed locally as the sum of the n RSS share vectors held

by each party). Here we suggest an alternate and more efficient method without using trusted parties. Observe that to generate the master sharing without a trusted party, we can use a key agreement protocol to generate secret keys shared among every subset of $n-t$ users. These secret keys then act as RSS shares held by those subsets. A suggested protocol based on the Diffie-Hellman key agreement to compute a secret shared key is given below.

(B) Key Agreement Protocol: Let g be a generator of a cyclic multiplicative group G of order prime p such that Computational Diffie-Hellman (CDH) problem in G is hard. User i generates $x_i \stackrel{R}{\leftarrow} \mathbb{Z}_p$ as the private key. The public key is $y_i = g^{x_i} \in G$. Some set S of users want to agree on a key.

Protocol 3.6: $k \leftarrow \mathsf{RandKey}(S)$

1 **foreach party** $i \in S$ **do parallel**
2 $r_i \stackrel{R}{\leftarrow} \mathbb{Z}_p$;
3 **foreach** $j \in S, j \neq i$ **do**
4 $z_{(i,j)} \leftarrow {y_j}^{r_i}$;
5 $\{z_{(i,j)} | j \in S, j \neq i\} \Rightarrow S$; // public information, broadcast
6 **foreach party** $i \in S$ **do**
7 $k \leftarrow g^{r_i}$;
8 **foreach** $j \in S, j \neq i$ **do**
9 $k \leftarrow k \cdot {z_{(j,i)}}^{1/x_i}$;
10 **return** k;

Protocol 3.7 generates the master sharing $[s_m]^{RQ}$ using Protocol 3.6.

Protocol 3.7: $[s_m]^{RQ} \leftarrow \mathsf{MasterRSS}(\mathbb{F})$

1 **foreach** $i \in [1..|A|]$ **do parallel**
2 $k_i \leftarrow \mathsf{RandKey}([1..n] \backslash X_i)$;
3 **foreach party** $j \in [1..n] \backslash X_i$ **do**
4 $r_i \leftarrow \mathcal{H}(k_i)$; // $\mathcal{H} : G \mapsto \mathbb{F}$ is a hash function.
5 **return** r_i;

Then the vector $(r_1, r_2, \ldots, r_{|A|})$ is a valid RSS sharing of a random field element s_m.

Security: The security of Protocol 3.7 is based on the hardness of the CDH problem.

Complexity: The cost of Protocol 3.7 is $\binom{n}{t}(n-t)$ parallel broadcasts, each of size $\Theta(n-t-1)$. On an average, this is more efficient than the method suggested in [15] for generating the initial sharing assuming broadcast channels.

Note that the overhead in the above protocol is only due to Step 2, where RandKey() is invoked. Therefore, a non-interactive key agreement protocol for an arbitrary group of users will allow non-interactive generation of the master sharing and will consequently allow true non-interactive generation of random field elements.

(C) Non-Interactive Generation of RSS Sharings: Let $[s_m]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$ be an RSS master sharing over some field $\mathbb{F}'$. Let $H : \mathbb{F}' \times \mathbb{Z}^+ \times \mapsto \mathbb{F}$ be a PRF with keys in $\mathbb{F}'$, outputs in $\mathbb{F}$, and inputs in $\mathbb{Z}^+$. For any $i \in \mathbb{Z}^+$, by $[H_{[s_m]}(i)]^{RQ}$, we denote the RSS sharing $(H_{r_1}(i), H_{r_2}(i), \ldots, H_{r_{|A|}}(i))$. Protocol 3.8 non-interactively generates a

random RSS sharing $[s]^{RQ}$.

Protocol 3.8: $[s]^{RQ} \leftarrow \mathsf{RandRSS}(\mathbb{F})$

```
1 static ctr ← 0;                          // static used as in C language
2 static [s_m]^RQ ← MasterRSS(F');          // F' is independent of F
3 [s]^RQ ← [H_[s_m](ctr)]^RQ;
4 ctr++;
5 return [s]^RQ;
```

Security: The security of the above protocol relies on the security of the PRF H [11].

Pseudorandom Replicated Secret sharing (PRSS): The general name given to this technique of generating secret random RSS-shared field elements using PRFs is PRSS.

3.2.4 PRSS-Based Protocols For Randoms

(A) Non-interactive shared random field elements: The following protocol outputs a Shamir-shared random element of $\mathbb{F}$ non-interactively:

Protocol 3.9: $[r]^{\mathbb{F}} \leftarrow \mathsf{PRandFld}(\mathbb{F})$

```
1 [r]^RQ ← RandRSS(F) ;
2 [r]^F ← RSStoShamir([r]^RQ);
3 return [r]^F;
```

Security: The above protocol is secure as long as s_m is never revealed and F satisfies the standard notions of security for PRFs.

Complexity: There is no communication involved. Regarding computation complexity, Step 1 requires each party to do $\binom{n-1}{t}$ PRF computations and Step 2 requires each party to compute the inner product of two $\binom{n-1}{t}$-vectors over $\mathbb{F}$, one of which is computed in Step 1 and the other can be precomputed even before the master sharing is obtained.

(B) Non Interactive Random Zero Sharing: The following protocol was proposed in [11]. It generates a random Shamir-sharing [0] using a polynomial of degree $2t < n$. The idea can be generalized to any degree.

Notation: As before, $A = \{X | X \subset [1..n] \wedge |X| = t\}$ is the set of all maximal unqualified subsets of parties for some threshold t, and $\{X_1, X_2, \ldots, X_{|A|}\}$ is an arbitrary labeling of elements of A. Let F be the set of polynomials over $\mathbb{F}$ of degree $\leq 2t$. Then F is a vector space over $\mathbb{F}$ of dimension $2t+1$. For every $i \in [1..|A|]$, define set $F_i \subset F$ as:

$$F_i = \{f | (f \in F) \wedge (f(0) = 0) \wedge (\forall j \in X_i : f(j) = 0)\}.$$

Then F_i is a subspace of F of dimension $(2t - (1+t)) + 1 = t$. Let $\{f_{i,1}, f_{i,2}, \ldots, f_{i,t}\}$ be any basis of F_i, which is public information. We describe below how to compute a basis. Let $H : \{0,1\}^\alpha \times \mathbb{Z}^+ \mapsto \mathbb{F}$ be a PRF with keys in $\{0,1\}^\alpha$, and let $H' : \mathbb{F}' \times \mathbb{Z}^+ \mapsto \{0,1\}^\alpha$ be a PRF with keys in $\mathbb{F}'$.

Protocol 3.10 generates a random Shamir sharing of 0 without interaction.

Protocol 3.10: $[z]^{\mathbb{F}} \leftarrow \mathsf{PRandZero}(\mathbb{F})$

1 **static** $ctr \leftarrow 0$;
2 **static** $[s]^{RQ} \leftarrow \mathsf{RandRSS}(\mathbb{F}')$; `// 𝔽′ is independent of 𝔽`
`// Let` $[s]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$, `the RSS shares`
3 **foreach** $i \in [1..|A|]$ **do**
4 **foreach party** $j \in X_i$ **do**
5 **foreach** $\ell \in [1..t]$ **do**
6 **static** $r_{i,\ell} \leftarrow H'_{r_i}(\ell)$; `//` $r_{i,\ell}$ `act like keys to a PRF`
7 **foreach party** $j \in [1..n]$ **do**
8 $z_j \leftarrow \sum_{i=1, j \notin X_i}^{|A|} \left(\sum_{\ell=1}^{t} H_{r_{i,\ell}}(ctr) \cdot f_{i,\ell}(j)\right)$;
9 **return** z_j;
10 ctr++;

Correctness: Consider the polynomial $f_0 = \sum_{i=1, j \notin X_i}^{|A|} \left(\sum_{\ell=1}^{t} H_{r_{i,\ell}}(ctr) \cdot f_{i,\ell}\right)$. Then $deg(f_0) \leq 2t$; $z_j = f(j)$ for each $j \in [0..n]$; and $f_0(0) = 0$. Thus, $(z_1, z_2, \ldots, z_n)$ is a consistent Shamir sharing of 0 using polynomial f_0 of degree $\leq 2t$.

Security: Recall that the security of a shared secret depends on the secrecy of the reconstruction polynomial. In this case, there is no 'secret'. However, it still makes sense to talk about the secrecy of the reconstruction polynomial. Observe that the inner sum in Step 8 generates a Shamir sharing $[0]$ using a polynomial of degree $\leq 2t$ such that the shares of parties in X_i are 0 and the other shares are random. This can be viewed as a $(2t, n)$ Shamir sharing where t shares have been revealed. Therefore, even if this polynomial has degree $2t$, only $t+1$ more shares are needed to reconstruct it. The outer sum (of all such $[0]$ sharings) results in a Shamir sharing $[0]$ using a polynomial of degree $\leq 2t$ where the share of every party is random. Consequently, $2t+1$ shares are needed to reconstruct this final polynomial.

Selecting a basis: For each $i \in [1..|A|]$ and each $\ell \in [1..t]$, consider the polynomial:

$$f_{i,\ell} = x^{\ell} \prod_{j \in X_i} (x - j)$$

By construction $f_i = \{f_{i,1}, f_{i,2}, \ldots, f_{i,t}\} \subset F_i$; each element of f_i is linearly independent of the others (because they are of different degree); and $|f_i| = t$. Thus, f_i is a basis of F_i.

(C) Multiplication of secrets with public output: Each party $i \in [1..n]$ holds a_i, b_i, the shares of $a, b \in \mathbb{F}$. Assume that $(z_1, z_2, \ldots, z_n)$ is a random Shamir sharing of 0. They follow Protocol 3.11:

Protocol 3.11: $c \leftarrow \mathsf{MulPub}([a]^{\mathbb{F}}, [b]^{\mathbb{F}})$

1 $[z]^{\mathbb{F}} \leftarrow \mathsf{PRandZero}(\mathbb{F})$; // Let $[z]^{\mathbb{F}} = (z_1, z_2, \ldots, z_n)$, the Shamir shares
2 **foreach party** $i \in [1..2t+1]$ **do parallel**
3 $d_i \leftarrow a_i b_i + z_i$;
4 $d_i \rightarrow [1..n]$; // 1 rnd, ≈1 inv ($\mathbb{F}$)
5 **foreach party** $j \in [1..n]$ **do**
6 $c \leftarrow \sum_{i=1}^{2t+1} \left(d_i \prod_{\ell=1, \ell \neq i}^{2t+1} \frac{-\ell}{i-\ell} \right)$;
7 **return** c;

Correctness: We refer the reader to [26, 11] for the correctness.

Security: Observe that $a_i b_i$ is the share of party i of a Shamir sharing $[ab]$ using a polynomial of degree $\leq 2t$, and therefore, $2t+1$ such shares are needed to reconstruct the polynomial (and then compute ab). However, the distribution of this polynomial is no longer uniform and so the revealed shares may leak some information about the original polynomials using which a, b were shared. In order to avoid this, we add shares of a random zero sharing $[0]$ to the original shares. In this case, the resulting shares d_i still represent the sharing $[ab]$ but whose polynomial is now uniformly distributed.

(D) Computing the Inverse of Secret Non-Zero Field Elements: Protocol 3.12 uses PRSS to implement a variant of a protocol presented in [2] to compute the Shamir sharing $[a^{-1}]$ given the sharing $[a]$ for some $a \in \mathbb{F} \backslash \{0\}$. It is necessary to ensure that $a \neq 0$, before invoking the protocol, otherwise this information is revealed.

Protocol 3.12: $[b]^{\mathbb{F}} \leftarrow \mathsf{Inv}([a]^{\mathbb{F}})$

1 $[r]^{\mathbb{F}} \leftarrow \mathsf{PRandFld}(\mathbb{F})$;
2 $x \leftarrow \mathsf{MulPub}([r]^{\mathbb{F}}, [a]^{\mathbb{F}})$; // 1 rnd, 1 inv ($\mathbb{F}$)
3 $y \leftarrow x^{-1}$;
4 $[b]^{\mathbb{F}} \leftarrow y[r]^{\mathbb{F}}$;
5 **return** $[b]^{\mathbb{F}}$;

In a real implementation, there should be a check after Step 2 to ensure that $x \neq 0$. If $x = 0$ then the protocol must be started from the beginning.

Security: Protocol 3.12 provides perfect privacy.

(E) Shared random bits Protocol 3.13 generates a secret random bit shared in $\mathbb{Z}_q$ using PRandFld. The complexity of the protocol is 1 round and 1 invocations (2 rounds and 2 invocations less than RandBit). This protocol works only in $\mathbb{Z}_q$.

Protocol 3.13: $[b] \leftarrow \mathsf{PRandBit}(q)$

// Requires $q \equiv 3 \pmod 4$
1 $[r] \leftarrow \mathsf{PRandFld}(\mathbb{Z}_q)$;
2 $u \leftarrow \mathsf{MulPub}([r], [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
3 $v \leftarrow u^{-\frac{q+1}{4}} \bmod q$; // 1 exp ($\mathbb{Z}_q$)
4 $[b] \leftarrow (v[r] + 1)(2^{-1} \bmod q)$;
5 **return** $[b]$;

Security: Protocol 3.13 provides perfect privacy.

3.3 Protocols Based On RISS

In this section we describe protocols for efficiently generating sharings in two fields of the same random bit. We do this using a share conversion protocol described in [15]. The protocol can be used for converting a Shamir sharing of a bit in a source field to the Shamir sharing of the same bit in a different target field.

We first describe another secret sharing scheme called RISS. This is derived from RSS where the computation is over the set of integers $\mathbb{Z}$ instead of the finite field $\mathbb{F}$. The difference between the two from an application point of view is that while RSS allows us to efficiently generate random field elements non-interactively, RISS allows us to convert bits Shamir-shared in one field to that in another field. RISS is described below.

3.3.1 Replicated Integer Secret Sharing (RISS)

RISS is very similar to RSS with the only difference that arithmetic is done over the ring of integers rather than a finite field. In the following, κ is a security parameter.

Original RISS: For clarity of presentation, we first describe a variant of RISS discussed in [15]. Our actual variant of RISS is slightly different from their's but the main ideas carry over.

1. *Share Generation:* To share a secret $s \in [-2^{\ell}..2^{\ell}]$, the dealer generates $r_i \xleftarrow{R} [-2^{\ell+\kappa}..2^{\ell+\kappa}]$ for $1 \leq i \leq |A|$ and sets $\theta \leftarrow (\sum_{i=1}^{|A|} r_i) - s$. Then $[s]^{RZ} \stackrel{\text{def}}{=} (r_1, r_2, \ldots, r_{|A|})$ is the RISS share vector of s. Note that $-(n2^{\ell+\kappa} + 2^{\ell}) \leq \theta \leq n2^{\ell+\kappa} + 2^{\ell}$. The pair $([s]^{RZ}, \theta)$ is called a RISS sharing of s.

2. *Share Distribution:* The dealer assigns some arbitrary labeling to elements of A and writes A as $\{X_1, X_2, \ldots X_{|A|}\}$. Then for $1 \leq i \leq |A|$, he distributes the shares as:

$$r_i \rightarrow [1..n] \backslash X_i$$

$$\theta \rightarrow [1..n]$$

3. *Secret Reconstruction:* Let $B \in [1..n]$ such that $|B| = t + 1$. Then members of B jointly share the entire vector $(r_1, r_2, \ldots, r_{|A|})$. They compute $s = (\sum_{i=1}^{|A|} r_i) - \theta$.

Security: The scheme provides statistical privacy. Specifically, let $B \in [1..n]$ be such that $|B| \leq t$. Let $s', s'' \xleftarrow{R} [-2^{\ell}..2^{\ell}]$ be two secrets and let s'_i, s''_i be their share vectors held by party i. Then $\Delta(\{s'_i | i \in B\}, \{s''_i | i \in B\}) \leq c/2^{\kappa}$ [11, 15, 14].

Modified RISS: In our application, RISS is not used to protect (i.e., share) any predefined secret but rather to generate random (RISS) shares in certain range, which would then correspond to some unknown secret value. Therefore, we consider a variation of RISS as follows:

1. The first difference is that we consider $\theta = 0$. The parties will obtain the RSS shares without a dealer, and these shares will correspond to some unknown secret.

2. The second difference is that we consider both the RISS shares and secrets to be unsigned integers instead of signed.

3.3.2 Conversion from RISS Shares to Shamir Shares

We describe a method similar to that in Section 3.2.2, where instead of inner product of two $\binom{n-1}{t}$-vectors over $\mathbb{F}$, we now take the $\mathbb{Z}$-module inner product $\mathbb{Z}^{\binom{n-1}{t}} \times \mathbb{F} \mapsto \mathbb{F}$. Specifically, let $([s]^{RZ})$ be a RISS sharing of secret $s \in \mathbb{Z}$ with $[s]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$. Then the share vector of each party $i \in [1..n]$ is $(r_j)_{i \notin X_j}$ with $\binom{n-1}{t}$ elements.

The following protocol converts the above RISS sharing to a Shamir sharing in $\mathbb{F}$.

Protocol 3.14: $[s']^{\mathbb{F}} \leftarrow \mathsf{RISStoShamir}([s]^{RZ}, \mathbb{F})$

foreach party $i \in [1..n]$ **do**

$$s_i \leftarrow \sum_{j=1, i \notin X_j}^{|A|} \left(r_j \prod_{\ell \in X_j} \frac{(\ell - i)}{\ell} \right) ;$$

`/* Local computation */`

`/* All arithmetic is done in` $\mathbb{F}$ `*/`

return s_i;

Then $(s_1, s_2, \ldots, s_n)$ is the Shamir sharing $[s']^{\mathbb{F}}$ for some $s' \in \mathbb{F}$ as follows [15]:

1. If $\mathbb{F} = \mathbb{Z}_q$, then $s' = s \bmod q$.
2. If $\mathbb{F} = \mathbb{F}_{2^\alpha}$, then $s' = s \bmod 2$.

Security: As in Protocol 3.5, security follows from the locality of conversion.

3.3.3 Non-Interactive Generation of RISS Shares

We can reuse a RSS master sharing discussed in Section 3.2.3 to non-interactively generate random RISS shared integers. In the following, $[s_m]^{RQ} = (r_1, r_2, \ldots, r_{|A|})$ is an RSS master sharing defined over some (large) finite field $\mathbb{F}$. Let $F^\alpha : \mathbb{F} \times \mathbb{Z}^+ \mapsto [0..2^\alpha - 1]$ be a PRF with keys in $\mathbb{F}$ for some parameter $\alpha \in \mathbb{Z}^+$. For any $i \in \mathbb{Z}$, by $[F^\alpha_{[s_m]}(i)]^{RZ}$, we denote the RISS sharing $(F^\alpha_{r_1}(i), F^\alpha_{r_2}(i), \ldots, F^\alpha_{r_{|A|}}(i)) \in [0, 2^\alpha - 1]^{|A|}$. The following protocol uses $[s_m]^{RQ}$ to non interactively generate a random RISS sharing $[s]^{RZ}$. The shares are integers in the range $[0..2^\alpha - 1]$.

Protocol 3.15: $[s]^{RZ} \leftarrow \mathsf{RandRISSshares}(\alpha)$

`/* Generates a RISS sharing of a secret with each share random in` $[0..2^\alpha - 1]$ `*/`

static $ctr \leftarrow 0$; `// static used as in C language`

static $[s_m]^{RQ} \leftarrow \mathsf{MasterRSS}(\mathbb{F})$; `//` $\mathbb{F}$ `is independent of` α

$[s]^{RZ} \leftarrow [F^\alpha_{[s_m]}(ctr)]^{RZ}$;

ctr++;

return $[s]^{RZ}$;

Then $[s]^{RQ}$ is a random sharing of some element $s \in [0..2^\alpha |A| - |A|]$.

Security: In the above variant with $\theta = 0$, the secret s may no longer be statistically protected. On the other hand, the α least significant bits of s are indeed statistically protected. Therefore $\mathbf{LSB}(s)$ is also statistically hidden. The following protocols base their security on the secrecy of this bit.

3.3.4 Bit-Share Conversions and Joint Bit Generation

We describe protocols for converting a Shamir sharing of a bit (or bounded integers) in a source field $\mathbb{F}$ to polynomial shares in another target field $\mathbb{F}'$. The protocols use a random RSS master sharing (via RandRISSshares) in some prime field $\mathbb{Z}_{q'}$. Note that this field may be different from any of the fields considered below. However, since elements of $\mathbb{Z}_{q'}$ will be used as secret keys to a PRF, it is necessary to have this field large enough to prevent a brute-force attack.

(A) Conversion of bit shares from $[b]^{\mathbb{Z}_q}$ to $[b]^{\mathbb{F}_{2^m}}$. Protocol 3.16 converts a shared bit from polynomial sharing in $\mathbb{Z}_q$ to polynomial sharing in $\mathbb{F}_{2^m}$.

Protocol 3.16: $[b]^{\mathbb{F}_{2^m}} \leftarrow \mathsf{BitZQtoF2M}([b]^{\mathbb{Z}_q}, m)$

1 $[r]^{RZ} \leftarrow \mathsf{RandRISSshares}(\kappa+1)$;
2 $[r]^{\mathbb{Z}_q} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{Z}_q)$;
3 $[r_0]^{\mathbb{F}_{2^m}} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{F}_{2^m})$;
4 $r' \leftarrow \mathsf{Output}([r]^{\mathbb{Z}_q} + [b]^{\mathbb{Z}_q})$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
5 $r'_0 \leftarrow \mathbf{LSB}(r')$;
6 $[b]^{\mathbb{F}_{2^m}} \leftarrow [r_0]^{\mathbb{F}_{2^m}} \oplus r'_0$;
7 **return** $[b]^{\mathbb{F}_{2^m}}$;

Correctness: Observe that $r'_0 = r_0 \oplus b$, hence the output is correct.
Security: Protocol 3.16 provides statistical privacy with security parameter κ.
Complexity: The complexity is one round and one invocation in $\mathbb{Z}_q$.

(B) Conversion of bit shares from $[b]^{\mathbb{Z}_{q_1}}$ to $[b]^{\mathbb{Z}_{q_2}}$, $q_2 > q_1$. Protocol 3.17 converts a shared bit from polynomial sharing in $\mathbb{Z}_{q_1}$ to polynomial sharing in $\mathbb{Z}_{q_2}$, where $q_2 > q_1$. For example, it can be used in order to carry out a binary computation in a smaller field and then convert the result to a larger field.

Protocol 3.17: $[b]^{\mathbb{Z}_{q_2}} \leftarrow \mathsf{BitZQtoZQ}([b]^{\mathbb{Z}_{q_1}}, q_2)$

1 $[r]^{RZ} \leftarrow \mathsf{RandRISSshares}(\kappa+1)$;
2 $[r]^{\mathbb{Z}_{q_1}} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{Z}_{q_1})$;
3 $[r]^{\mathbb{Z}_{q_2}} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{Z}_{q_2})$;
4 $r' \leftarrow \mathsf{output}([r]^{\mathbb{Z}_{q_1}} + [b]^{\mathbb{Z}_{q_1}})$; // 1 rnd, 1 inv ($\mathbb{Z}_{q_1}$)
5 $[b]^{\mathbb{Z}_{q_2}} \leftarrow r' - [r]^{\mathbb{Z}_{q_2}}$;
6 **return** $[b]^{\mathbb{Z}_{q_2}}$;

Security: Protocol 3.17 provides statistical privacy with security parameter κ.
Complexity: The complexity is one round, one invocation in a small field $\mathbb{Z}_{q_1}$.

(C) Joint generation of a shared random bit in $\mathbb{Z}_q$ for large q. Protocol 3.18 allows to efficiently generate shared random bits in $\mathbb{Z}_q$ when q is large. The protocol first generates a shared random bit in a small field $\mathbb{Z}_{q_1}$ and then converts the shares to the larger target field $\mathbb{Z}_q$, where $q \gg q_1$.

Protocol 3.18: $[b]^{\mathbb{Z}_q} \leftarrow \mathsf{PRandBitL}(q)$

1 $[b]^{\mathbb{Z}_{q_1}} \leftarrow \mathsf{PRandBit}(q_1)$; // 1 rnd, 1 inv ($\mathbb{Z}_{q_1}$)
2 $[b]^{\mathbb{Z}_q} \leftarrow \mathsf{BitZQtoZQ}([b]^{\mathbb{Z}_{q_1}}, q)$; // 1 rnd, 1 inv ($\mathbb{Z}_{q_1}$)
3 **return** $[b]^{\mathbb{Z}_q}$;

Security: Protocol 3.18 provides statistical privacy with security parameter κ.

Complexity: The complexity of protocol 3.18 is 2 rounds and 2 invocations in $\mathbb{Z}_{q_1}$. Working in the small field $\mathbb{Z}_{q_1}$ reduces the computation complexity (Step 3 in PRandBit, especially) and the communication complexity. For $q \gg q_1$ the overall efficiency gain is substantial, especially when generating large batches of shared random bits. Taking $\log_2(q_1) = 64$ bits is sufficient for our purposes. Moreover, we can also use the bits shared in $\mathbb{Z}_{q_1}$ to improve the efficiency of binary computation (instead of bits shared in $\mathbb{F}_{2^8}$).

(D) Joint generation of a random bit shared in both $\mathbb{F}_{2^m}$ and $\mathbb{Z}_q$ with large q. The purpose of protocol 3.19 is to improve the efficiency of a family of protocols presented in Chapter 5 (e.g., truncation, comparison, bit decomposition). It combines the protocols 3.18 and 3.16 in order to efficiently generate a double sharing of a random bit b. The sharing $[b]^{\mathbb{F}_{2^m}}$ is used for efficient binary computation, while the sharing $[b]^{\mathbb{Z}_q}$ is used for integer computation (see protocol 5.2).

Protocol 3.19: $([b]^{\mathbb{F}_{2^m}}, [b]^{\mathbb{Z}_q}) \leftarrow \mathsf{PRandBitD}(m, q)$

/* select small prime q_1 with $q_1 \ll q$ */
1 $[b]^{\mathbb{Z}_{q_1}} \leftarrow \mathsf{PRandBit}(q_1)$; // 1 rnd, 1 inv, 1 exp ($\mathbb{Z}_{q_1}$)
2 $[r]^{RZ} \leftarrow \mathsf{RandRISSshares}(\kappa + 1)$;
3 $[r]^{\mathbb{Z}_{q_1}} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{Z}_{q_1})$;
4 $[r_0]^{\mathbb{F}_{2^m}} \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{F}_{2^m})$;
5 $r' \leftarrow \mathsf{output}([r]^{\mathbb{Z}_{q_1}} + [b]^{\mathbb{Z}_{q_1}})$; // 1 rnd, 1 inv ($\mathbb{Z}_{q_1}$)
6 $r'_0 \leftarrow \mathbf{LSB}(r')$;
7 $[b]^{\mathbb{F}_{2^m}} \leftarrow [r_0]^{\mathbb{F}_{2^m}} \oplus r'_0$;
8 $[b]^{\mathbb{Z}_q} \leftarrow r' - [r]^{\mathbb{Z}_q}$;
9 **return** $([b]^{\mathbb{F}_{2^m}}, [b]^{\mathbb{Z}_q})$;

Security: Protocol 3.19 provides statistical privacy with security parameter κ.

Complexity: Protocol 3.19 has the same complexity as protocol 3.18, i.e., 2 rounds and 2 invocations in $\mathbb{Z}_{q_1}$ (the additional conversion is essentially for free).

(E) Conversion of bit shares from $[b]^{\mathbb{F}_{2^m}}$ to $[b]^{\mathbb{Z}_q}$. Protocol 3.20 converts a shared bit from polynomial sharing in $\mathbb{F}_{2^m}$ to polynomial sharing in $\mathbb{Z}_q$.

Protocol 3.20: $[b]^{\mathbb{Z}_q} \leftarrow \mathsf{BitF2MtoZQ}([b]^{\mathbb{F}_{2^m}}, q)$

1 $([b']^{\mathbb{F}_{2^m}}, [b']^{\mathbb{Z}_q}) \leftarrow \mathsf{PRandBitD}(m, q)$; // 2 rnd, 2 inv, 1 exp ($\mathbb{Z}_{q_1}$)
2 $b'' \leftarrow \mathsf{output}([b]^{\mathbb{F}_{2^m}} \oplus [b']^{\mathbb{F}_{2^m}})$; // 1 rnd, 1 inv ($\mathbb{F}_{2^m}$)
3 $[b]^{\mathbb{Z}_q} \leftarrow b'' + [b']^{\mathbb{Z}_q} - 2b''[b']^{\mathbb{Z}_q}$; // $[b]^{\mathbb{Z}_q} \leftarrow [b']^{\mathbb{Z}_q} \oplus b''$
4 **return** $([b]^{\mathbb{Z}_q})$;

Security: Protocol 3.20 provides statistical privacy with security parameter κ.

Complexity: The complexity is 3 rounds and 3 invocations in small fields. However,

in typical applications, the random bit b' shared in $\mathbb{F}_{2^m}$ and $\mathbb{Z}_q$ can be precomputed (see protocol 5.2). In this case, Step 1 is not necessary, and the conversion takes one round and one invocation in $\mathbb{F}_{2^m}$. The modified protocol takes as input these random bits and has the interface BitF2MtoZQPre$([b]^{\mathbb{F}_{2^m}}, [b']^{\mathbb{F}_{2^m}}, [b']^{\mathbb{Z}_q})$.

3.3.5 Generation of Shared Randoms in Range Using RISS

In this section we describe RISS-based protocols for non-interactively generating random Shamir-shared field elements of $\mathbb{Z}_q$ in a sub-range with some distribution. As before, n are the number of parties and $t < n$ is the number of maximum corrupted parties.

(A) Secret Integers in range $[0..\binom{n}{t}(2^m-1)]$. The following protocol generates using RISS a Shamir sharing $[r]$ of a random integer $r \in [0..\binom{n}{t}(2^m-1)]$. The protocol requires that $q > \binom{n}{t}2^m$ to ensure that no wraparound modulo q occurs in the sum of shares.

Protocol 3.21: $[r] \leftarrow$ PRandInt(q, m)

1 $[r]^{RZ} \leftarrow$ RandRISSshares(m);
2 $[r] \leftarrow$ RISStoShamir$([r]^{RZ}, \mathbb{Z}_q)$;
3 **return** $[r]$;

Security: Protocol 3.21 provides statistical security. This follows from the locality of computation.

(B) Shared Integers in range $[0..2^m-1]$. The protocols of this section generate Shamir-shared random numbers in range $[0..2^m-1]$ assuming that $q > 2^m \gg \binom{n}{t}$.

Notation: Let $\nu = \binom{n}{t}$ and let $\tau : \mathbb{Z}_\nu \times [1..\nu] \mapsto \{0,1\}$ be a public function s.t. $\forall x \in \mathbb{Z}_\nu : \sum_{i=1}^{\nu} \tau(x,i) = x$. Recall that $A = \{X_1, X_2, \ldots X_\nu\}$ is the set of all maximal unqualified subsets of parties. In the following, $[s_m]^{RQ} = (r_1, r_2, \ldots, r_\nu)$ is an RSS master sharing defined over a large finite field $\mathbb{F}$ s.t. parties $[1..n]\backslash X_i$ know r_i. Let $F^\beta : \mathbb{F} \times \mathbb{Z}^+ \mapsto [0..\beta]$ be a PRF with keys in $\mathbb{F}$ for some parameter $\beta \in \mathbb{Z}^+$.

The following protocol produces random RISS shares $(r'_1, r'_2, \ldots, r'_\nu)$ of some random element $s \in [0, z]$ for a $z \in \mathbb{Z}^+$.

Protocol 3.22: $[s]^{RZ} \leftarrow$ RandRISSrange(z)

/* Generates a RISS sharing of a secret in $[0..z]$ */
1 **static** $ctr \leftarrow 0$; // static used as in C language
2 **static** $[s_m]^{RQ} \leftarrow$ MasterRSS$(\mathbb{F})$; // $\mathbb{F}$ is independent of z
// Let $[s_m] = (r_1, r_2, \ldots, r_\nu)$, the RSS shares of s_m
3 **foreach party** $j \in [1..n]$ **do**
4 **foreach** $\ell \in [1..\nu]$ **do**
5 **if** $j \notin X_\ell$ **then**
6 $\beta \leftarrow \lfloor z/\nu \rfloor + \tau(z \bmod \nu, \ell)$;
7 $r'_\ell \leftarrow F^\beta_{r_\ell}(i)$;
8 **return** r'_ℓ;
9 ctr++;

Let $[s]^{RZ} = (r'_1, r'_2, \ldots, r'_\nu)$. Then $[s]^{RZ}$ is a random consistent RISS sharing of some element $s \in [0..z]$. Note that RandRISSrange$(\binom{n}{t}(2^\alpha-1))$ is equivalent to RandRISSshares(α).

The following protocol generates Shamir-shared random elements in range $[0..2^m - 1]$ non-interactively. The protocol provides some security only if $2^m \gg \nu$. Note that the protocol works for any arbitrary upper bound instead of just $2^m - 1$.

Protocol 3.23: $[r] \leftarrow \mathsf{PRand2mN}(q, m)$

1 $[r]^{RZ} \leftarrow \mathsf{RandRISSrange}(2^m - 1)$;
2 $[r] \leftarrow \mathsf{RISStoShamir}([r]^{RZ}, \mathbb{Z}_q)$;
3 **return** $[r]$;

Correctness: Correctness follows from the fact that $q > 2^m$. Since the original RISS secret is $s \in [0..2^m - 1]$, the converted Shamir-shared secret is $r \bmod q = r$.

Security: Since the secret is the sum of $\binom{n}{y}$ uniform shares, the protocol may not provide statistical security of the entire secret. However, $\log_2(\lfloor 2^m / \binom{n}{t} \rfloor)$ low-order bits of r are guaranteed to be secure. It is possible that the security provided may be adequate for certain applications (such as truncation in fixed-point multiplication - see Section 5.4.1).

The following protocol uses PRSS to generate uniform randoms in range $[0..2^m - 1]$.

Protocol 3.24: $[r], [r]_B \leftarrow \mathsf{PRand2mU}(q, m)$

1 **foreach** $i \in [0..m-1]$ **do parallel**
2 $\quad [r_i] \leftarrow \mathsf{PRandBit}(q)$; `// 1 rnd,` m `inv,` m `exp` $(\mathbb{Z}_q)$
3 $[r] \leftarrow \sum_{i=0}^{m-1} 2^i [r_i]$;
4 $[r]_B \leftarrow ([r_0], [r_1], \ldots, [r_{m-1}])$;
5 **return** $[r], [r]_B$;

Security: Protocol 3.24 provides perfect security.

Protocol	Distribution of output	Security	Complexity	Output
$\mathsf{Rand2mU}(q, m)$	uniform in $[0..2^m - 1]$	perfect	3 rounds $3m$ invocations	random with bits
$\mathsf{PRand2mU}(q, m)$	uniform in $[0..2^m - 1]$	perfect	1 round m invocations	random with bits
$\mathsf{PRandInt}(q, m)$	sum of $\binom{n}{t}$ uniform variables in $[0..2^m - 1]$	statistical	non-interactive	random only
$\mathsf{Rand2mN}(q, m)$	$r \in [0..2^m - 1]$, sum of n uniform variables	statistical	1 round 1 invocation	random only
$\mathsf{PRand2mN}(q, m)$	$r \in [0..2^m - 1]$, sum of $\binom{n}{t}$ uniform variables	statistical	non-interactive	random only

Table 3: Comparison of protocols for generating shared randoms in range.

3.4 Summary

This chapter described protocols for generating (Shamir-shared) random secrets. The three types of secrets we need are: (1) random field elements, (2) random bits, and (3) random numbers in range. We presented an RSS-based method for generating uniformly random field elements non-interactively (assuming that interaction for generating a master RSS sharing is ignored). We described RISS-based methods for converting Shamir-

sharings of bits from one field to another and for generating the Shamir sharing of the same uniformly random bit in two different fields. Finally, we described a RISS-based method for non-interactively generating normally distributed random field elements in a range that hides the low-order bits of the secret. The main bottleneck in our framework are protocols for the generation of a Shamir-shared random bit in a prime field. As of now, the problem of non-interactively generating such random bits is still open.

Complexity of The Protocols: Table 4 summarizes the complexity of the protocols discussed in this chapter:

Output	Protocol Interface	Field	Rounds	Invoc.	Exp.
$[r]^{\mathbb{F}}$	RandFld($\mathbb{F}$)	$\mathbb{F}$	1	1	-
$[r]^{\mathbb{F}}$	PRandFld($\mathbb{F}$)		0^*	0	-
$[r]^{\mathbb{Z}_q}$	RandBit(q)	$\mathbb{Z}_q$	3	3	1
$[b]^{\mathbb{Z}_q}$	PRandBit(q)	$\mathbb{Z}_q$	1^*	1	1
$[b]^{\mathbb{Z}_q}$	PRandBitL(q)†	$\mathbb{Z}_{q_1}$	2^*	2	1
$([b]^{\mathbb{F}_{2^m}}, [b]^{\mathbb{Z}_q})$	PRandBitD(m, q)†	$\mathbb{Z}_{q_1}$	2^*	2	1
$([r], [r]_B)$	Rand2mU(q, m)	$\mathbb{Z}_q$	3	$3m$	m
$([r], [r]_B)$	PRand2mU(q, m)	$\mathbb{Z}_q$	1	m	m
$[r]$	Rand2mN(q, m)	$\mathbb{Z}_q$	1	1	-
$[r]$	PRand2mN(q, m)		0^*	0	-
$[r]$	PRandInt(q, m)		0^*	0	-
$[b]^{\mathbb{F}_{2^m}}$	BitZQtoF2M($[b]^{\mathbb{Z}_q}, m$)	$\mathbb{Z}_q$	1^*	1	-
$[b]^{\mathbb{Z}_{q_2}}$	BitZQtoZQ($[b]^{\mathbb{Z}_{q_1}}, q_2$)†	$\mathbb{Z}_{q_1}$	1^*	1	-
$[b]^{\mathbb{Z}_q}$	BitF2MtoZQ($[b]^{\mathbb{F}_{2^m}}, q$)†	$\mathbb{F}_{2^m}$ $\mathbb{Z}_{q_1}$	1^* 2^*	1 2	- 1
$[b]^{\mathbb{Z}_q}$	BitF2MtoZQPre($[b]^{\mathbb{F}_{2^m}}, [b']^{\mathbb{F}_{2^m}}, [b']^{\mathbb{Z}_q}$)	$\mathbb{F}_{2^m}$	1^*	1	-
$[c]^{\mathbb{F}}$	MulPub($[a]^{\mathbb{F}}, [b]^{\mathbb{F}}$)	$\mathbb{F}$	1^*	1	-
$[b]^{\mathbb{F}}$	Inv($[a]^{\mathbb{F}}$)	$\mathbb{F}$	1^*	1	-

† Indicates that $q_1 \ll q$.
* Indicates that a master RSS sharing is needed.

Table 4: Complexity of main protocols discussed in Chapter 3.

4 k-ary, Prefix and Bit-Wise Operations

Outline. Deliverable D9.1 presents protocols proposed in the literature [13, 30] for computing unbounded fan-in boolean functions, prefixes, and operations with bitwise-shared integers. These protocols run in constant number of rounds, but their communication complexity and sometimes even round complexity are high (e.g., prefix-OR and bitwise comparison based on prefix-OR).

In applications that perform many parallel operations (like linear programming), protocols with lower communication complexity can offer better performance. We can even trade off a few additional rounds for an important reduction of the number of invocations.

We use the term k-ary to refer to an unbounded fan-in operation. We present generic protocols for k-ary and prefix operations in Section 4.1, followed by addition and comparison of bitwise-shared values in Section 4.2.

4.1 K-ary and Prefix Operations

We present in this section several protocols that perform k-ary and prefix operations with low communication complexity. We give general protocols that work for any associative binary operation, with secret inputs and outputs encoded in a finite field $\mathbb{F}$. These protocols are general building blocks used in secure comparison, fixed-point arithmetic, and other applications.

k-ary and prefix operators. Let $\mathcal{A}$ be a set and $\odot : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ an associative binary operator. We denote $[a] \odot [b]$ a secure evaluation of $a \odot b$ with secret inputs and secret output. We consider the following secure computation tasks (extensions of a binary operation):

- k-ary operation: $[p] = [a_1] \odot \ldots \odot [a_k] = \bigodot_{i=1}^{k} [a_i]$.
- Prefix operation: $([p_1], \ldots, [p_k]) = \mathsf{pre}_{\odot}([a_1], \ldots, [a_k])$, $[p_j] = \bigodot_{i=1}^{j} [a_i]$, $1 \leq j \leq k$.

In particular, we are interested in k-ary and prefix operations for multiplication (in $\mathbb{Z}$, $\mathbb{Z}_q$, $\mathbb{F}_{2^m}$) and boolean functions (OR, AND, XOR). Secure evaluation of these binary operations takes one round and one secure multiplication. A protocol that evaluates k-ary and prefix operations in the naive way as a sequence of binary operations needs $k-1$ rounds and $k-1$ secure multiplications. The communication complexity (i.e., number of invocations) in this solution is optimal, but the performance is low due to the large number of rounds.

Assuming that one evaluation $[c] \leftarrow [a] \odot [b]$ takes α rounds and β invocations in some field $\mathbb{F}$, we present three generic and efficient protocols for evaluating k-ary and prefix operations in $O(\log(k))\alpha$ rounds. The protocols can be used for any associative binary operation. They provide perfect privacy if the binary operation is evaluated with perfect privacy. The protocols are based on well known techniques from computer arithmetic [17] and parallel algorithms [19]. The protocols are specified assuming that k is a power of 2, but can easily be adapted to any k. The protocols assume that computation in done in some field $\mathbb{F}$.

4.1.1 k-ary Operations in $\log(k)$ Rounds

Protocol 4.1 computes $[p] = \bigodot_{i=1}^{k}[a_i]$. The principle is illustrated in Figure 1. The complexity of this protocol is $\alpha\log(k)$ rounds and $\beta(k-1)$ invocations. The communication complexity is optimal.

Protocol 4.1: $[p] \leftarrow \mathsf{KOpL}(\odot, [a_1], \ldots, [a_k])$

if $k > 1$ **then**
  foreach $i \in [1..k/2]$ **do parallel**
    $[u_i] \leftarrow [a_{2i}] \odot [a_{2i-1}]$; // α rnd, $\beta(\frac{k}{2}-1)$ inv ($\mathbb{F}$)
  $[p] \leftarrow \mathsf{KOpL}(\odot, [u_1], \ldots, [u_{k/2}])$; // $\alpha\log_2(\frac{k}{2})$ rnd, $\beta\frac{k}{2}$ inv ($\mathbb{F}$)
else
  $[p] \leftarrow [a_1]$;
return $[p]$;

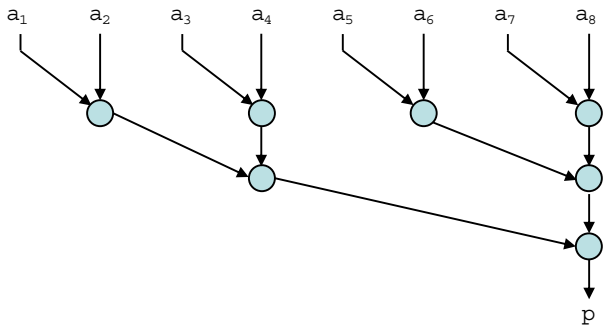

Figure 1: kOpL: k-ary operations in $\log(k)$ rounds with optimal communication complexity (example for $k = 8$).

4.1.2 Prefix operations in $O(\log(k))$ rounds.

A prefix protocol computes the prefixes $[p_j] = \odot_{i=1}^{j}[a_i]$, for $1 \le j \le k$. We describe two variants of the protocol with different tradeoffs. The first variant offers the best combinations of rounds and invocations.

Protocol 4.2 has lower number of rounds but more invocations. The principle is illustrated in Figure 2. The complexity of this protocol is $\alpha\log(k)$ rounds and $\beta\frac{k}{2}\log(k)$ invocations. For usual values of k, this variant offers a better trade-off between the number of rounds and the communication complexity.

Protocol 4.2: $([p_1], [p_2], \ldots, [p_k]) \leftarrow \mathsf{PreOpL}(\odot, [a_1], \ldots, [a_k])$

foreach $i \in [1..\log_2(k)]$ **do**
  foreach $j \in [1..k/2^i]$ **do parallel**
    $y \leftarrow 2^{i-1} + j \cdot 2^i$;
    foreach $z \in [1..2^{i-1}]$ **do parallel**
      $[a_{y+z}] \leftarrow [a_y] \odot [a_{y+z}]$; // $\alpha\log_2(k)$ rnd, $\beta(\frac{k}{2}\log_2(k))$ inv ($\mathbb{F}$)
return $([a_1], \ldots, [a_k])$;

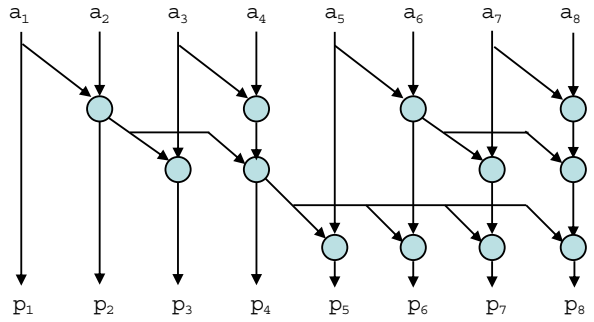

Figure 2: Prefix operations in $\log(k)$ rounds (example for $k = 8$).

Protocol 4.3 computes the prefixes $[p_j] = \odot_{i=1}^{j}[a_i]$, for $1 \leq j \leq k$ as in Protocol 4.2 but requires less invocations at the expense of more rounds. The principle is illustrated in Figure 3. The complexity of protocol 4.3 is $\alpha(2\log(k)-1)$ rounds and $\beta(2k-\log(k)-2)$ invocations. The number of invocations is optimal for $O(log(k))$ rounds.

Protocol 4.3: $([p_1],[p_2],\ldots,[p_k]) \leftarrow \mathsf{PreOpL2}(\odot,[a_1],\ldots,[a_k])$

1 $[p_1] \leftarrow [a_1]$;
2 **if** $k > 1$ **then**
3 **foreach** $i \in [1..k/2]$ **do parallel**
4 $[u_i] \leftarrow [a_{2i}] \odot [a_{2i-1}]$; // α rnd, $\beta(\frac{k}{2}-1)$ inv ($\mathbb{F}$)
5 $([v_1],\ldots,[v_{k/2}]) \leftarrow \mathsf{PreOpL2}(\odot,[u_1],\ldots,[u_{k/2}])$;
 // $\alpha(2\log_2(\frac{k}{2})-1)$ rnd, $\beta(k-\log_2(k))$ inv ($\mathbb{F}$)
6 **foreach** $i \in [1..k/2]$ **do**
7 $[p_{2i}] \leftarrow [v_i]$;
8 **foreach** $i \in [2..k/2]$ **do parallel**
9 $[p_{2i-1}] \leftarrow [a_{2i-1}] \odot [v_{i-1}]$; // α rnd, $\beta(\frac{k}{2}-1)$ inv ($\mathbb{F}$)
10 **return** $([p_1],\ldots,[p_k])$;

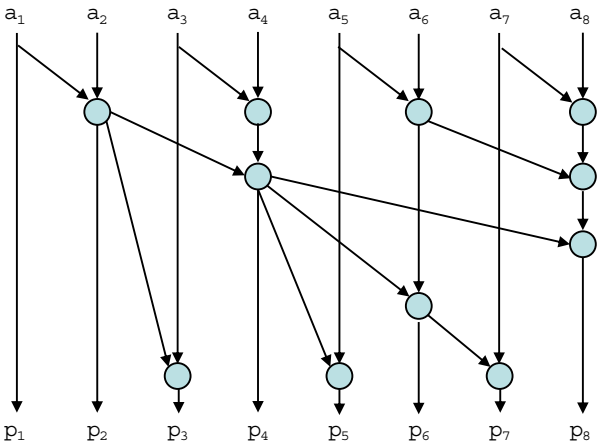

Figure 3: Prefix operations in $2\log(k)$ rounds with optimal communication complexity for $O(\log(k))$ rounds (example for $k = 8$).

4.1.3 Summary

Security: All the protocols in this section provide perfect privacy if the operation $\odot$ is evaluated with perfect privacy (e.g., boolean functions and multiplication).

Complexity: Table 5 summarizes the complexity of the protocols discussed in this section.

Protocol (k inputs)	Rounds	Invocations ($\mathbb{F}$)
KOpL	$\alpha \log(k)$	$\beta(k-1)$
PreOpL	$\alpha \log(k)$	$\beta(\frac{k}{2}\log(k))$
PreOpL2	$\alpha(2\log(k)-1)$	$\beta(2k-\log(k)-2)$

Table 5: Complexity of the protocols for k-ary and prefix operations.

Main k-ary and prefix operations: In our case, $\odot \in \{\mathsf{Mul}, \mathsf{AND}, \mathsf{OR}, \mathsf{XOR}, \circ\}$.[6] Each of these operations take one round and one invocations. Recall that α, β is the number of rounds and invocations for one computation of $\odot$. For convenience, we reproduce the values for relevant operations in Table 6.

$\odot$ (operation)	Field	α	β
Mul	any	1	1
AND	any	1	1
OR	any	1	1
XOR	any	1	1
XOR	$\mathbb{F}_{2^m}$	0	0
$\circ$	any	1	2

Table 6: Parameters α, β for the main k-ary and prefix operations.

Log rounds versus constant rounds protocols for k-ary and prefix operations: Constant rounds protocols for k-ary and prefix operations are presented in [13, 30] (see also D9.1 [26]). For example, k-ary OR can be computed in 6 rounds and $5k$ invocations, while kOpL runs in $\log(k)$ rounds and $k-1$ invocations. In this case, we trade off rounds versus communication and computation complexity. The complexity of the constant rounds variant can be reduced using PRSS. On the other hand, prefix-OR can be computed in 12 rounds and $12k + 5\sqrt{k}$ invocations. PreOpL runs in $\log(k)$ rounds with only $0.5k\log(k)$ invocations, so it needs less rounds and invocations even for very large k.

The constant rounds protocols can benefit from precomputation of the random values. However, online precomputation is not "for free", and low communication complexity remains a better option for applications that perform a very large number of parallel operations.

[6] The last one is the carry propagation operation - see Section 4.2.1.

4.2 Bitwise Operations

Outline. This section contains several protocols for operations with bitwise-shared values: binary addition, in Section 4.2.1, and comparison, in Section 4.2.2. These protocols are used as building blocks for secure comparison and arithmetic with fixed-point numbers. Protocols for addition and comparison of bitwise-shared integers were introduced in the deliverables D9.1 and D3.1. We present more efficient variants and include (for clarity and completeness) revised specifications and analysis for all the building blocks.

4.2.1 Binary Addition

The binary addition protocols presented in this section are based on standard algorithms from computer arithmetic [17], and use the general protocols for k-ary and prefix operations discussed in the previous section. These protocols run in $\log(k)$ rounds with low communication complexity and are more suitable for our applications than constant rounds solutions proposed in the literature [13, 22, 30].

Algorithm. Given the bitwise representations $(a_k, a_{k-1}, \ldots, a_1)$ and $(b_k, b_{k-1}, \ldots, b_1)$ of two k-bit numbers a and b respectively, we want to compute the bitwise representation $(s_{k+1}, s_k, \ldots s_1)$ of the $k+1$-bit number $s = a+b$. The protocols use the algorithm below.

For each $i \in [1...k]$, the algorithm takes as input three bits (a_i, b_i, cin_i) and outputs two bits (c_i, s_i), where cin_i, c_i denote the ith carry-in and carry bits respectively. The bits are assigned as follows: a_i, b_i are the input bits; $cin_1 = 0$; $cin_{i+1} = c_i$; and (c_i, s_i) are computed using Table 7. Finally s_{k+1} is set to c_k. Then $(s_{k+1}, s_k, \ldots s_1)$ is the bitwise representation of $a + b$.

a_i	b_i	cin_i	c_i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 7: Computing c_i, s_i

In other words, $s_i = a_i \oplus b_i \oplus cin_i$ and $c_i = ((a_i \oplus b_i) \wedge cin_i) \vee (a_i \wedge b_i)$. Rewriting the above, $s_i = a_i \oplus b_i \oplus c_{i-1}$ and $c_i = ((a_i \oplus b_i) \wedge c_{i-1}) \vee (a_i \wedge b_i)$ with $c_0 = 0$.

We call $a_i \wedge b_i$ the *carry generation bit* and $a_i \oplus b_i$ the *carry propagation bit*.

Computation of the carry bits c_i. Let $a = (a_k, \ldots, a_1)$ and $b = (b_k, \ldots, b_1)$ the two integer inputs, $p_i = a_i \oplus b_i$ the carry propagation bit and $g_i = a_i \wedge b_i$ the carry generation bit, for $1 \leq i \leq k$. The carry bits can be computed as follows:

$$c_1 = g_1, \;\; c_i = g_i \vee (p_i \wedge c_{i-1}), \;\; 2 \leq i \leq k.$$

We define the following carry propagation operator:

$$\circ : \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}^2$$

$$(p,g) = (p_2,g_2) \circ (p_1,g_1) = (p_1 \wedge p_2, g_2 \vee (p_2 \wedge g_1)).$$

For bits encoded in $\mathbb{Z}_q$ or $\mathbb{F}_{2^m}$ we obtain:

$$p_i = a_i + b_i - 2a_ib_i$$

$$g_i = a_ib_i$$

$$c_1 = g_1,\ c_i = g_i + p_ic_{i-1},\ 2 \le i \le k.$$

$$(p,g) = (p_2,g_2) \circ (p_1,g_1) = (p_1p_2, g_2 + p_2g_1).$$

For secret inputs and output, the operator $\circ$ can be computed in 1 round and 2 secure multiplications in parallel (i.e., 1 round, 2 invocations in the corresponding field).

Let $(P_1,G_1) = (p_1,g_1)$ and $(P_i,G_i) = (p_i,g_i) \circ (P_{i-1},G_{i-1})$, for $2 \le i \le k$. Observe that $c_i = 1$ if and only if $G_i = 1$. In other words, we can compute the carry bits c_i by computing the prefixes (P_i,G_i) and taking $c_i = G_i$, for $1 \le i \le k$. The carry propagation operator is associative, so the carry bits can be computed by adapting the generic prefix protocols 4.2 or 4.3.

Binary addition. Protocol 4.4 takes as inputs two bitwise-shared k-bit integers $[a]_B = ([a_k],\ldots,[a_1])$ and $[b]_B = ([b_k],\ldots,[b_1])$, and returns a bitwise-shared $k+1$-bit integer $[s]_B$ such that $s = a + b$. To simplify the description of the protocol we assume that k is a power of 2. It is easy to derive a general variant for any k.

Steps 2 requires 1 round and k bit multiplications. If one of the operands is public, this becomes local computation. Step 3 needs $\log(k)$ rounds and $\frac{k}{2}\log(k)$ evaluations of the carry propagation operator, i.e., $k\log(k)$ bit multiplications. The remaining steps are local computation.

Protocol 4.4: $[s]^{\mathbb{F}}_B \leftarrow$ AddBitwise$(([a_k]^{\mathbb{F}},\ldots,[a_1]^{\mathbb{F}}),([b^{\mathbb{F}}_k],\ldots,[b_1]^{\mathbb{F}}))$

1 **foreach** $i \in [1..k]$ **do parallel**
2 $\quad [d_i]^{\mathbb{F}}_B \leftarrow ([a_i]^{\mathbb{F}} + [b_i]^{\mathbb{F}} - 2[a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}}, [a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}})$; // 1 `rnd`, k `inv` $(\mathbb{F})$ // $d_i = (p_i,g_i)$
3 $([c_i]^{\mathbb{F}})_{i\in[1..k]} \leftarrow$ PreOpL$(\circ,[d_k]^{\mathbb{F}}_B,\ldots,[d_1]^{\mathbb{F}}_B)$; // $\log_2 k$ `rnd`, $k\log_2 k$ `inv` $(\mathbb{F})$
4 $[s_1]^{\mathbb{F}} \leftarrow [a_1]^{\mathbb{F}} + [b_1]^{\mathbb{F}} - 2[c_1]^{\mathbb{F}}$; // $c_1 = a_1b_1$
5 **foreach** $i \in [2..k]$ **do**
6 $\quad [s_i]^{\mathbb{F}} \leftarrow [a_i]^{\mathbb{F}} + [b_i]^{\mathbb{F}} + [c_{i-1}]^{\mathbb{F}} - 2[c_i]^{\mathbb{F}}$;
7 $[s_{k+1}]^{\mathbb{F}} \leftarrow [c_k]^{\mathbb{F}}$;
8 $[s]^{\mathbb{F}}_B \leftarrow ([s_{k+1}]^{\mathbb{F}},\ldots,[s_1]^{\mathbb{F}})$;
9 **return** $[s]^{\mathbb{F}}_B$;

Computation of the carry-out bit. In some applications (e.g., protocol 4.8) we need only the carry-out bit of a binary addition. This bit can be efficiently computed, without the result of the addition, as shown in Protocol 4.5, CarryOut. The protocol takes as inputs two bitwise-shared integers, $([a_k],\ldots,[a_1])$ and $([b_k],\ldots,[b_1])$, $a_i,b_i \in \{0,1\}$, and returns the carry bit c_k. Protocol 4.6 is an adaptation of the generic protocol 4.1 (kOpL).

The complexity of the protocol 4.6 is $\log(k)$ rounds and $k-1$ evaluations of the carry propagation operator, i.e., $2(k-1)$ bit multiplications. Protocol 4.5 needs an additional round and k bit multiplications in order to prepare the inputs for 4.6. However, if one of the operands is public, this initialization becomes a local computation and then the protocol 4.5 has the same round and communication complexity as 4.6.

Protocol 4.5: $[g]^{\mathbb{F}} \leftarrow \mathsf{CarryOut}(([a_k]^{\mathbb{F}}, \ldots, [a_1]^{\mathbb{F}}), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}}))$

1 **foreach** $i \in [1..k]$ **do parallel**
2 $\quad [d_i]_B^{\mathbb{F}} \leftarrow ([a_i]^{\mathbb{F}} + [b_i]^{\mathbb{F}} - 2[a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}}, [a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}})$; // 1 rnd, k inv ($\mathbb{F}$)
// $d_i = (p_i, g_i)$
3 $[d]_B^{\mathbb{F}} \leftarrow \mathsf{CarryOutAux}([d_k]_B^{\mathbb{F}}, \ldots, [d_1]_B^{\mathbb{F}}, k)$; // $\log_2 k$ rnd, $2k-2$ inv ($\mathbb{F}$)
4 $([p]^{\mathbb{F}}, [g]^{\mathbb{F}}) \leftarrow [d]_B^{\mathbb{F}}$;
5 **return** $[g]^{\mathbb{F}}$;

Protocol 4.6: $[d]_B^{\mathbb{F}} \leftarrow \mathsf{CarryOutAux}([d_k]_B^{\mathbb{F}}, \ldots [d_1]_B^{\mathbb{F}}, k)$

1 **if** $k > 1$ **then**
2 $\quad$ **foreach** $i \in [1..k/2]$ **do parallel**
3 $\quad\quad [u_i]_B^{\mathbb{F}} \leftarrow [d_{2i}]_B^{\mathbb{F}} \circ [d_{2i-1}]_B^{\mathbb{F}}$; // 1 rnd, $k-2$ inv ($\mathbb{F}$)
4 $\quad [d]_B^{\mathbb{F}} \leftarrow \mathsf{CarryOutAux}(([u_{\frac{k}{2}}]_B^{\mathbb{F}}, \ldots, [u_1]_B^{\mathbb{F}}), \frac{k}{2})$; // $\log_2 k - 1$ rnd, k inv ($\mathbb{F}$)
5 **else**
6 $\quad [d]_B^{\mathbb{F}} \leftarrow [d_1]_B^{\mathbb{F}}$;
7 **return** $[d]_B^{\mathbb{F}}$;

The protocols 4.4 and 4.5 can be extended in order to take as additional input a carry-in bit (e.g., by modifying the computation of g_1, s_1). In particular, we need Protocol 4.7, a variant of CarryOut with public carry-in bit, denoted c. The complexity is the same.

Protocol 4.7: $[g]^{\mathbb{F}} \leftarrow \mathsf{CarryOutCin}(([a_k]^{\mathbb{F}}, \ldots, [a_1]^{\mathbb{F}}), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}}), c)$

1 **foreach** $i \in [1..k]$ **do parallel**
2 $\quad [d_i]_B^{\mathbb{F}} \leftarrow ([a_i]^{\mathbb{F}} + [b_i]^{\mathbb{F}} - 2[a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}}, [a_i]^{\mathbb{F}}[b_i]^{\mathbb{F}})$; // 1 rnd, k inv ($\mathbb{F}$)
// $d_i = (p_i, g_i)$
3 $[g_1]^{\mathbb{F}} \leftarrow [g_1]^{\mathbb{F}} + c[p_1]^{\mathbb{F}}$;
4 $[d]_B^{\mathbb{F}} \leftarrow \mathsf{CarryOutAux}([d_k]_B^{\mathbb{F}}, \ldots, [d_1]_B^{\mathbb{F}}, k)$; // $\log_2 k$ rnd, $2k-2$ inv ($\mathbb{F}$)
5 $([p]^{\mathbb{F}}, [g]^{\mathbb{F}}) \leftarrow [d]_B^{\mathbb{F}}$;
6 **return** $[g]^{\mathbb{F}}$;

4.2.2 Comparison of Bitwise-Shared Values

The protocol BitLT for comparison of bitwise-shared integers described in [13, 30] (and D9.1 [26]), uses the relatively complex prefix-OR protocol. We present a more efficient solution based on binary addition.

Inequality test for bitwise-shared integers. Given two bitwise-shared k-bit unsigned integers $[a]_B$ and $[b]_B$, we want to compute a secret bit $[s]$ such that $s = (a < b)$.

Let $d = 2^k + a - b$. Observe that $0 < d < 2^{k+1}$ and let $d = d_k, \ldots, d_0$ be its binary representation. If $a - b < 0$ then $d_k = 0$ and if $a - b \geq 0$ then $d_k = 1$. Therefore, $s = 1 - d_k$.

Let $b' = \neg b$, where $\neg b$ is the bitwise negation of b, and observe that $2^k - b = b' + 1$. The bit d_k can be computed using Protocol 4.7, CarryOutCin, with inputs $[a]_B$, $[b']_B$, and the carry-in bit set.

We need as building block a variant with one input public, shown as Protocol 4.8. This protocol computes $[a < b]$ given the public k-bit integer a and the bitwise-shared integer $[b]^{\mathbb{F}}_B = ([b_{k-1}]^{\mathbb{F}}, \ldots, [b_0]^{\mathbb{F}})$.

Protocol 4.8 has the same round and communication complexity as the protocol CarryOutCin (with one input public).

Protocol 4.8: $[s]^{\mathbb{F}} \leftarrow \mathsf{BitLT}(a, [b]^{\mathbb{F}}_B)$

1 **foreach** $i \in [0..k-1]$ **do**
2 $\quad [b'_i]^{\mathbb{F}} \leftarrow 1 - [b_i]^{\mathbb{F}}$;
3 $[s]^{\mathbb{F}} \leftarrow 1 - \mathsf{CarryOutCin}((a_{k-1}, \ldots, a_0), ([b'_{k-1}]^{\mathbb{F}}, \ldots, [b'_0]^{\mathbb{F}}), 1)$; `// Set carry-in`
`// ` $\log_2 k$ `rnd,` $2k-2$ `inv (`$\mathbb{F}$`)`
4 **return** $[s]^{\mathbb{F}}$;

4.2.3 Summary

Security. The protocols in this section use building blocks with perfect privacy. Furthermore, no value is ever output in any of the protocols. Consequently, all the protocols discussed in this section provide perfect privacy.

Complexity. Table 8 gives the complexity of the protocols for bitwise operations.

Output	Protocol	Input	Fld	Rounds	Invoc.
$[s]^{\mathbb{F}}_B$	AddBitwise	$([a_k]^{\mathbb{F}}, \ldots, [a_1]^{\mathbb{F}}), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}})$	$\mathbb{F}$	$1 + \log_2 k$	$k + k \log_2 k$
$[s]^{\mathbb{F}}_B$	AddBitwise	$(a_k, \ldots, a_1), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}})$	$\mathbb{F}$	$\log_2 k$	$k \log_2 k$
$[g]^{\mathbb{F}}$	CarryOut	$([a_k]^{\mathbb{F}}, \ldots, [a_1]^{\mathbb{F}}), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}})$	$\mathbb{F}$	$1 + \log_2 k$	$3k - 2$
$[g]^{\mathbb{F}}$	CarryOut	$(a_k, \ldots, a_1), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}})$	$\mathbb{F}$	$\log_2 k$	$2k - 2$
$[g]^{\mathbb{F}}$	CarryOutCin	$([a_k]^{\mathbb{F}}, \ldots, [a_1]^{\mathbb{F}}), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}}), c$	$\mathbb{F}$	$1 + \log_2 k$	$3k - 2$
$[g]^{\mathbb{F}}$	CarryOutCin	$(a_k, \ldots, a_1), ([b_k]^{\mathbb{F}}, \ldots, [b_1]^{\mathbb{F}}), c$	$\mathbb{F}$	$\log_2 k$	$2k - 2$
$[s]^{\mathbb{F}}$	BitLT	$a, [b]^{\mathbb{F}}_B$	$\mathbb{F}$	$\log_2 k$	$2k - 2$

Table 8: Complexity of main protocols discussed in this section.

BitLT variants. The BitLT protocol presented in [13, 30] uses prefix-OR as main building block. The constant rounds solution for prefix-OR is impractical, and the variant with $\log(k)$ rounds needs $0.5k \log(k)$ invocations. The BitLT protocol based on binary addition (Protocol 4.8) runs in $\log(k)$ rounds with only $2k - 2$ invocations, hence it is significantly more efficient.

5 Arithmetic and Comparison

Outline. This chapter presents protocols for secure arithmetic and comparison with integers and fixed-point rational numbers. Variants of several protocols were introduced in the deliverables D9.1 and D3.1. We present and analyze more efficient variants and add new protocols. We include (for clarity and completeness) revised specifications and analysis for the entire family of arithmetic and comparison protocols, using the building blocks described in the previous chapters.

Throughout this chapter we assume that integers and fixed-point numbers are encoded in $\mathbb{Z}_q$ as specified in Section 2.5 and $q > 2^{k+\kappa+\nu+1}$, where $\nu = \lceil \log(\binom{n}{t}) \rceil$ and κ is a security parameter (as usually, n is the number of parties and t is the corruption threshold).

Protocols for arithmetic with integers are a trivial extension of the protocols for arithmetic with field elements described in Section 2.4.2 using the ideas of Section 2.5.2.

5.1 Truncation

Let $x \in [-2^{k-1}..2^{k-1}-1]$ be a k-bit integer. Truncation is the process of 'chopping off' some m (with $m < k$) least significant bits from the binary representation of x. Note that truncation by m bits is equivalent to computing the quotient of division by 2^m. All our truncation protocols take as input the sharing $[\mathbf{Fld}_q(x)]$ of a k-bit integer x encoded in $\mathbb{Z}_q$, and the public integers m and k. They output $[\mathbf{Fld}_q(x')]$, where x' depends on the truncation protocol being used.

We consider three variants of truncation protocols:

1. Trunc: This protocol computes a signed integer obtained by chopping off the m least significant bits of x, i.e., $x' = \lfloor x/2^m \rfloor$. An important application of Trunc is secure integer comparison (protocol 5.9, LTZ).

2. TruncPr: This protocol computes $x/2^m$ with probabilistic rounding towards the nearest integer. It returns $x' = \lfloor x/2^m \rfloor + u$, where $u \in \{0, 1\}$ is a random bit distributed such that x' is the signed integer closest to the rational value $x/2^m$. TruncPr is more efficient than Trunc. Its main applications are secure fixed-point multiplication and division.

3. TruncApp: This is protocol AppDiv2m described in D3.1 [27], an adaptation of a protocol proposed in [1] to signed integers encoded in $\mathbb{Z}_q$. TruncApp efficiently computes an approximation of $x/2^m$ with absolute error $\epsilon \leq n$, where n is the number of parties. More precisely, it returns $x' = \lfloor x/2^m \rfloor + \epsilon$, where $\epsilon \leq n$ is a random value. TruncApp can be used for secure fixed-point multiplication, but it is less accurate than TruncPr.

Section 5.1.1 presents revised versions of a protocol for reduction modulo 2^m introduced in D3.1, which is the core component of Trunc. Protocol Trunc is described in Section 5.1.2, followed by the new protocol TruncPr in Section 5.1.3.

5.1.1 Reduction Modulo 2^m

The protocol Mod2m computes $x' = x \bmod 2^m$ for any $x \in [-2^{k-1}..2^{k-1}-1]$ and $0 < m < k$. The inputs are $[a] = [\mathbf{Fld}_q(x)]$ and the public integers k and m. The output is $[\mathbf{Fld}_q(x \bmod 2^m)]$. The protocol provides statistical privacy with security parameter κ.

A first version of this protocol was described in D3.1 [27], using building blocks presented in D9.1 [26]. The performance of this first version degrades with the growth of the parameters q, k, and m, and becomes very poor for the values needed in our applications. We present in the following more efficient versions, based on the following improvements: (1) generation of m shared random bits instead of $k+\kappa$ bits; (2) efficient encoding of the bits in $\mathbb{F}_{2^8}$ instead of $\mathbb{Z}_q$ for comparison of bitwise shared integers; (3) more efficient building blocks for generating shared random bits and integers and for comparison of bitwise shared integers.

Protocol 5.1 is the reference variant, where all data types (integers, bits) are encoded and shared in the same field $\mathbb{Z}_q$. This solution is suitable for small values of q and m (e.g., $\log(q) \leq 128$ bits). As these values grow, encoding the bits in the same field as the integers becomes inefficient.

Protocol 5.1: $[b'] \leftarrow \mathsf{Mod2m}([a], k, m)$

1 $[b] \leftarrow 2^{k-1} + [a]$;
2 **foreach** $i \in [0..m-1]$ **do parallel**
3 $\quad [r_i] \leftarrow \mathsf{PRandBit}(q)$; // 1 rnd, m inv, m exp ($\mathbb{Z}_q$)
4 $[r']_B \leftarrow ([r_{m-1}], \ldots, [r_0])$;
5 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$;
6 $[r''] \leftarrow \mathsf{PRandInt}(k+\kappa-m)$;
7 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
8 $c \leftarrow \mathsf{Output}([b] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
9 $c' \leftarrow c \bmod 2^m$;
10 $[u] \leftarrow \mathsf{BitLT}(c', [r']_B)$; // $\log m$ rnd, $2m-2$ inv ($\mathbb{Z}_q$)
11 $[b'] \leftarrow c' - [r'] + [u] \cdot 2^m$;
12 **return** $[b']$;

Correctness. The protocol maps $x \in [-2^{k-1}..2^{k-1}-1]$ to $b \in [0..2^k-1]$ by computing $b = (2^{k-1}+a) \bmod q = 2^{k-1}+x$. Observe that $b \bmod 2^m = x \bmod 2^m$ for any $0 < m < k$. Next, it generates a random secret $r \in [0..2^{k+\kappa}-1]$ and reveals $c = (b+r) \bmod q$. Since $q > 2^{k+\kappa+1}$ we have $q > b + r$ and hence $c = b + r$. Let $c' = c \bmod 2^m$, $b' = b \bmod 2^m$, and $r' = r \bmod 2^m$. We see that $c' = (b' + r') \bmod 2^m$ and $b' = c' - r' + u \cdot 2^m$, where $u = 1$ if $c' < r'$ and $u = 0$ if $c' \geq r'$. Since $b' = x \bmod 2^m = \mathbf{Fld}_q(x \bmod 2^m)$, the output is correct.

Mod2m with efficient bit-sharing. Protocol 5.2 is a variant of Mod2m that uses bits shared in small fields in order to improve the efficiency of bit operations for large q and m (lower communication and computation complexity).

The protocol generates a double bitwise sharing of r', with bits shared in $\mathbb{Z}_q$ and in $\mathbb{F}_{2^8}$, using PRandBitD (Steps 2-3). We denote $[u]^{\mathbb{F}_{2^8}}$ a Shamir sharing of bit u in $\mathbb{F}_{2^8}$. The sharing in $\mathbb{Z}_q$ is used to compute $[r']$ (Step 5), while the sharing in $\mathbb{F}_{2^8}$ is used as argument for BitLT (Step 10). Step 11 converts the output of BitLT from $\mathbb{F}_{2^8}$ to $\mathbb{Z}_q$ (using BitF2MtoZQpre), as needed in Step 12. This conversion is efficiently achieved (1 round, 1 invocation) using a double-shared random bit generated in Steps 2-3.

Protocol 5.2: $[b'] \leftarrow \mathsf{Mod2mF}([a], k, m)$

1 $[b] \leftarrow 2^{k-1} + [a]$;
2 **foreach** $i \in [0..m]$ **do parallel**
3 $\quad ([r_i]^{\mathbb{F}_{2^8}}, [r_i]) \leftarrow \mathsf{PRandBitD}(8, q)$; // 2 rnd, $2m+2$ inv, $m+1$ exp ($\mathbb{Z}_{q_1}$)
4 $[r']_B^{\mathbb{F}_{2^8}} \leftarrow ([r_{m-1}]^{\mathbb{F}_{2^8}}, \ldots, [r_0]^{\mathbb{F}_{2^8}})$;
5 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$;
6 $[r''] \leftarrow \mathsf{PRandInt}(k + \kappa - m)$;
7 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
8 $c \leftarrow \mathsf{Output}([b] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
9 $c' \leftarrow c \bmod 2^m$;
10 $[u]^{\mathbb{F}_{2^8}} \leftarrow \mathsf{BitLT}(c', [r']_B^{\mathbb{F}_{2^8}})$; // $\log m$ rnd, $2m-2$ inv ($\mathbb{F}_{2^8}$)
11 $[u] \leftarrow \mathsf{BitF2MtoZQPre}([u]^{\mathbb{F}_{2^8}}, [r_m]^{\mathbb{F}_{2^8}}, [r_m])$; // 1 rnd, 1 inv ($\mathbb{F}_{2^8}$)
12 $[b'] \leftarrow c' - [r'] + [u] \cdot 2^m$;
13 **return** $[b']$;

Security. Protocols 5.1 and 5.2 can leak information only in step 8, where a value is output. From Theorem 5 we conclude that $\Delta(c, r) \leq 2^{-\kappa}$. The other components provide perfect or statistical privacy. Therefore, the protocols provide statistical privacy with security parameter κ.

Complexity. The complexity of these protocols depends on the solution used for binary computation:

- Protocol 5.1, with bits shared in $\mathbb{Z}_q$: 1 round and $m+1$ invocations for Steps 2-3 (PRandBit); 1 round and 1 invocation for Step 8; $\log(m)$ rounds and $2m-2$ invocations for Step 10 (BitLT). Overall, $\log(m)+2$ rounds and $3m$ invocations in $\mathbb{Z}_q$. Steps 2-3 can be precomputed (1 rnd, $m+1$ invocations in $\mathbb{Z}_q$). Each of the m shared random bits generated in Step 1 requires an exponentiation $r^{\frac{q+1}{4}}$, which slows down considerably the protocol for large q and m, and many parallel instances (e.g., secure Simplex with integer pivoting).

- Protocol 5.2, with bits shared in $\mathbb{F}_{2^8}$: 2 rounds and $2m+2$ invocations in a small field $\mathbb{Z}_{q_1}$ for Steps 2-3 (PRandBitD); 1 round and 1 invocation in $\mathbb{Z}_q$ for Step 8; $\log(m)$ rounds and $2m-2$ invocations in $\mathbb{F}_{2^8}$ for Step 10 (BitLT); 1 round and 1 invocation in $\mathbb{F}_{2^8}$ for Step 11. Overall, $\log(m)+4$ rounds and $2m+2$ invocations in $\mathbb{Z}_{q_1}$, 1 invocation in $\mathbb{Z}_q$, and $2m-1$ invocations in $\mathbb{F}_{2^8}$. Steps 2-3 dominate the communication complexity and can be precomputed (2 rounds, $2m+2$ invocations in $\mathbb{Z}_{q_1}$). The protocol PRandBitD generates shared random bits in a small field $\mathbb{Z}_{q_1}$, reducing substantially the computation complexity.

Reduction modulo 2 (extraction of the least significant bit). Protocol 5.3 computes $x' = x \bmod 2$ for any $x \in [-2^{k-1}..2^{k-1}-1]$, i.e., extracts the least significant bit of x (for 2's complement representation of negative integers). It uses the same method as Mod2m, but the protocol is much simpler because we do not need BitLT. The complexity of the protocol is 2 rounds and 2 invocations.

Protocol 5.3: $[a_0] \leftarrow \mathsf{LSB}([a], k)$

1 $[b] \leftarrow \mathsf{PRandBit}()$; // 1 rnd, 1 inv, 1 exp ($\mathbb{Z}_q$)
2 $[r] \leftarrow \mathsf{PRandInt}(k + \kappa - 1)$;
3 $c \leftarrow \mathsf{Output}(2^{k-1} + [a] + 2[r] + [b])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
4 $[a_0] \leftarrow c_0 + [b] - 2c_0[b]$;
5 **return** $[a_0]$;

5.1.2 Truncation

Protocols 5.4 and 5.5 compute $\lfloor x/2^m \rfloor = (x - (x \bmod 2^m))2^{-m}$ for any $x \in [-2^{k-1}..2^{k-1} - 1]$ and $0 < m < k$. This is equivalent to cutting off the m least significant bits of the binary representation of x. The protocols take as inputs $[a] = [\mathbf{Fld}_q(x)]$ and the public integers k and m, and return $[\mathbf{Fld}_q(\lfloor x/2^m \rfloor)]$. They provide statistical privacy with security parameter κ.

Protocol 5.4: $[d] \leftarrow \mathsf{Trunc}([a], k, m)$

1 $[a'] \leftarrow \mathsf{Mod2m}([a], k, m)$; // $2 + \log m$ rnd, $3m - 1$ inv, m exp ($\mathbb{Z}_q$)
2 $[d] \leftarrow ([a] - [a'])(2^{-m} \bmod q)$;
3 **return** $[d]$;

Protocol 5.5: $[d] \leftarrow \mathsf{TruncF}([a], k, m)$

1 $[a'] \leftarrow \mathsf{Mod2mF}([a], k, m)$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
// 2 rnd, $2m + 2$ inv, $m + 1$ exp ($\mathbb{Z}_{q_1}$)
// $1 + \log m$ rnd, $2m - 1$ inv ($\mathbb{F}_{2^8}$)
2 $[d] \leftarrow ([a] - [a'])(2^{-m} \bmod q)$;
3 **return** $[d]$;

Correctness. Let $d' = (a - a') \bmod q$. Since $a = x \bmod q$ and $a' = x \bmod 2^m$ we have $d' = (x - (x \bmod 2^m)) \bmod q = (\lfloor x/2^m \rfloor \cdot 2^m) \bmod q$. The protocol returns $d = d' \cdot (2^{-m} \bmod q) \bmod q$. Correctness follows from the fact that $d = \lfloor x/2^m \rfloor \bmod q = \mathbf{Fld}_q(\lfloor x/2^m \rfloor)$.

Analysis. Security/complexity are the same as for protocols 5.1 and 5.2, respectively.

Division by 2. For $m = 1$ we can compute $y = \lfloor x/2 \rfloor$ using the simpler protocol 5.3.

5.1.3 Truncation With Probabilistic Rounding

Rounding to the nearest integer. Suppose now that we have to compute an approximation of $x/2^m$, rather than cutting off the m least significant bits of x (e.g., for multiplication of fixed-point numbers). Protocols 5.4 and 5.5 compute $d = \lfloor x/2^m \rfloor$, which approximates $x/2^m$ with the absolute error $\epsilon = |x/2^m - d| < 1$. We can reduce the error to ≤ 0.5 by rounding to the nearest integer , i.e., by computing $d' = \lfloor x/2^m \rfloor + u$, where $u = 0$ if $x \bmod 2^m < 2^m/2$ and $u = 1$ if $x \bmod 2^m \geq 2^m/2$. However, this rounding requires a secure comparison, hence it is inefficient.

Protocol 5.6 offers probabilistic rounding "for free". It computes $\lfloor x/2^m \rfloor + u$, where u is a random bit, for any $x \in [-2^{k-1}..2^{k-1} - 1]$ and $0 < m < k$. The random bit

u is distributed such that the fraction $x/2^m$ is probabilistically rounded to the nearest integer. The inputs are $[a] = [\mathbf{Fld}_q(x)]$ and the public integers k and m. The output is $[\mathbf{Fld}_q(\lfloor x/2^m \rfloor + u)]$. The protocol provides statistical privacy with security parameter κ.

Protocol 5.6: $[d] \leftarrow \mathsf{TruncPr}([a], k, m)$

1 $([r'], [r']_B) \leftarrow \mathsf{PRand2mU}(q, m)$; `// 1 rnd, m inv, m exp (Z_q)`
2 $[r''] \leftarrow \mathsf{PRandInt}(q, k + \kappa - m)$;
3 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
4 $[b] \leftarrow 2^{k-1} + [a]$;
5 $c \leftarrow \mathsf{Output}([b] + [r])$; `// 1 rnd, 1 inv (Z_q)`
6 $c' \leftarrow c \bmod 2^m$;
7 $[a'] \leftarrow c' - [r']$;
8 $[d] \leftarrow ([a] - [a'])2^{-m}$;
9 **return** $[d]$;

Correctness. Steps 1-7 compute the value $a' = (x \bmod 2^m) - u \cdot 2^m$, where $u \in \{0, 1\}$, using the same method as protocol 5.1, Mod2m. To see that, observe that $b = (2^{k-1} + a) \bmod q = 2^{k-1} + x$, $b \bmod 2^m = x \bmod 2^m$, and $a' = (b \bmod 2^m) - u \cdot 2^m$. Let $d' = (a - a') \bmod q$ and observe that $d' = (x - (x \bmod 2^m) + u \cdot 2^m) \bmod q = (\lfloor x/2^m \rfloor \cdot 2^m + u \cdot 2^m) \bmod q$. The protocol returns $d = d' \cdot (2^{-m} \bmod q) \bmod q$. We have $d = (\lfloor x/2^m \rfloor + u) \bmod q = \mathbf{Fld}_q(\lfloor x/2^m \rfloor + u)$, hence the output is correct.

Probabilistic rounding. Let $x' = x \bmod 2^m$ and $r' = r \bmod 2^m$. Observe that $x'+r' \in [0..2^{m+1}-2]$ and $u = 1$ if $x'+r' \geq 2^m$ and $u = 0$ if $x'+r' < 2^m-1$. Denote $p(x')$ the probability that $u = 1$ for given x'. It follows that $p(x') = \mathbb{P}_{x'}(u = 1) = \mathbb{P}_{x'}(r' \geq 2^m - x')$. We see that $p(x')$ grows with x' from $p(0) = 0$ to $p(2^m - 1) \approx 1$. For example, if r' is random uniform in $[0..2^m - 1]$, we obtain $p(x') = x'/2^m$, hence $p(0) = 0$, $p(2^m/2) = 1/2$, and $p(2^m - 1) = 1 - 2^{-m}$.

Absolute error. Let ϵ_1 be the error of protocols 5.4/5.5 and ϵ_2 the error of protocol 5.6. Observe that $\epsilon_2 < 1$ and $\epsilon_2 < \epsilon_1$ in all cases except when $x' < 2^m/2$ and $r' > 2^m - x'$.

Complexity. Step 1 needs 1 round and m invocations, and step 5 needs 1 round and 1 invocation (in $\mathbb{Z}_q$). Overall, 2 rounds and $m + 1$ invocations. Note that Steps 1-2 can be precomputed.

More efficient variant for large q. We can improve the efficiency of Protocol 5.6 by generating the shared random bits in a small field using PRandBitL (Protocol 3.18), as shown in Protocol 5.7.

Protocol 5.7: $[d] \leftarrow \mathsf{TruncPrF}([a], k, m)$

1 **foreach** $i \in [0..m-1]$ **do parallel**
2 $\quad [r_i] \leftarrow \mathsf{PRandBitL}(q)$; // 2 rnd, $2m$ inv, m exp ($\mathbb{Z}_{q_1}$)
3 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$;
4 $[r''] \leftarrow \mathsf{PRandInt}(q, k+\kappa-m)$;
5 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
6 $[b] \leftarrow 2^{k-1} + [a]$;
7 $c \leftarrow \mathsf{Output}([b] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
8 $c' \leftarrow c \bmod 2^m$;
9 $[a'] \leftarrow c' - [r']$;
10 $[d] \leftarrow ([a] - [a'])2^{-m}$;
11 **return** $[d]$;

Security. Protocols 5.6 and 5.7 can leak information only when they output $c = b+r$. From Theorem 5 we conclude that $\Delta(c, r) \leq 2^{-\kappa}$. The other components provide perfect or statistical privacy. Therefore, the protocols provide statistical privacy with security parameter κ.

Fast truncation for fixed-point multiplication. Protocol 5.8 is a variant of TruncPr that generates r' without interaction, using PRand2mN (instead of PRand2mU). This variant is very efficient: 1 round and 1 invocation in $\mathbb{Z}_q$. On the other hand, it provides weaker protection for the truncated part. Observe that for input $[a] = [\mathbf{Fld}_q(x)]$ all variants reveal $c' = ((x \bmod 2^m) + r') \bmod 2^m$, where $r' \in [0, 2^m - 1]$. The difference is that in TruncPrN the secret random r' is a sum of random integers, while in the previous variants r' is uniformly distributed. However, the protection offered by TruncPrN can be sufficient for particular applications, e.g., secure multiplication of fixed-point numbers (see section 5.4.1).

Protocol 5.8: $[d] \leftarrow \mathsf{TruncPrN}([a], k, m)$

1 $[r'] \leftarrow \mathsf{PRand2mN}(q, m)$;
2 $[r''] \leftarrow \mathsf{PRandInt}(q, k+\kappa-m)$;
3 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
4 $[b] \leftarrow 2^{k-1} + [a]$;
5 $c \leftarrow \mathsf{Output}([b] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
6 $c' \leftarrow c \bmod 2^m$;
7 $[a'] \leftarrow c' - [r']$;
8 $[d] \leftarrow ([a] - [a'])2^{-m}$;
9 **return** $[d]$;

5.1.4 Comparison of Truncation Variants

The protocol TruncPr is substantially more efficient than Trunc (essentially, because it avoids BitLT). On the other hand, from the point of view of applications we distinguish the following cases:

- If the task is to compute $\lfloor x/2^m \rfloor$ (i.e., cut off the m least significant bits) then Trunc provides the exact result, while TruncPr computes an approximation with maximum

absolute error $\epsilon < 1$. Some applications need the exact result and can only use Trunc (e.g., integer comparison).

- If the task is to compute an approximation of $x/2^m$ then the maximum absolute error of both protocols is $\epsilon < 1$. However, the error of TruncPr is more likely less than $\epsilon \leq 0.5$. Since TruncPr is much more efficient, it is obvious choice for this type of applications (e.g., fixed-point multiplication).

The protocol TruncPr is more accurate than TruncApp, which computes the truncation with absolute error $\epsilon \leq n$. With precomputation of $[r']$, TruncPr performs a truncation in 1 round and 1 invocation, hence it becomes more efficient than TruncApp, which needs 3 rounds and 4 invocations. We call this variant TruncPrPre$([a], k, m, [r'])$. A similarly optimized variant of TruncPrF is called TruncPrFPre.

TruncPrN is the most efficient variant of TruncPr: 1 round and 1 invocation in $\mathbb{Z}_q$. On the other hand, it provides weaker security.

5.2 Integer Comparison

Family of protocols for integer comparison. The entire family of secure integer comparison operators with secret inputs and outputs can be constructed based on two protocols: EQZ(a), that computes $a = 0$, and LTZ(a), that computes $a < 0$. The names of the protocols and the constructions are summarized in Table 9. For the operators with two inputs, one input may be a public value.

Operation	Protocol	Construction	Complexity
$a = 0$	EQZ(a)	Primitive	see Table 10
$a = b$	EQ(a, b)	EQ$(a, b) =$ EQZ$(a - b)$	same as EQZ
$a < 0$	LTZ(a)	Primitive (sign of a)	see Table 10
$a > 0$	GTZ(a)	GTZ$(a) =$ LTZ$(-a)$	same as LTZ
$a \leq 0$	LEZ(a)	LEZ$(a) = 1 -$ LTZ$(-a)$	same as LTZ
$a \geq 0$	GEZ(a)	GEZ$(a) = 1 -$ LTZ(a)	same as LTZ
$a < b$	LT(a, b)	LT$(a, b) =$ LTZ$(a - b)$	same as LTZ
$a > b$	GT(a, b)	GT$(a, b) =$ LTZ$(b - a)$	same as LTZ
$a \leq b$	LE(a, b)	LE$(a, b) = 1 -$ LTZ$(b - a)$	same as LTZ
$a \geq b$	GE(a, b)	GE$(a, b) = 1 -$ LTZ$(a - b)$	same as LTZ

Table 9: The family of protocols for integer comparison.

Protocol LTZ (Less Than Zero). Protocol 5.9 computes $s = (x < 0)$ for any $x \in [-2^{k-1}..2^{k-1} - 1]$. The protocol takes as inputs $[a] = [\mathbf{Fld}_q(x)]$ and the public integer k, and returns $[s] = [\mathbf{Fld}_q(x < 0)]$. It provides statistical privacy with security parameter κ.

Protocol 5.9: $[s] \leftarrow$ LTZ$([a], k)$

1 $[s] \leftarrow -$Trunc$([a], k, k - 1)$;
// $2 + \log(k - 1)$ `rnd`, $3k - 4$ `inv`, $k - 1$ `exp` $(\mathbb{Z}_q)$
2 **return** $[s]$;

Protocol 5.10 is a variation that trades off between rounds and data transmitted by having more rounds but invocations in a smaller field.

Protocol 5.10: $[s] \leftarrow \mathsf{LTZF}([a], k)$

1 $[s] \leftarrow -\mathsf{TruncF}([a], k, k-1)$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
// 2 rnd, $2k$ inv, k exp ($\mathbb{Z}_{q_1}$)
// $1 + \log(k-1)$ rnd, $2k-3$ inv ($\mathbb{F}_{2^8}$)
2 **return** $[s]$;

Correctness. The protocols return $s = -\lfloor x/2^{k-1} \rfloor$. Since $x \in [-2^{k-1}..2^{k-1}-1]$ it follows that $s = 1$ for any $x \in [-2^{k-1}..-1]$ and $s = 0$ for any $x \in [0..2^{k-1}-1]$, hence the output is correct.

Remark. When LTZ is used to construct comparison operators with two inputs, care must be taken to ensure that subtraction of the inputs does not overflow. For example, if a and b are k-bit integers and we need an operator LT (less than) that works for the entire range of the inputs, we can use $\mathsf{LT}([a], [b], k) = \mathsf{LTZ}([a] - [b], k+1)$.

Protocol EQZ (Equal Zero). Protocol 5.11 computes $s = (x = 0)$ for any secret integer $x \in [-2^{k-1}..2^{k-1}-1]$. The protocol takes as inputs $[a] = [\mathbf{Fld}_q(x)]$ and the public integer k, and returns $[s] = [\mathbf{Fld}_q(x = 0)]$.

Protocol 5.11: $[s] \leftarrow \mathsf{EQZ}([a], k)$

1 $([r'], [r']_B) \leftarrow \mathsf{PRand2mU}(q, k)$; // 1 rnd, k inv, k exp ($\mathbb{Z}_q$)
2 $[r''] \leftarrow \mathsf{PRandInt}(\kappa)$;
3 $[r] \leftarrow 2^k \cdot [r''] + [r']$;
4 $c \leftarrow \mathsf{Output}([a] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
5 parse $[r']_B$ as $([r'_0], [r'_1], \ldots, [r'_{k-1}])$;
6 **foreach** $i \in [0..k-1]$ **do**
7 $[s_i] \leftarrow c_i + [r'_i] - 2c_i[r'_i]$; // $c_i \oplus [r'_i]$
8 $[s] \leftarrow 1 - \mathsf{KOpL}(\mathsf{OR}, ([s_0], [s_1], \ldots, [s_{k-1}]))$; // $\log k$ rnd, $k-1$ inv ($\mathbb{Z}_q$)
9 **return** $[s]$;

Correctness. The protocol generates a random integer $0 \le r \le 2^{k+\kappa+\nu}$ and then computes $c = (a + r) \bmod q = (x + r) \bmod q$. Let $c' = c \bmod 2^k$ and $r' = r \bmod 2^k$, and observe that $c' = r'$ if and only if $x = 0$. In steps 6-8 the protocol tests if $c' = r'$ by computing the bitwise-XOR of c' and r' and then testing if the result is null using the k-ary OR protocol.

Security. The protocol can leak information about the input only in step 4, when it reveals $c = x + r$. From Theorem 5 we can conclude that $\Delta(c, r) \le 2^{-\kappa}$. The other building blocks provide perfect privacy or statistical privacy with security parameter κ. Therefore, the protocol provides statistical privacy with security parameter κ.

Variant with efficient bit-sharing. We can improve the efficiency of EQZ (for larger q) using the same methods as in Protocol 5.2, as shown in Protocol 5.12. This variant returns a bit shared in $\mathbb{F}_{2^8}$ (conversion to $\mathbb{Z}_q$ requires an additional round).

Protocol 5.12: $[s]^{\mathbb{F}_{2^8}} \leftarrow \mathsf{EQZF}([a], k)$

1 **foreach** $i \in [0..k-1]$ **do parallel**
2 $\quad([r_i]^{\mathbb{F}_{2^8}}, [r_i]) \leftarrow \mathsf{PRandBitD}(8, q)$; // 2 rnd, $2k$ inv, k exp ($\mathbb{Z}_{q_1}$)
3 $[r']_B^{\mathbb{F}_{2^8}} \leftarrow ([r_{k-1}]^{\mathbb{F}_{2^8}}, \ldots, [r_0]^{\mathbb{F}_{2^8}})$;
4 $[r'] \leftarrow \sum_{i=0}^{k-1} 2^i \cdot [r_i]$;
5 $[r''] \leftarrow \mathsf{PRandInt}(\kappa)$;
6 $[r] \leftarrow 2^k \cdot [r''] + [r']$;
7 $c \leftarrow \mathsf{Output}([a] + [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
8 **foreach** $i \in [0..k-1]$ **do**
9 $\quad[s_i]^{\mathbb{F}_{2^8}} \leftarrow c_i \oplus [r'_i]^{\mathbb{F}_{2^8}}$;
10 $[s]^{\mathbb{F}_{2^8}} \leftarrow \mathsf{KOpL}(\mathsf{OR}, ([s_0]^{\mathbb{F}_{2^8}}, \ldots, [s_{k-1}]^{\mathbb{F}_{2^8}}))$; // $\log k$ rnd, $k-1$ inv ($\mathbb{F}_{2^8}$)
11 **return** $1 - [s]^{\mathbb{F}_{2^8}}$;

Equality with public output. Protocol 5.13 is a very efficient solution for equality test with secret inputs and public output. The protocol offers perfect privacy.

Given the secret inputs $[a]$ and $[b]$, the parties generate a shared random secret $[r]$, compute $[c] = ([a] - [b])[r]$, and then open c. If $c = 0$ then $(a = b) = 1$, otherwise $(a = b) = 0$. The output is incorrect if $a \neq b$ and $r = 0$, but this occurs with probability $< 1/q$, which is negligible in our setting.

Protocol 5.13: $[s] \leftarrow \mathsf{EQPub}([a], [b])$

1 $[r] \leftarrow \mathsf{PRandFld}(\mathbb{Z}_q)$;
2 $c \leftarrow \mathsf{MulPub}([a] - [b], [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
3 $s \leftarrow (c = 0)?1 : 0$;
4 **return** s;

Faster Variant of GTZ: We also consider a faster variant of GTZ called GTZF, that uses LTZF (instead of LTZ). The complexity of this protocol is the same as that of LTZF.

5.3 Bit Decomposition

The protocol BitDec is a general tool that provides a bridge between secure computation with integers shared in $\mathbb{Z}_q$ and with integers that are bitwise-shared in $\mathbb{Z}_q$ or $\mathbb{F}_{2^8}$. In particular, we use BitDec in the protocol that computes the reciprocal of a fixed-point number using the Newton-Raphson method.

The inputs are $[a] = [\mathbf{Fld}_q(x)]$ and the public integers k and m, where $x \in [-2^{k-1}..2^{k-1}-1]$ and $0 < m \leq k$. The output is an array of shared bits equal to the m least significant bits of the 2's complement binary representation of x. The protocol provides statistical privacy with security parameter κ.

Protocols 5.14 and 5.15 are more efficient variants of the bit decomposition protocol introduced in D3.1 [27]. Protocol 5.14 uses bits encoded and shared in the same field $\mathbb{Z}_q$ as the integers. This variant is suitable for small q and m and applications that need output bits shared in $\mathbb{Z}_q$. Protocol 5.15 uses bits encoded and shared in $\mathbb{F}_{2^8}$, and it is more efficient for large q and m and output bits shared in $\mathbb{F}_{2^8}$ (in particular, for the reciprocal of fixed-point numbers). Both variants generate only m secret random bits, instead of $k + \kappa$ random bits in the version presented in D3.1.

Protocol 5.14: $([a_{m-1}], \ldots, [a_0]) \leftarrow \mathsf{BitDec}([a], k, m)$

1 **foreach** $i \in [0..m-1]$ **do parallel**
2 $\quad [r_i] \leftarrow \mathsf{PRandBit}(q)$; // 1 rnd, m inv, m exp ($\mathbb{Z}_q$)
3 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$;
4 $[r''] \leftarrow \mathsf{PRandInt}(k+\kappa-m)$;
5 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
6 $c \leftarrow \mathsf{Output}(2^{k+\kappa+\nu} + 2^k + [a] - [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
7 $([a_{m-1}], \ldots, [a_0]) \leftarrow \mathsf{AddBitwise}((c_{m-1}, \ldots, c_0), ([r_{m-1}], \ldots, [r_0]))$;
// $\log m$ rnd, $m \log m$ inv ($\mathbb{Z}_q$)
8 **return** $([a_{m-1}], \ldots, [a_0])$;

Protocol 5.15: $([a_{m-1}]^{\mathbb{F}_{2^8}}, \ldots, [a_0]^{\mathbb{F}_{2^8}}) \leftarrow \mathsf{BitDecF}([a], k, m)$

1 **foreach** $i \in [0..m-1]$ **do parallel**
2 $\quad [r_i]^{\mathbb{F}_{2^8}}, [r_i] \leftarrow \mathsf{PRandBitD}(m, q)$; // 2 rnd, $2m$ inv, m exp ($\mathbb{Z}_{q_1}$)
3 $[r'] \leftarrow \sum_{i=0}^{m-1} 2^i \cdot [r_i]$;
4 $[r''] \leftarrow \mathsf{PRandInt}(k+\kappa-m)$;
5 $[r] \leftarrow 2^m \cdot [r''] + [r']$;
6 $c \leftarrow \mathsf{Output}(2^{k+\kappa+\nu} + 2^k + [a] - [r])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
7 $([a_{m-1}]^{\mathbb{F}_{2^8}}, \ldots, [a_0]^{\mathbb{F}_{2^8}}) \leftarrow \mathsf{AddBitwise}((c_{m-1}, \ldots, c_0), ([r_{m-1}]^{\mathbb{F}_{2^8}}, \ldots, [r_0]^{\mathbb{F}_{2^8}}))$;
// $\log m$ rnd, $m \log m$ inv ($\mathbb{F}_{2^8}$)
8 **return** $([a_{m-1}]^{\mathbb{F}_{2^8}}, \ldots, [a_0]^{\mathbb{F}_{2^8}})$;

Precomputation. The random bits output in Step 2 can be precomputed. This results in a saving of 2 rounds in $\mathbb{Z}_{q_1}$. We call this variant BitDecFPre.

Correctness. Let $b = (2^k + a) \bmod q$, and observe that $b = 2^k + x$ and $b \bmod 2^k = x \bmod 2^k$. The protocol generates a random integer $0 \le r \le 2^{k+\kappa+\nu} - 1$ and then computes $c = (2^{k+\kappa+\nu} + 2^k + a - r) \bmod q$. Since $q > 2^{k+\kappa+\nu+1}$ we have $c = 2^{k+\kappa+\nu} + 2^k + x - r$.

Observe that if $x \ge 0$ then $(r + c) \bmod 2^k = x$ and if $x < 0$ then $(r + c) \bmod 2^k = 2^k - |x|$, so the value $(r + c) \bmod 2^k$ is equal to the 2's complement representation of x. The protocol uses binary addition to compute the $m \le k$ least significant bits of x.

Security. The protocols 5.14 and 5.15 can leak information only in step 6, where a value is output. From Theorem 5 we conclude that $\Delta(c, r) \le 2^{-\kappa}$. The other building blocks provide perfect privacy or statistical privacy with security parameter κ. Therefore, the protocols provide statistical privacy with security parameter κ.

Complexity. The complexity of the protocol depends on the method used for encoding the binary values:

- Protocol 5.14, with bits shared in $\mathbb{Z}_q$: 1 round and m invocations in $\mathbb{Z}_q$ for steps 1-2 (PRandBit); 1 round and 1 invocation in $\mathbb{Z}_q$ for step 6; $\log(m)$ rounds and $m \log(m)$ invocations in $\mathbb{Z}_q$ for step 7 (AddBitwise with one input public). Overall, $\log(m) + 2$ rounds and $m \log(m) + 1$ invocations in $\mathbb{Z}_q$.

- Protocol 5.15, with bits shared in $\mathbb{F}_{2^8}$: 2 rounds and $2m$ invocations in the small field $\mathbb{Z}_{q_1}$ for steps 1-2 (PRandBitD); 1 round and 1 invocation in $\mathbb{Z}_q$ for step 6; $\log(m)$ rounds and $m \log(m)$ invocations in $\mathbb{F}_{2^8}$ for step 7; 1 round and 1 invocation in $\mathbb{Z}_q$ for step 7. Overall, $\log(m) + 3$ rounds consisting of $2m$ invocations in $\mathbb{Z}_{q_1}$, $m \log(m)$

invocations in $\mathbb{F}_{2^8}$, and 1 invocation in $\mathbb{Z}_q$. For large q and m, the communication and computation complexity are substantially reduced using protocol PRandBitD and binary addition with bits shared in $\mathbb{F}_{2^8}$.

5.4 Fixed-Point Arithmetic

In this section, we give protocols for the following operations on fixed-point numbers: multiplication, inner product of two vectors, and reciprocal. We refer the reader to Section 2.5 for data representation. We will use the letters e and f to denote the magnitude and resolution of all fixed-point numbers and omit them when there is no ambiguity.

5.4.1 Fixed-Point Multiplication

Let x, y be fixed-point numbers. As mentioned earlier, we represent both x and y as integers $x' = 2^f x$ and $y' = 2^f y$. We then compute the product $z' = 2^f xy$ as follows: first we compute $z'' = x'y' = 2^{2f} xy$. Then we truncate z'' by f bits to obtain an approximation (with a loss of f bits) of the product z'. The truncation is not necessary if one of x, y is an integer. Let $[a], [b]$ be the sharings of field elements a, b encoding fixed-point numbers. The multiplication protocol is as follows:

Protocol 5.16: $[d] \leftarrow \mathsf{FPMul}([a], [b], e, f)$

1 $[c] \leftarrow [a][b]$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
2 $[d] \leftarrow \mathsf{Truncate}([c], 2e + 2f, f)$; // See below
3 **return** $[d]$;

Here Truncate $\in \{$Trunc, TruncF, TruncPr, TruncPrF, TruncPrN, TruncApp$\}$. We use the notation FPMul, FPMulF, FPMulPr, FPMulPrF, FPMulPrN, and FPMulApp respectively to denote the variants. The complexity is given in Table 11.

Security. All variants except FPMulPrN provide statistical privacy. FPMulPrN is much more efficient than the other variants (2 rounds and 2 invocations), but offers slightly weaker privacy. This variant uses the truncation protocol TruncPrN which reveals $w = (z'' \bmod 2^f + r') \bmod 2^f$, where $r' \in [0, 2^f - 1]$ is not uniformly distributed. Observe that the weaker protection affects only the least significant f bits of the integer product of the inputs, and that these bits are discarded. The protocol only leaks information that can be inferred about inputs and output from w and the distribution of r'. We assume that the information leaked is negligible for the secure Simplex protocols in Chapter 6.

5.4.2 Fixed-Point Inner Product

Notation. Let $\mathbf{a} = (a_1, a_2, \ldots, a_m) \in {\mathbb{Z}_q}^m$ be an m-vector of field elements representing fixed-point numbers. For $i \in [1..m]$, let $(a_{(i,1)}, a_{(i,2)}, \ldots, a_{(i,n)})$ be a sharing of $\mathbf{a}_i$. For $j \in [1..n]$, define $\mathbf{a}^*_j \stackrel{\text{def}}{=} (a_{(1,j)}, a_{(2,j)}, \ldots, a_{(m,j)})$. By $[\mathbf{a}]$, we denote the n-vector $(\mathbf{a}^*_1, \mathbf{a}^*_2, \ldots, \mathbf{a}^*_n)$ representing the sharing of $\mathbf{a}$ such that party i holds $\mathbf{a}^*_i$.

Protocol 5.17 is an efficient protocol for computing the inner product of two m-vectors $\mathbf{a}$ and $\mathbf{b}$ representing fixed-point numbers. The protocol uses a similar optimization principle as that in Protocol 2.2.

Protocol 5.17: $[d] \leftarrow \mathsf{FPInner}([\mathbf{a}], [\mathbf{b}], e, f)$

```
1 [c] ← Inner([a], [b]);                    // 1 rnd, 1 inv (Z_q)
2 [d] ← Truncate([c], 2e + 2f, f);          // See below
3 return [d];
```

Here Truncate $\in \{$Trunc, TruncF, TruncPr, TruncPrF, TruncPrN, TruncApp$\}$. We use the notation FPInner, FPInnerF, FPInnerPr, FPInnerPrF, FPInnerPrN and FPInnerApp respectively to denote the variants. The complexity is given in Table 11.

The security analysis of the above variants is similar to that of the FPMul variants.

5.4.3 Fixed-Point Reciprocal

Basic Newton-Raphson algorithm. Given an equation $y = f(x)$, we can obtain an approximation of a root of $f(x)$ as follows: we start with an initial approximation r_0 and iteratively improve it by computing $r_{n+1} = r_n - f(r_n)/f'(r_n)$. To use this technique to find the reciprocal of a, we set $f(x) = 1/a - x$. We then obtain a sequence $r_0, r_1, \ldots$ given by $r_{n+1} = r_n(2 - ar_n)$ with the property that if $0 < r_0 < 2/a$ then the sequence converges quadratically to $1/a$.

In our scenario, a is a signed fixed-point number in the range $[-2^e, 2^e - 2^{-f}]$ with resolution f (represented as the signed integer $2^f a \in [-2^{e+f}..2^{e+f} - 1]$) and we want to compute the reciprocal $1/a$ with the same resolution (again represented as an integer) assuming no overflow. In order to find an initial approximation we perform an initial scaling of a to ensure that $0.5 < a < 1$. The final result is obtained by a corresponding scaling of the reciprocal in the reverse direction.

Initial Scaling. A key issue is to determine an initial approximation that ensures quadratic convergence. Moreover, an important performance gain can be obtained by starting with an accurate initial approximation, in order to reduce the number of iterations. The usual approach is to compute the normalized input $r \in [0.5, 1)$ (or $r \in [1, 2)$) and then find an approximation of $1/r$. A linear approximation $1/r \approx \alpha \cdot r + \beta$ can easily be computed and offers good accuracy. For example, $x_0 = 2.9142 - 2r$ approximates $r \in [0.5, 1)$ with error $\epsilon_0 < 0.08578$, i.e., accuracy of $m = 3.5$ bits. For a secret normalized value r this linear approximation is computed without interaction.

More accurate initial approximations can be obtained by table lookup. For example, a piece-wise linear approximation using a lookup table with 2^k entries offers initial approximations with accuracy of $m = 2k + 2$ bits [21]. A reciprocal with 64-bit accuracy can thus be computed in 2 Newton-Raphson iterations, with an initial approximation based on a table with only 128 entries.

Reciprocal Protocol. Protocol 5.18 takes as input a secret-shared value $[d]$, where $d = \mathbf{Fld}_q(\mathbf{Int}_f(d^*))$ for some $d^* \in [2^{-f}, 2^e - 2^{-f}]$ and returns the secret-shared result $[z]$ where $z = \mathbf{Fld}_q(\mathbf{Int}_f(z^*))$ for some $z^* \approx 1/d^*$.

The protocol uses the simple linear approximation shown above. Secure table lookup can be achieved quite efficiently, and piece-wise linear approximation will be included in a future version of the protocol. However, the complexity of the reciprocal protocol is dominated by scaling, so reducing the number of iterations will have limited effects.

Protocol 5.18: $[z] \leftarrow \mathsf{RecNR}([d], e, f, p)$

Input: $([d], e, f, p)$ s.t. $d = \mathbf{Fld}_q(\mathbf{Int}_f(d^*))$ for some fixed point number $d^* \in [2^{-f}, 2^e - 2^{-f}]$ and p is an accuracy parameter.
Output: $[z]$ s.t. $z - \mathbf{Fld}_q(\mathbf{Int}_f(z^*))$ for fixed point number $z^* \in [2^{-f}, 2^e - 2^{-f}]$ with $z^* \approx 1/d^*$

1 $(\theta, \ell, h, j) \leftarrow ((e+f),\ \lceil \log \frac{p}{m} \rceil,\ (e+f-p),\ (p+e-f))$;
2 $[v] \leftarrow \mathsf{ScaleUpFactor}([d], \theta)$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$ // 4 rnd†, 4θ inv, 2θ exp $(\mathbb{Z}_{q_1})$ // $1 + 2\log\theta$ rnd, $\theta + \frac{3\theta}{2}\log\theta$ inv $(\mathbb{F}_{2^8})$
3 $[r] \leftarrow [d][v]$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
4 **if** $(e + f - p > 0)$ **then**
5 $[r] \leftarrow \mathsf{TruncPrF}([r], e+f, h)$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$ // 2 rnd†, $2f$ inv, f exp $(\mathbb{Z}_{q_1})$
6 $[x] \leftarrow \alpha - \beta[r]$;
7 $\gamma \leftarrow \mathbf{Fld}_q(\mathbf{Int}_{2p}(2.0))$;
8 **foreach** $i \in [1..\ell]$ **do sequential**
9 $[y] \leftarrow \gamma - [x][r]$; // ℓ rnd, ℓ inv $(\mathbb{Z}_q)$
10 $[y] \leftarrow [x][y]$; // ℓ rnd, ℓ inv $(\mathbb{Z}_q)$
11 $[x] \leftarrow \mathsf{TruncPrF}([y], 3+3p, 2p)$; // ℓ rnd, ℓ inv $(\mathbb{Z}_q)$ // 2ℓ rnd†, $2\ell f$ inv, ℓf exp $(\mathbb{Z}_{q_1})$
12 $[z] \leftarrow [x][v]$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
13 $[z] \leftarrow \mathsf{TruncPrF}([z], p+e+f, j)$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$ // 2 rnd†, $2f$ inv, f exp $(\mathbb{Z}_{q_1})$
14 **return** $[z]$;

Protocol 5.19: $[v] \leftarrow \mathsf{ScaleUpFactor}([d], k)$

Input: $([d], k)$ s.t. $d \in [0..2^k - 1]$
Output: $[v]$ s.t. $v = 2^u \wedge vd \in [2^{k-1}..2^k - 1]$

1 $([d_{k-1}]^{\mathbb{F}_{2^8}}, \ldots, [d_0]^{\mathbb{F}_{2^8}}) \leftarrow \mathsf{BitDecF}([d], k, k)$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$ // 2 rnd†, $2k$ inv, k exp $(\mathbb{Z}_{q_1})$ // $\log k$ rnd, $k \log k$ inv $(\mathbb{F}_{2^8})$
2 $([a_{k-1}]^{\mathbb{F}_{2^8}}, \ldots, [a_0]^{\mathbb{F}_{2^8}}) \leftarrow \mathsf{PreOpL}(\mathsf{OR}, [d_{k-1}]^{\mathbb{F}_{2^8}}, \ldots, [d_0]^{\mathbb{F}_{2^8}})$; // $\log k$ rnd, $\frac{k}{2}\log k$ inv $(\mathbb{F}_{2^8})$
3 **foreach** $i \in [0..k-2]$ **do**
4 $[b_i]^{\mathbb{F}_{2^8}} \leftarrow [a_i]^{\mathbb{F}_{2^8}} - [a_{i+1}]^{\mathbb{F}_{2^8}}$;
5 $[b_{k-1}]^{\mathbb{F}_{2^8}} \leftarrow [a_{k-1}]^{\mathbb{F}_{2^8}}$;
6 **foreach** $i \in [0..k-1]$ **do parallel**
7 $[b_i] \leftarrow \mathsf{BitF2MtoZQ}([b_i]^{\mathbb{F}_{2^8}})$; // 2 rnd†, $2k$ inv, k exp $(\mathbb{Z}_{q_1})$ // 1 rnd, k inv $(\mathbb{F}_{2^8})$
8 $[v] \leftarrow \sum_{i=0}^{k-1} [b_{k-1-i}] \cdot 2^i$;
9 **return** $[v]$;

Optimization with Precomputation: We describe a variant of the above protocols with precomputation of random bits that reduces the round complexity while keeping the

remaining parameters same. These rounds are indicated with the superscript †. The following steps in RecNR allow precomputation:

1. Step 2 - ScaleUpFactor (see below) - saving of 4 rounds in $\mathbb{Z}_{q_1}$.
2. Step 5 - Use of TruncPrFPre instead of TruncPrF - saving of 2 rounds in $\mathbb{Z}_{q_1}$.
3. Step 11 - Use of TruncPrFPre instead of TruncPrF - saving of 2ℓ rounds in $\mathbb{Z}_{q_1}$.
4. Step 13 - Use of TruncPrFPre instead of TruncPrF - saving of 2 rounds in $\mathbb{Z}_{q_1}$.

   In ScaleUpFactor:

5. Step 1 - Use of BitDecFPre instead of BitDecF - saving of 2 rounds in $\mathbb{Z}_{q_1}$.
6. Step 7 - Use of BitF2MtoZQPre instead of BitF2MtoZQ - saving of 2 rounds in $\mathbb{Z}_{q_1}$.

Overall this results in a saving of $6 + 2\ell$ rounds in $\mathbb{Z}_{q_1}$ considering 2 rounds for precomputation. We call this variant RecNRPre.

Correctness: The protocol uses the Newton-Raphson method discussed above. Signed inputs are handled by a simple extension of this protocol, requiring a secure integer comparison and two secure multiplications. A detailed correctness proof is available in the deliverable D3.1 [27]. The main steps of the computation are explained below:

1. Range reduction (initial scaling): Scale the input $d^* \in [2^{-f}, 2^e - 2^{-f}]$ to the range $[1/2, 1)$. Let $r^* = 2^s \cdot d^* \in [1/2, 1)$ the scaled value.
2. Iterations: Let $x_0^* \in (1, 2]$ an initial approximation of $1/r^*$ with $\epsilon_0 \leq 2^{-m}$. Compute $x_{i+1}^* = x_i^*(2 - x_i^* \cdot r^*)$, for $0 \leq i \leq c$. The number of iterations depends on the desired accuracy and the initial approximation error. If $\epsilon_0 \leq 2^{-m}$ an accuracy of p bits is obtained after $c = \lceil \log(p/m) \rceil$ iterations.
3. Range expansion (final scaling): After the iterations we obtain $x_c^* \approx 1/r^* = 1/(d^* \cdot 2^s)$. Scale x_c^* to obtain the (approximate) reciprocal $1/d^* \approx x_c^* \cdot 2^s$.

The normalized input $r^* \in [1/2, 1)$ is a fixed-point number with resolution 2^{-p}. The protocol RecNR obtains r^* as follows. The input $\mathbf{Int}_f(d^*)$ is first scaled up by a secret factor 2^u, obtaining the integer $r' = 2^u \cdot \mathbf{Int}_f(d^*) \in [2^{e+f-1}, 2^{e+f} - 1]$. If $p = e + f$, then $\mathbf{Int}_p(r^*) = r'$. If $p < e + f$, then r' is divided by the public value 2^{e+f-p} to obtain $\mathbf{Int}_p(r^*) = 2^{-(e+f-p)}r' \in [2^{p-1}, 2^p - 1]$. Observe that $r^* \approx 2^{u-e}d^*$. The secret factor 2^u is computed using Protocol 5.19, ScaleUpFactor.

The Newton-Raphson iterations start with the normalized input $r^* \approx 2^{u-e}d^*$ and the initial approximation x_0^*, and compute $x_c^* \in (1, 2]$ such that $x_c^* \approx 1/r^*$ with accuracy 2^{-p}. The final scaling computes $\mathbf{Int}_f(z^*) = 2^u 2^{-(p+e-f)}\mathbf{Int}_p(x_c^*) \approx 2^{f+u-e}(1/r^*) \approx \mathbf{Int}_f(1/d^*)$.

We assume $e = f$ to avoid overflow for the entire range of the input. The parameter $p \in [f, e + f]$ allows a trade-off between the accuracy of the output and efficiency. If $p = e + f = 2f$, the accuracy is limited by the resolution of the fixed-point numbers, 2^{-f}. If $p < 2f$, then the input is truncated by cutting off the least significant $2f - u - p$ bits, i.e., keeping p bits after the most significant non-zero bit. If $p = f$ the truncation of the input introduces an error $< 2^{-f}$.

On the other hand, p determines the bit-length of the values during iterations and the number of iterations. The efficiency gain obtained by taking $p < 2f$ depends on how the Newton-Raphson iterations are computed (including the variant of truncation protocol) and the bit-length of the input. Taking $p = f$ offers the best tradeoff. Different variants for computing the Newton-Raphson iterations are analyzed in D3.1 [27].

Security: The protocol provides statistical privacy. It does not output any value and all the building blocks are protocols with perfect or statistical privacy.

5.5 Performance Measurements

The protocols presented in this chapter were gradually included in JSMC, a collection of Java libraries for secure computation using secret sharing developed in Work Package 9. The main purpose of JSMC is to enable fast prototyping, testing, and analysis of protocol components. The current JSMC libraries include most of the current protocols and their building blocks described in Chapter 3. We summarize in this section the results of experiments with some of the main components of secure Simplex.

JSMC follows the simple computation model described in Section 2.3, and the parties are processes running on different PCs, with full mesh interconnection topology. We measured the running time of the protocols for 5 parties. We used a heterogeneous group of PCs, equipped with Intel Pentium 4HT at 2.8 GHz or Intel Core Duo at 1.8 GHz.

The experiments were carried out in an isolated network, for two settings: LAN with 100 Mbps and WAN with 10 Mbps links. The end-to-end delay of the WAN was 16 ms without traffic and about 50 ms (average) during the computation. The purpose of the LAN experiments is to determine an upper bound for the protocol performance, which corresponds to a network with high bandwidth and low delay. On the other hand, WAN experiments show how the performance degrades when the network delay increases and the bandwidth decreases.

We show the running time for the protocols LTZF (LAN tests in Figure 4 and WAN tests in Figure 5), TruncPrF (LAN tests in Figure 6 and WAN tests in Figure 7, and RecNR (LAN tests in Figure 8 and WAN tests in Figure 9

Each protocol was tested for three parameter settings relevant for our applications, corresponding to fixed-point arithmetic with $e = f = 32$, $e = f = 48$, and $e = f = 64$. Note that the running time of these protocols is essentially determined by the bit-length of the data values, rather than the size of the modulus q.

We measured the running time for a single protocol instance and for batches of 10, 50, and 100 instances executed in parallel. Observe that the running time per instance decreases rapidly with the size of the batch, so an important performance gain is obtained by parallel execution. As these protocols are relatively complex, the gain levels off at about 50 instances in our test environments. For a single instance, the running time is dominated by the network delay (reflected by the round complexity), and for many parallel instances by the amount of transmitted data and local computation (communication and computation complexity).

All these protocols need an important amount of shared random bits, which can be precomputed. We measured for each protocol the running time after precomputation, besides the total time. The tests show that the generation of shared bits has an important contribution to the running time, even with the relatively efficient protocols PRandBitD

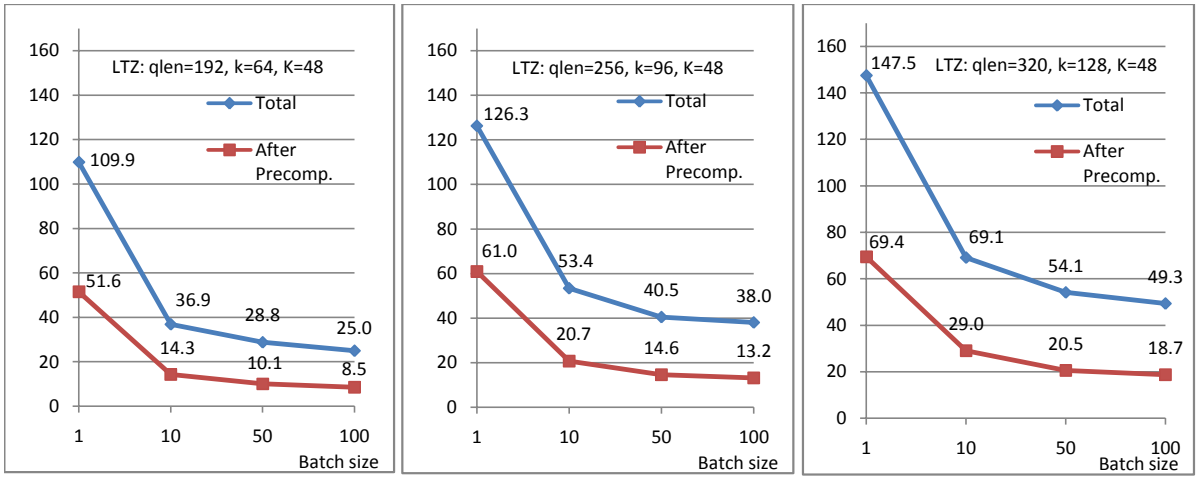

Figure 4: Running time of the protocol LTZF in LAN (millisec.).

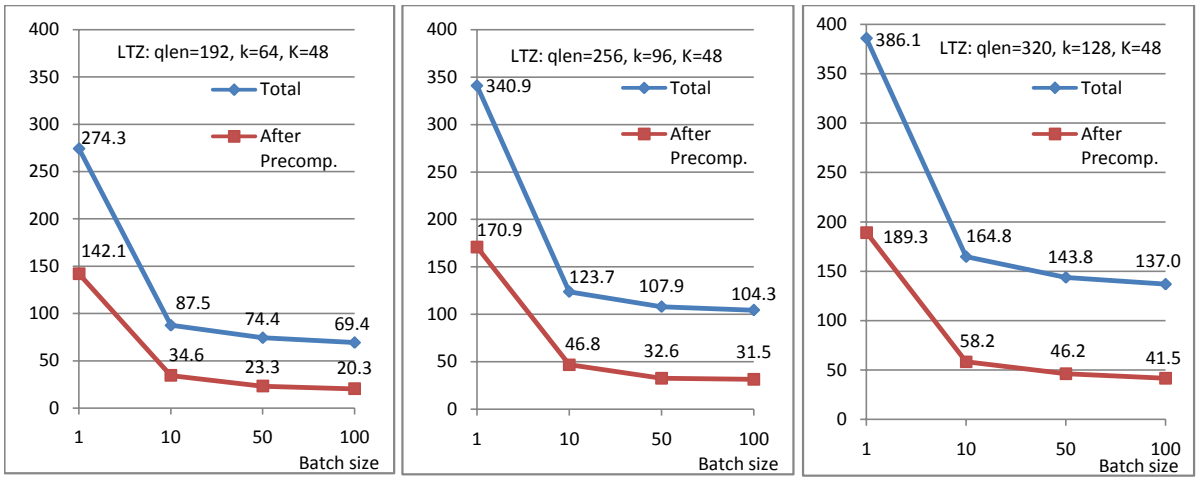

Figure 5: Running time of the protocol LTZF in WAN (millisec.).

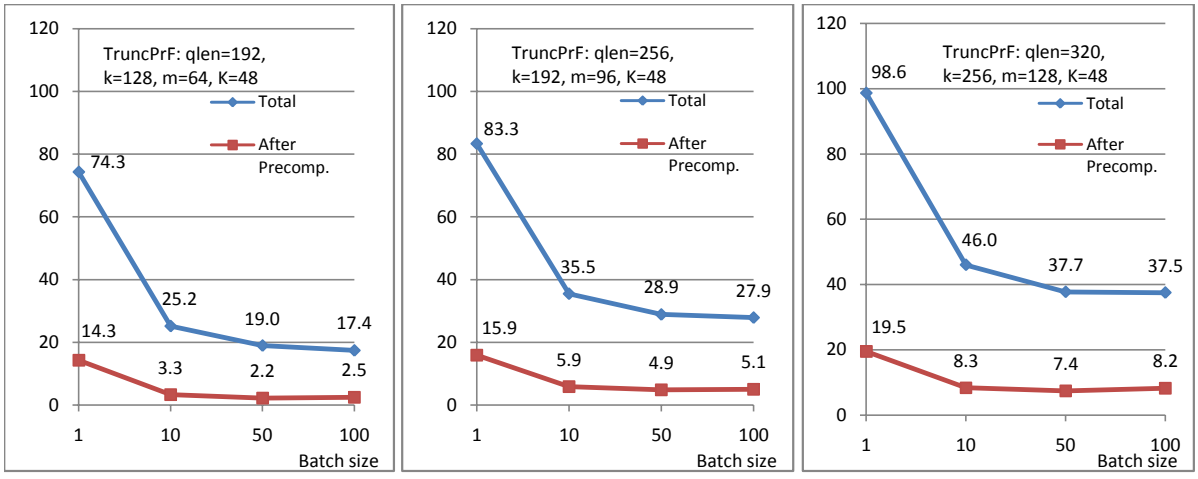

Figure 6: Running time of the protocol TruncPrF in LAN (millisec.).

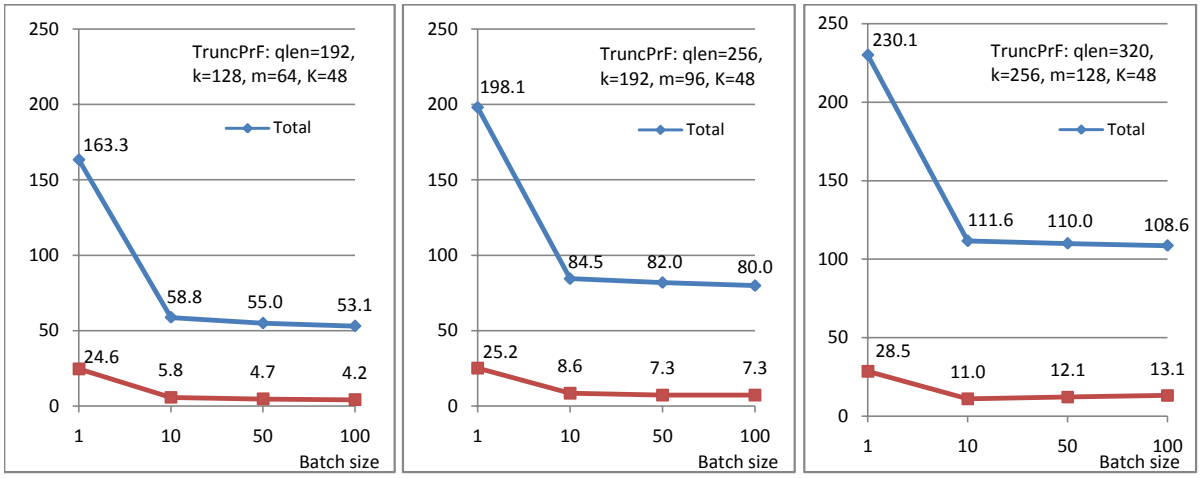

Figure 7: Running time of the protocol TruncPrF in WAN (millisec.).

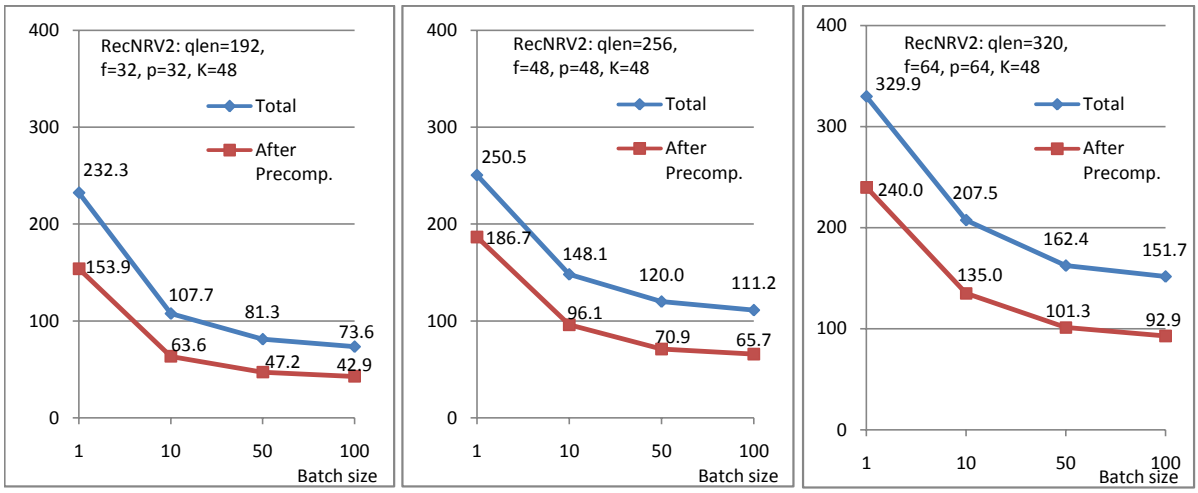

Figure 8: Running time of the protocol RecNR (using TruncPrN) in LAN (millisec.).

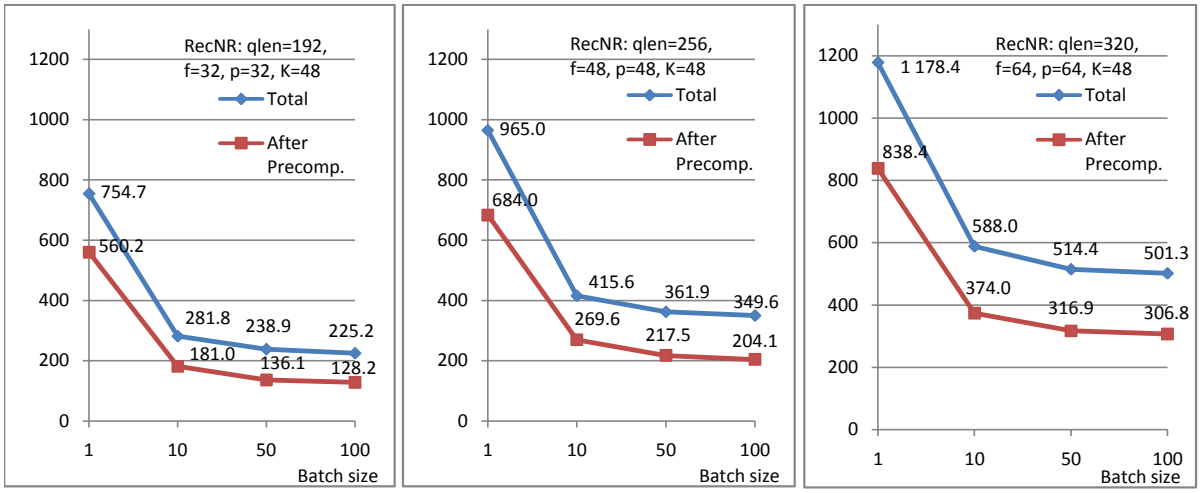

Figure 9: Running time of the protocol RecNR (using TruncPrN) in WAN (millisec.).

and PRandBitL. On the other hand, precomputation can offer an important performance gain (e.g., for the comparisons used in the Simplex protocols, see Chapter 6).

Data encoding tradeoffs. We carried out additional experiments in order to study various data encoding tradeoffs. Several relevant results are shown in Figure 10 for LAN, and in Figure 11 for WAN.

We tested the following variants of the LTZ protocol:

1. Variant with all data encoded in the same field $\mathbb{Z}_q$, where the bit-length of q is determined by the bit-length k of the encoded integers. This variant is Protocol 5.9. The shared random bits are generated in $\mathbb{Z}_q$ using PRandBit.

2. Variant with efficient random bit generation in $\mathbb{Z}_{q_1}$ ($|q_1| = 64$) using PRandBitL and bits encoded in $\mathbb{Z}_q$ for BitLT. We call this variant LTZL.

3. Variant with efficient random bit generation in $\mathbb{Z}_{q_1}$ ($|q_1| = 64$) using PRandBitD and efficient bit encoding in $\mathbb{F}_{2^8}$ for BitLT. This variant is Protocol 5.10, LTZF.

Each variant was tested for four parameter settings relevant for our applications, allowing integer of fixed-point arithmetic with: $k = 64$ ($k = 2f$); $k = 128$ ($k = 2f$ or $k = 4f$); $k = 192$ ($k = 4f$); $k = 256$ ($k = 4f$).

The tests show that the performance of PRandBit degrades rapidly with the growth of q and k, due to the increase of its computation and communication complexity. For $k \geq 128$, PRandBit is much slower than PRandBitD and PRandBitL, although these protocols need an additional round for share conversion.

Similarly, computation with bits encoded in $\mathbb{Z}_q$ becomes slower than with bits encoded in $\mathbb{F}_{2^8}$, although the latter needs an additional round for share conversion. The gain obtained by encoding the bits in $\mathbb{F}_{2^8}$ is larger in WAN (due to lower bandwidth).

The other protocols presented in this chapter are constructed using same building blocks and techniques as LTZ (except TruncPrN), so the data encoding has similar effects on their running time.

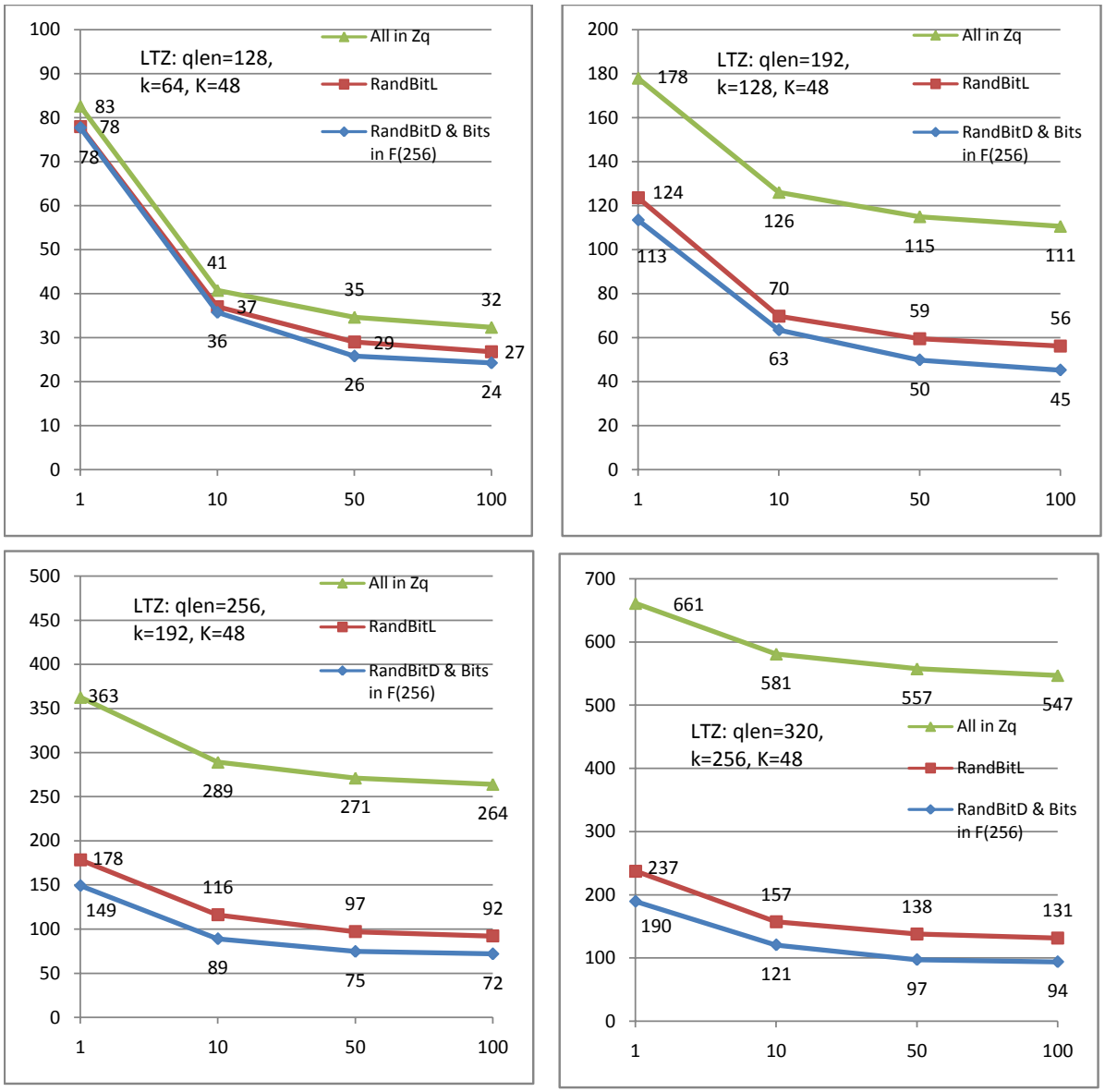

Figure 10: Data encoding tradeoffs: LAN tests with LTZ (time in millisec.).

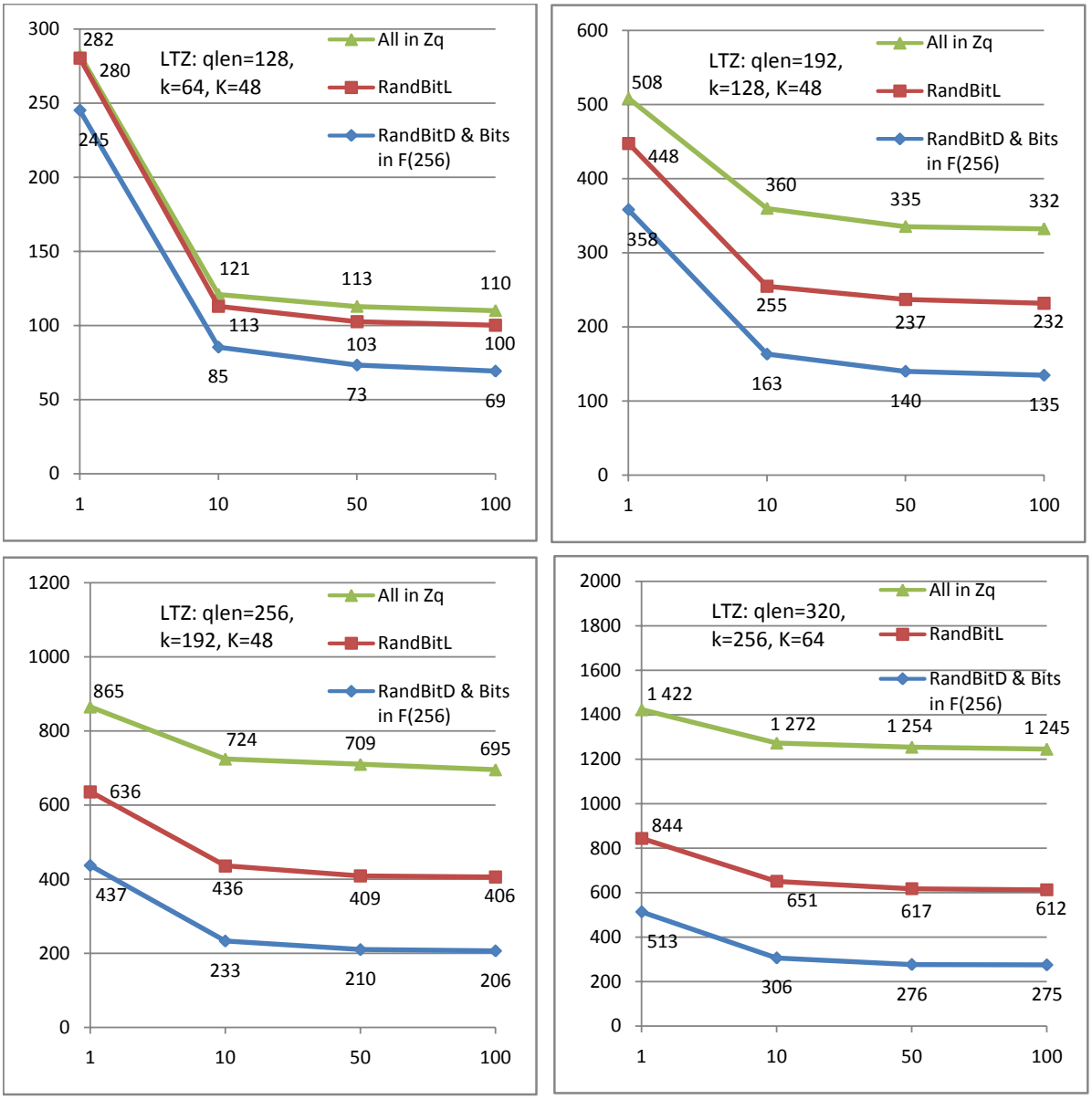

Figure 11: Data encoding tradeoffs: WAN tests with LTZ (time in millisec.).

5.6 Summary

Tables 10-12 summarize the complexity of the protocols discussed in this chapter.

Protocol	Field	Rounds	Invocations	Exp.
Mod2m$([a], k, m)$	$\mathbb{Z}_q$	$2 + \log m$	$3m - 1$	m
Mod2mF$([a], k, m)^*$	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	1 2 $1 + \log m$	1 $2m + 2$ $2m - 1$	- $m + 1$ -
LSB$([a], k)$	$\mathbb{Z}_q$	2	2	1
Trunc$([a], k, m)$	as in Mod2m			
TruncF$([a], k, m)^*$	as in Mod2mF			
TruncPr$([a], k, m)$	$\mathbb{Z}_q$	2	$m + 1$	m
TruncPrPre$([a], k, m, [r'])$	$\mathbb{Z}_q$	1	1	-
TruncPrF$([a], k, m)^*$	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$	1 2	1 $2f$	- f
TruncPrFPre$([a], k, m)^*$	$\mathbb{Z}_q$	1	1	-
TruncPrN$([a], k, m)^\dagger$	$\mathbb{Z}_q$	1	1	-
TruncApp$([a], k, m)$	$\mathbb{Z}_q$	3	4	-
EQZ$([a], k)$	$\mathbb{Z}_q$	$2 + \log k$	$2k$	k
EQZF$([a], k)^*$	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	1 2 $\log k$	1 $2k$ $k - 1$	- k -
EQPub$([a], [b])$	$\mathbb{Z}_q$	1	1	-
LTZ$([a], k)$	$\mathbb{Z}_q$	$2 + \log(k - 1)$	$3k - 4$	$k - 1$
LTZF$([a], k)^*$	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	1 2 $1 + \log(k - 1)$	1 $2k$ $2k - 3$	- k -
GTZF$([a], k)$	as in LTZF			
BitDec$([a], k, m)$	$\mathbb{Z}_q$	$2 + \log m$	$m + 1 + m \log m$	m
BitDecF$([a], k, m)^*$	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	1 2 $\log m$	1 $2m$ $m \log m$	- m -
BitDecFPre$([a], k, m)^*$	$\mathbb{Z}_q$ $\mathbb{F}_{2^8}$	1 $\log m$	1 $m \log m$	- -

* implies that $q_1 \ll q$
† implies weaker (than statistical) security.

Table 10: Complexity of integer arithmetic protocols in this chapter.

Protocol	Field	Rounds	Invocations	Exponentiations
FPMul$([a],[b],e,f)$	$\mathbb{Z}_q$	$3+\log f$	$3f$	f
FPMulF$([a],[b],e,f)^*$	$\mathbb{Z}_q$	2	2	-
	$\mathbb{Z}_{q_1}$	2	$2f+2$	$f+1$
	$\mathbb{F}_{2^8}$	$1+\log f$	$2f-1$	-
FPMulPr$([a],[b],e,f)$	$\mathbb{Z}_q$	3	$f+2$	-
FPMulPrF$([a],[b],e,f)^*$	$\mathbb{Z}_q$	2	2	-
	$\mathbb{Z}_{q_1}$	2	$2f$	f
FPMulPrN$([a],[b],e,f)^\dagger$	$\mathbb{Z}_q$	2	2	-
FPMulApp$([a],[b],e,f)$	$\mathbb{Z}_q$	4	5	-
FPInner$([\mathbf{a}],[\mathbf{b}],e,f)$	as in FPMul			
FPInnerF$([\mathbf{a}],[\mathbf{b}],e,f)^*$	as in FPMulF			
FPInnerPr$([\mathbf{a}],[\mathbf{b}],e,f)$	as in FPMulPr			
FPInnerPrF$([\mathbf{a}],[\mathbf{b}],e,f)$	as in FPMulPrF			
FPInnerPrN$([\mathbf{a}],[\mathbf{b}],e,f)^\dagger$	as in FPMulPrN			
FPInnerApp$([\mathbf{a}],[\mathbf{b}],e,f)$	as in FPMulApp			

* implies that $q_1 \ll q$
† implies weaker (than statistical) security.

Table 11: Complexity of fixed-point arithmetic protocols in this chapter.

Protocol	Fld.	Rounds	Invocations	Exp.
RecNR$([d],e,f,p)^*$	$\mathbb{Z}_q$	$5+3\lceil\log\frac{p}{m}\rceil$	$5+3\lceil\log\frac{p}{m}\rceil$	-
	$\mathbb{Z}_{q_1}$	$8+2\lceil\log\frac{p}{m}\rceil$	$4e+2f(4+\lceil\log\frac{p}{m}\rceil)$	$f(3+\lceil\log\frac{p}{m}\rceil)+e$
	$\mathbb{F}_{2^8}$	$1+2\log(e+f)$	$\frac{e+f}{2}(2+3\log(e+f))$	-
RecNRPre$([d],e,f,p)^*$	$\mathbb{Z}_q$	$5+3\lceil\log\frac{p}{m}\rceil$	$5+3\lceil\log\frac{p}{m}\rceil$	-
	$\mathbb{Z}_{q_1}$	2	$4e+2f(4+\lceil\log\frac{p}{m}\rceil)$	$f(3+\lceil\log\frac{p}{m}\rceil)+e$
	$\mathbb{F}_{2^8}$	$1+2\log(e+f)$	$\frac{e+f}{2}(2+3\log(e+f))$	-

Table 12: Complexity of RecNR.

6 Linear Programming Protocols

Outline. This chapter presents protocols for secure linear programming (LP) using the Simplex method. Deliverable D3.1 [27] recommends several variants of Simplex suitable for secure computation. For example, ST-RP (Small Tableau, Rational Pivoting) follows the classical Simplex algorithm and uses arithmetic with fixed-point rational numbers. On the other hand, ST-IP (Small Tableau, Integer Pivoting) uses integer arithmetic and avoids rounding errors. ST-IP uses simpler protocols for secure multiplication and division, but the values in the tableau grow during the iterations and can reach thousands of bits. Moreover, it is not clear how to choose $\mathbb{Z}_q$ such that to minimize the communication overhead and at the same time avoid overflow of the secure arithmetic operations, since we do not have a tight upper bound for the values in the tableau.

We reviewed the protocols proposed in D3.1, in order to check their completeness (identify missing components) and analyze their correctness, complexity, and security. The pivot selection protocols are similar in all variants, while the protocols for updating the tableau are different. We selected for analysis the ST-RP variant. The variants based on integer pivoting are simpler and use a subset of the building blocks needed by ST-RP.

Section 6.2.1 introduces the secret indexing protocols used to hide the operations performed on the Simplex tableau. Section 6.2.2 presents a complete specification of the ST-RP protocol, with additional protocols and revised protocol descriptions (consistent with the building blocks in the previous chapters). Moreover, we give a more efficient solution for the selection of the pivot row. The protocols were implemented in Java and tested. Section 6.3 gives a summary of the performance measurements.

Notation. In this chapter, n refers to the number of variables in the LP problem, and not to the number of parties in the secure computation as was done in earlier chapters.

6.1 Linear Programming Using ST-RP Simplex

Linear Programming Problem: A Linear Programming (LP) maximization problem in *canonical form* is defined as follows: given n variables x_j, for $1 \leq j \leq n$, and m inequalities $\sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i$, for $1 \leq i \leq m$, maximize a function $f = \sum_{j=1}^{n} f_j \cdot x_j$ where $f_j, a_{i,j}, b_i, x_j \in \mathbb{R}$ and $x_j \geq 0$. The m inequalities are called the *constraints* and f is called the *objective function*. Any set of variables satisfying the constraints is called a *feasible solution* and the set of variables maximing f is called the *optimal solution* (if it exists). If there are infinitely many solutions, then the problem is said to be *unbounded*. If the is no solution, the problem is said to be *infeasible*. In this work we only consider problems which are unbounded or have an optimal solution. In other words, we require every $b_i \geq 0$.[7]

Simplex Algorithm. The most suitable algorithm to solve LP problems in our context is Simplex [16]. The algorithm starts by transforming the constraints to equalities by adding m "slack" variables x_{n+i}, for $1 \leq i \leq m$. It then selects an initial feasible solution and iteratively improves it until either an optimum is reached (if it exists), or the problem is found to be unbounded. At each improvement, the problem is rewritten in an equivalent form and the various constraints are stored in a matrix. The process of improving the

[7] WP3 will develop protocols for solving arbitrary LP problems.

solution is called pivoting. In the literature there are several variations of Simplex (i.e., different sizes of matrices and different pivoting rules). We consider the variant called Small-Tableau Rational-Pivoting (ST-RP). The following discussion gives a very brief overview relevant for our purpose. For details see [27, 23, 16, 30].

In ST-RP Simplex, there are the following data structures. A $m \times n$ matrix A, a m vector B, a n vector F, a variable Z, a n vector U and a m vector S. See Figure 12.

$$\begin{pmatrix} A(1,1) & \dots & A(1,n) \\ \vdots & \ddots & \vdots \\ A(m,1) & \dots & A(m,n) \\ F(1) & \dots & F(n) \\ U(1) & \dots & U(n) \end{pmatrix} \quad \begin{pmatrix} B(1) \\ \vdots \\ B(m) \end{pmatrix} \quad Z \quad \begin{pmatrix} S(1) \\ \vdots \\ S(m) \end{pmatrix}$$

Figure 12: Global data structures in ST-RP simplex.

1. *Initialization:* For $1 \le i \le m$ and $1 \le j \le n$, set $A(i,j) \leftarrow a_{i,j}$, $F(j) \leftarrow -f_j$, $B(i) \leftarrow b_i$, $Z \leftarrow 0$, $U(j) \leftarrow j$, $S(i) \leftarrow n+i$.

2. *Repeat Forever:*

   (a) *Get Pivot Column:* Select $c \in [1..n]$ such that $F(c) < 0$. If no such c, report "Optimal Solution" and exit. If more options, choose at random or using Bland's rule (minimum $U(c)$).

   (b) *Get Pivot Row:* Select $r \in [1..m]$, such that $A(r,c) > 0$ and $|B(r)/A(r,c)|$ is minimal. If no such r, report "Unbounded Problem" and exit. If more options, choose at random or using Bland's rule (minimum $S(r)$).

   (c) *Update the tableau (pivoting):* Set $T \leftarrow \begin{pmatrix} A & B \\ F & Z \end{pmatrix}$ then do the following:

   $$\begin{array}{lcll} C(i) & \leftarrow & T(i,c) & 1 \le i \le m+1 \\ R(j) & \leftarrow & T(r,j) & 1 \le j \le n+1 \\ p & \leftarrow & R(c) & \\ T(i,j) & \leftarrow & T(i,j) - \frac{C(i) \cdot R(j)}{p} & 1 \le i \le m+1, i \ne r, 1 \le j \le n+1 \\ T(i,c) & \leftarrow & -C(i)/p & 1 \le i \le m+1 \\ T(r,j) & \leftarrow & R(j)/p & 1 \le j \le n+1 \\ T(r,c) & \leftarrow & 1/p & \\ U(c) & \leftrightarrow & S(r) & \text{(swap)} \end{array}$$

   Decompose T back into A, B, F, Z.

3. *Final solution:* For $1 \le i \le m$ variable $x_{S(i)}$ takes the value $B(i)$. All other variables take the value 0. The objective function takes the value Z.

Discussion. The set of m variables pointed to by the vector S is called the *basis*, while the remaining n variables (i.e., those pointed to by the vector U) is called the *co-basis*. Initially the basis contains all (and only) the slack variables. At each iteration, the problem is first checked to see if the solution can be improved. If so, the solution **is** improved by

rewriting the problem in an equivalent form and swapping a variable from the basis and co-basis. The process is repeated until the existing solution cannot be improved or the problem is found to be unbounded.

6.2 Secure Linear Programming Using ST-RP Simplex

Notation. Arrays are named with capital letters $(A, B, \ldots)$ and indexes are enclosed within round brackets (). Elements of an array A with n elements start from index 1 and are written successively as $A(1), A(2), \ldots, A(n)$. For any array $A = (A(1), A(2), \ldots, A(n))$, we denote by $[A]$ the array $([A(1)], [A(2)], \ldots, [A(n)])$. Multidimensional arrays are handled similarly. For convenience, bitmask vectors will be represented using boldface letters $(\mathbf{A}, \mathbf{B}, \ldots)$.

6.2.1 Secret Indexing

The secure Simplex protocol hides the operations performed on the LP tableau using the secret indexing method proposed in [30]. Let $[A]$ be an array of secret-shared values, using Shamir sharing. Given a secret index $[v]$, we want to read (or write) $[A(v)]$ without revealing the value $A(v)$ and the index v. The element at secret index $[v]$ is selected using an array of secret binary values $[\mathbf{V}]$ (bitmask), so that $[\mathbf{V}(i)] = [0]$ for $i \neq v$ and $[\mathbf{V}(v)] = [1]$. The lengths of the secret bitmask and the vector are usually equal. We allow bitmasks shorter than the vectors.

Working with an array. Protocols 6.1 and 6.2 allow secret reading and writing from/to a vector. The secure multiplications re-randomize the shares, providing perfect privacy.

Protocol 6.1: $[s] \leftarrow \mathsf{SecRead}([A], [\mathbf{V}])$

Input: m-vector $[A]$, m-vector $[\mathbf{V}]$.
Output: value $[s]$.
1 $[s] \leftarrow \mathsf{Inner}([A], [\mathbf{V}])$; `// 1 rnd, 1 inv (`$\mathbb{Z}_q$`)`
2 **return** $[s]$;

In Protocol 6.1, one of $\{[s], [A]\}$ may be replaced by a public value. If the output is public, we can use a variation of Protocol Inner with public output based on the technique of Protocol 3.11. In this case the number of rounds and invocations remains unchanged. If $[A]$ is replaced by a public value then the protocol requires no interaction.

Protocol 6.2: $[A] \leftarrow \mathsf{SecWrite}([A], [\mathbf{V}], [s])$

Input: m-vector $[A]$, m-vector $[\mathbf{V}]$, and a value $[s]$
Output: m-vector $[A]$ (overwritten)
1 **foreach** $i \in [1..m]$ **do parallel**
2 $\quad [A(i)] \leftarrow [A(i)] + [\mathbf{V}(i)]\,([s] - [A(i)])$; `// 1 rnd, `$m$` inv (`$\mathbb{Z}_q$`)`
3 **return** $[A]$;

In Protocol 6.2, $[s]$ can be replaced by a public value. The complexity remains same.

Working with a matrix. Protocols 6.3, 6.4, 6.5, and 6.6 are used for secretly reading/writing rows/cols of matrix. The principle is similar to that in Protocols 6.1 and 6.2.

Protocol 6.3: $[R] \leftarrow \mathsf{SecReadRow}([T], [\mathbf{V}])$

Input: $(m+1) \times (n+1)$-matrix $[T]$, m-vector $[\mathbf{V}]$.
Output: $(n+1)$-vector $[R]$.

1 **foreach** $i \in [1..n+1]$ **do parallel**
2 **foreach** $j \in [1..m]$ **do** $[X(j)] \leftarrow [T(j,i)]$;
3 $[R(i)] \leftarrow \mathsf{SecRead}([X], [\mathbf{V}])$; // 1 rnd, $n+1$ inv ($\mathbb{Z}_q$)
4 **return** $[R]$;

Protocol 6.4: $[C] \leftarrow \mathsf{SecReadCol}([T], [\mathbf{W}])$

Input: $(m+1) \times (n+1)$-matrix $[T]$, n-vector $[\mathbf{W}]$.
Output: $(m+1)$-vector $[C]$.

1 **foreach** $i \in [1..m+1]$ **do parallel**
2 **foreach** $j \in [1..n]$ **do** $[X(j)] \leftarrow [T(i,j)]$;
3 $[C(i)] \leftarrow \mathsf{SecRead}([X], [\mathbf{W}])$; // 1 rnd, $m+1$ inv ($\mathbb{Z}_q$)
4 **return** $[C]$;

In Protocols 6.3, 6.4 if $[T]$ is replaced by a public value then the protocol requires no interaction.

Protocol 6.5: $[T] \leftarrow \mathsf{SecWriteRow}([T], [\mathbf{V}], [R])$

Input: $(m+1) \times (n+1)$-matrix $[T]$, m-vector $[\mathbf{V}]$, $(n+1)$-vector $[R]$
Output: $(m+1) \times (n+1)$-matrix $[T]$ (overwritten)

1 **foreach** $i \in [1..n+1]$ **do parallel**
2 $[T(*,i)] \leftarrow \mathsf{SecWrite}(T(*,i), [\mathbf{V}], [R(i)])$; // 1 rnd, $m(n+1)$ inv ($\mathbb{Z}_q$)
3 **return** $[T]$;

Protocol 6.6: $[T] \leftarrow \mathsf{SecWriteCol}([T], [\mathbf{W}], [C])$

Input: $(m+1) \times (n+1)$-matrix $[T]$, n-vector $[\mathbf{W}]$, $(m+1)$-vector $[C]$
Output: $(m+1) \times (n+1)$-matrix $[T]$ (overwritten)

1 **foreach** $i \in [1..m+1]$ **do parallel**
2 $[T(i,*)] \leftarrow \mathsf{SecWrite}([T(i,*)], [\mathbf{W}], [C(i)])$; // 1 rnd, $n(m+1)$ inv ($\mathbb{Z}_q$)
3 **return** $[T]$;

Similar to Protocol 6.2, both $[R]$ and $[C]$ in Protocols 6.5 and 6.6 respectively can be replaced by a vector of public values. The complexity remains same.

6.2.2 Secure ST-RP Simplex Protocol

Protocol 6.7, STRPSimplex, computes the iterations of the ST-RP algorithm. The computation is structured into several sub-protocols, which select the pivot (column and row) and then update the tableau $[T]$ and the vectors $[S]$ and $[U]$.

Protocol 6.7: $([T], [S], \mathsf{Result}) \leftarrow \mathsf{STRPSimplex}([T])$

Input: $(m+1) \times (n+1)$-matrix $[T]$
Output: $(m+1) \times (n+1)$-matrix $[T]$, m-vector $[S]$, $\mathsf{Result} \in \{\mathsf{Opt}, \mathsf{Unb}\}$

1 $([S], [U]) \leftarrow \mathsf{InitVar}(m, n)$;
2 **repeat forever**
3 $[\mathbf{V}] \leftarrow \mathsf{GetPivCol}([T])$; // see Table 15
4 **if** *Null*$([\mathbf{V}])$ **then return** $([T], [S], \mathsf{Opt})$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
5 $[C] \leftarrow \mathsf{SecReadCol}([T], [\mathbf{V}])$; // 1 rnd, $m+1$ inv $(\mathbb{Z}_q)$
6 $([\mathbf{W}], [\mathbf{D}]) \leftarrow \mathsf{GetPivRow}([B], [C])$; // see Table 15
7 **if** *Null*$([\mathbf{D}])$ **then return** $([T], [S], \mathsf{Unb})$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
8 $[R] \leftarrow \mathsf{SecReadRow}([T], [\mathbf{W}])$; // 1 rnd, $n+1$ inv $(\mathbb{Z}_q)$
9 $[p] \leftarrow \mathsf{SecRead}([R], [\mathbf{V}])$; // 1 rnd, n inv $(\mathbb{Z}_q)$
10 $[T] \leftarrow \mathsf{STRPUpdTab}([T], [C], [R], [\mathbf{V}], [\mathbf{W}], [p])$; // see Table 15
11 $([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [\mathbf{V}], [\mathbf{W}])$; // 2 rnd, $2m+2$ inv $(\mathbb{Z}_q)$

The input to the protocol, $[T]$ is a $(m+1) \times (n+1)$ matrix representing the matrices A, B, F, Z as defined above. The matrix $[T]$ is assumed to already exist in the system. Deliverable 3.2 will discuss methods to construct this matrix from user data.

The following sections discuss the building blocks used in Protocol 6.7.

Maintaining Variable States. Protocol 6.8 (InitVar) creates the secret-shared variables $[S]$ and $[U]$ with initial public values. Protocol 6.9, UpdVar, updates $[S]$ and $[U]$ in each iteration to reflect the current sets of basic and non-basic variables.

Protocol 6.8: $([S], [U]) \leftarrow \mathsf{InitVar}(m, n)$

Input: integers m, n
Output: m-vector $[S]$, n-vector $[U]$

1 $[z] \leftarrow \mathsf{PRandZero}(\mathbb{Z}_q)$;
2 **foreach** $j \in [1..n]$ **do** $[U(j)] \leftarrow [z] + j$;
3 **foreach** $i \in [1..m]$ **do** $[S(i)] \leftarrow [z] + i + n$;
4 **return** $([S], [U])$;

Protocol 6.9: $([S], [U]) \leftarrow \mathsf{UpdVar}([S], [U], [\mathbf{V}], [\mathbf{W}])$

Input: m-vectors $[S]$ and $[\mathbf{V}]$, n-vectors $[U]$ and $[\mathbf{W}]$
Output: m-vector $[S]$, n-vector $[U]$ (overwritten)

1 **do parallel**
2 $[s] \leftarrow \mathsf{SecRead}([S], [\mathbf{V}])$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
3 $[u] \leftarrow \mathsf{SecRead}([U], [\mathbf{W}])$; // 1 inv $(\mathbb{Z}_q)$
4 **do parallel**
5 $[S] \leftarrow \mathsf{SecWrite}([S], [\mathbf{V}], [u])$; // 1 rnd, m inv $(\mathbb{Z}_q)$
6 $[U] \leftarrow \mathsf{SecWrite}([U], [\mathbf{W}], [s])$; // m inv $(\mathbb{Z}_q)$
7 **return** $[S]$, $[U]$;

Testing for Null Bitmask. Protocol 6.10 tests if a bitmask $[\mathbf{V}]$ is null and outputs a public value. It assumes that at most one bit in the bitmask is set. If the bitmask may contain multiple bits set, a variation using Protocol 5.13 (EQPub) must be used.

Protocol 6.10: $s \leftarrow \mathsf{Null}([\mathbf{V}])$

Input: ℓ-vector $[\mathbf{V}]$.
Output: public bit s.

1 $[v] \leftarrow [\mathbf{V}(1)]$;
2 **foreach** $i \in [2..\ell]$ **do**
3 $\quad [v] \leftarrow [v] + [\mathbf{V}(i)]$;
4 $s \leftarrow \mathsf{Output}([v])$; // 1 rnd, 1 inv ($\mathbb{Z}_q$)
5 **return** s;

Pivot Selection. Protocol 6.11, GetPivCol, determines the index c of the pivot column as a secret bitmask denoted $[\mathbf{V}]$. If none of the values in F is negative, then $[\mathbf{V}]$ is null ($\mathbf{V}(i) = 0$, $1 \leq i \leq n$). The Simplex protocol has found the optimal solution and terminates. Termination is detected by checking if $[\mathbf{V}]$ is null using the protocol Null.

Protocol 6.11: $[\mathbf{V}] \leftarrow \mathsf{GetPivCol}([T])$

Input: $(m+1) \times (n+1)$-matrix $[T]$.
Output: n-vector $[\mathbf{V}]$.

1 **foreach** $i \in [1..n]$ **do parallel**
2 $\quad [\mathbf{D}(i)] \leftarrow \mathsf{LTZF}([T(m+1,i)], k)$; // n parallel LTZFs
3 $[\mathbf{D}'] \leftarrow \mathsf{PreOpL}(\mathsf{OR}, [\mathbf{D}])$; // $\log n$ rnd, $\frac{n}{2}\log n$ inv ($\mathbb{Z}_q$)
4 $[\mathbf{V}(1)] \leftarrow [\mathbf{D}'(1)]$;
5 **foreach** $i \in [2..n]$ **do**
6 $\quad [\mathbf{V}(i)] \leftarrow [\mathbf{D}'(i)] - [\mathbf{D}'(i-1)]$;
7 **return** $[\mathbf{V}]$;

Protocol 6.12, GetPivRow, determines the index r of the pivot row as a secret bitmask denoted $[\mathbf{W}]$. If none of the values in C is strictly positive, then $[\mathbf{V}]$ is null. The Simplex protocol terminates and reports unbounded problem.

Protocol 6.12: $([\mathbf{W}], [\mathbf{D}]) \leftarrow \mathsf{GetPivRow}([B], [C])$

Input: m-vectors $[B]$, $[C]$.
Output: m-vector $[\mathbf{W}]$.

1 **foreach** $i \in [1..m]$ **do parallel**
2 $\quad [\mathbf{D}(i)] \leftarrow \mathsf{GTZF}([C(i)])$; // m parallel GTZFs
3 $\quad [C^*(i)] \leftarrow [C(i)][\mathbf{D}(i)]$; // 1 rnd, m inv ($\mathbb{Z}_q$)
4 $\quad [B^*(i)] \leftarrow [B(i)] + 1 - [\mathbf{D}(i)]$;
// C^* = copy of C with non-app entries set to zero
// B^* = copy of B with non-app entries set to non-zero
5 $[\mathbf{W}] \leftarrow \mathsf{MinCons}([B^*], [C^*], m)$; // $(5 + \log(k-1)) \log m$ rnd,
// $m(3k+5)$ inv, $m(k-1)$ exp ($\mathbb{Z}_q$)
6 **return** $([\mathbf{W}], [\mathbf{D}])$;

The bitmask $[\mathbf{W}]$ is computed by the protocol MinCons, using the protocol CompCons as comparison operator.

Protocol 6.13 [30] computes the secret bitmask of the minimum value in a vector with m elements in $\log(m)$ steps using $m-1$ comparisons. The number of comparisons

and hence the communication and computation complexity are optimal. Protocols that compute the minimum in less steps (and hence less rounds) can be obtained using standard techniques [19], as shown in [30]. These techniques are less efficient for secure computation, because the number of comparisons is substantially larger and the comparison protocols are relatively complex. However, a limited performance gain is still possible. This issue is left for further study.

Protocol 6.13: $[\mathbf{W}] \leftarrow \mathsf{MinCons}([B^*], [C^*], \alpha)$

```
   /* Assume α is a power of 2 */
1  if α = 1 then
2      return [1];
3  else
4      foreach i ∈ [1..α/2] do parallel
5          [A_0] ← ([B*(2i − 1)], [C*(2i − 1)]);
6          [A_1] ← ([B*(2i)], [C*(2i)]);
7          [W(i)] ← CompCons([A_0], [A_1]);          // 3 + log(k − 1) rnd,
                                                     // (α/2)(3k + 2) inv,
                                                     // (α/2)(k − 1) exp (Z_q)
8          do parallel
                                                     // 1 rnd, α inv (Z_q)
9              [B*_new(i)] ← [W(i)]([B*(2i − 1)] − [B*(2i)]) + [B*(2i)];
10             [C*_new(i)] ← [W(i)]([C*(2i − 1)] − [C*(2i)]) + [C*(2i)];
11     [W_new] ← MinCons([B*_new], [C*_new], α/2);   // (5 + log(k − 1)) log(α/2) rnd,
                                                     // (3k + 5)(α − 1) inv,
                                                     // (k − 1)(α − 1) exp (Z_q)
12     foreach i ∈ [1..α/2] do parallel
13         [W(2i)] ← [W(i)][W_new(i)];               // 1 rnd, (α/2) inv (Z_q)
14         [W(2i − 1)] ← [W_new(i)] − [W(2i)];
15     return [W];
```

Protocol 6.14: $[\mathbf{z}] \leftarrow \mathsf{CompCons}([A_0], [A_1])$

```
1  parse [A_0] as ([b*_0], [c*_0]);
2  parse [A_1] as ([b*_1], [c*_1]);
3  [x] ← [b*_0][c*_1] − [b*_1][c*_0];                // 1 rnd, 2 inv (Z_q)
4  return LTZ([x], k);          // 2 + log(k − 1) rnd, 3k inv, k − 1 exp (Z_q)
```

Table 13 gives the truth-table for Protocol 6.14 (CompCons). b_0, b_1, c_0, c_1 are the original constraints corresponding to $b_0^*, b_1^*, c_0^*, c_1^*$ respectively.

Faster CompCons for STRP Simplex. For fixed-point numbers, the comparison in protocol 6.14 can be computed more efficiently with protocol 6.15 than with an integer comparison protocol. The idea is to compute $x = b_0^* c_1^* - b_1^* c_0^*$, then truncate x to obtain $y = \lfloor x/2^f \rfloor + u$, where $u \in \{0, 1\}$, and then obtain the sign of y as $s = -\lfloor y/(2^{k-1}) \rfloor$.

Protocol 6.15 determines the sign of the input by combining the methods used in the Protocols Trunc, TruncPr, and LTZ (Protocols 5.4, 5.6 and 5.9). The initial truncation by f bits is achieved without interaction. This truncation reduces the bit-length of the

$c_0 > 0$	$c_1 > 0$	c_0^*	c_1^*	$b_0^* \cdot c_1^* < b_1^* \cdot c_0^*$	Output	Constraints	Selection
0	0	0	0	$0 < 0$	0	Not applicable	$\frac{b_1}{c_1}$
0	1	0	c_1	$b_0 c_1 < 0$	0	$\frac{b_0}{c_0}$ not applicable	$\frac{b_1}{c_1}$
1	0	c_0	0	$0 < b_1 c_0$	1	$\frac{b_1}{c_1}$ not applicable	$\frac{b_0}{c_0}$
1	1	c_0	c_1	$b_0 c_1 < b_1 c_0$	0	$\frac{b_1}{c_1} \leq \frac{b_0}{c_0}$	$\frac{b_1}{c_1}$
					1	$\frac{b_0}{c_0} < \frac{b_1}{c_1}$	$\frac{b_0}{c_0}$

Table 13: Truth table for Protocol 6.14 (CompCons).

inputs of the protocol BitLT, eliminating $2f$ bit multiplications and one round (we assume $f \geq 48$ bits). Also, the f-bit random value r' can be locally generated using PRand2mN (instead of PRandBit), when weaker privacy is acceptable.

Protocol 6.15: FPLTZ$([a], k, f)$

1 **foreach** $i \in [0..k+f-1]$ **do parallel**
2 $\quad [r_i] \leftarrow$ PRandBit(q); // 1 rnd, $k+f$ inv, $k+f$ exp $(\mathbb{Z}_q)$
3 $[r'] \leftarrow \sum_{i=0}^{f-1} 2^i \cdot [r_i]$;
4 $[r''] \leftarrow \sum_{i=f}^{k+f-1} 2^i \cdot [r_i]$;
5 $[r'']_B) \leftarrow (r_{k+f-1}, r_{k+f-2}, \ldots, r_f)$;
6 $[r'''] \leftarrow$ PRandInt(κ);
7 $[r] \leftarrow 2^{k+f} \cdot [r'''] + 2^f \cdot [r''] + [r']$;
8 $[b] \leftarrow 2^{k+f-1} + [a]$;
9 $c \leftarrow$ Output$([b] + [r])$; // 1 rnd, 1 inv $(\mathbb{Z}_q)$
10 $c' \leftarrow c \bmod 2^f$;
11 $[a'] \leftarrow c' - [r']$;
12 $[d] \leftarrow ([a] - [a'])2^{-f}$;
13 $c'' \leftarrow c/2^f \bmod 2^{k-1}$;
14 $[u] \leftarrow$ BitLT$(c'', [r'']_B)$; // $\log(k-1)$ rnd, $2k-4$ inv $(\mathbb{Z}_q)$
15 $[d'] \leftarrow c'' - [r''] + [u] \cdot 2^{k-1}$;
16 $[e] \leftarrow ([d] - [d'])2^{-(k-1)}$;
17 **return** $-[e]$;

The modified protocol FPCompCons is given below.

Protocol 6.16: $[\mathbf{z}] \leftarrow$ FPCompCons$([A_0], [A_1])$

1 parse $[A_0]$ as $([b_0^*], [c_0^*])$;
2 parse $[A_1]$ as $([b_1^*], [c_1^*])$;
3 $[x] \leftarrow [b_0^*][c_1^*] - [b_1^*][c_0^*])$; // 1 rnd, 2 inv $(\mathbb{Z}_q)$
4 **return** FPLTZ$([x], k, f)$; // $2 + \log f$ rnd, $3f - 1 + k$ inv, $k+f$ exp $(\mathbb{Z}_q)$

Correctness. The protocol maps $x \in [-2^{k+f-1}..2^{k+f-1}-1]$ to $b \in [0..2^{k+f-1}]$ by computing $b = (2^{k+f-1} + a) \bmod q = 2^{k+f-1} + x$. Observe that $b \bmod 2^m = x \bmod 2^m$ for any $0 < m < k+f$. Next, it generates a random secret $r \in [0..2^{k+f+\kappa}-1]$ and reveals $c = (b+r) \bmod q$. Since $q > 2^{k+f+\kappa+1}$ we have $q > b + r$ and hence $c = b + r$.

Let $c' = c \bmod 2^f$, $b' = b \bmod 2^f$, and $r' = r \bmod 2^f$. We see that $c' = (b' + r') \bmod 2^f$ and $b' = c' - r' + u' \cdot 2^f$, where $u' = 1$ if $c' < r'$ and $u' = 0$ if $c' \geq r'$. Steps 10-11 compute the value $a' = (b' - u' \cdot 2^f) \bmod q = ((x \bmod 2^f) - u' \cdot 2^f) \bmod q$. Step 12 computes

$d = (\lfloor x/2^f \rfloor + u') \bmod q$.

Let $c'' = \lfloor c/2^f \rfloor \bmod 2^{k-1}$, $b'' = \lfloor b/2^f \rfloor \bmod 2^{k-1}$, and $r'' = \lfloor r/2^f \rfloor \bmod 2^{k-1}$. We see that $b'' = \lfloor x/2^f \rfloor \bmod 2^{k-1}$. Observe that $c'' = (b'' + r'' + u') \bmod 2^{k-1}$ and hence $(b'' + u') \bmod 2^{k-1} = c'' - r'' + u \cdot 2^{k-1}$, where $u = 1$ if $c'' < r''$ and $u = 0$ if $c'' \geq r''$.

Steps 13-15 compute $d' = (b'' + u') \bmod 2^{k-1}$. If $b'' < 2^{k-1} - 1$ then $d' = b'' + u' = (\lfloor x/2^f \rfloor \bmod 2^{k-1}) + u'$ and the protocol returns the correct output, $e = (d - d')2^{-(k-1)} \bmod q = (\lfloor x/2^f \rfloor + u' - (\lfloor x/2^f \rfloor \bmod 2^{k-1}) - u')2^{-(k-1)} \bmod q = \lfloor x/2^{k+f-1} \rfloor \bmod q$. If $u' = 1$ and $b'' = 2^{k-1} - 1$ then $d' = 0$ and the output may be incorrect. This occurs in two cases: $\lfloor x/2^f \rfloor = 2^{k-1} - 1$ or $\lfloor x/2^f \rfloor = -1$. These errors are negligible. Let y denote the fixed-point rational number corresponding to the integer $\lfloor x/2^f \rfloor$. The first case corresponds to $y = 2^{k-1} - 2^{-f}$ (largest positive value) and the second case to $y = -2^{-f}$ (negative value closest to 0).

Update of the Tableau. Protocol 6.17 updates the LP tableau. Fixed-point multiplications consist of integer multiplications followed by truncation using TruncPrN. We can truncate f bits at each multiplication or $2f$ bits for two multiplications. Protocol 6.17 uses the second method. Intermediate values are at least f bits larger in this case, but the number of truncations is only $(m + 1)(n + 1)$ (in parallel). The protocol RecNR computes the reciprocal of the pivot using Newton-Raphson.

Protocol 6.17: $[T] \leftarrow$ STRPUpdTab$([T], [C], [R], [V], [W], [p])$

Input: $(m + 1) \times (n + 1)$-matrix $[T]$, $(m + 1)$-vector $[C]$, $(n + 1)$-vector $[R]$, n-vector $[\mathbf{V}]$, m-vector $[\mathbf{W}]$, value $[p]$.

Output: $(m + 1) \times (n + 1)$-matrix $[T]$ (overwritten).

1 $[s] \leftarrow$ RecNR$([p], e, f, f)$; // see Table 12
2 **foreach** $j \in [1..n + 1]$ **do parallel**
3 $\quad [R(j)] \leftarrow [R(j)][s]$; // 1 rnd, $n + 1$ inv $(\mathbb{Z}_q)$
4 **foreach** $i \in [1..m + 1]$ **do parallel**
5 $\quad [C'(i)] \leftarrow [C(i)]$;
6 $\quad [C(i)] \leftarrow -2^f[C(i)][s]$; // $m + 1$ inv $(\mathbb{Z}_q)$
7 SecWrite$([C], [W], [s])$; // 1 rnd, m inv $(\mathbb{Z}_q)$
8 **foreach** $i \in [1..m + 1]$, $j \in [1..n + 1]$ **do parallel**
9 $\quad [T(i, j)] \leftarrow 2^{2f}[T(i, j)] - [C'(i)][R(j)]$; // 1 rnd, $(m + 1)(n + 1)$ inv $(\mathbb{Z}_q)$
10 **foreach** $j \in [1..n + 1]$ **do**
11 $\quad [R(j)] \leftarrow 2^f[R(j)]$;
12 SecWriteRow$([T], [W], [R])$; // 1 rnd, $m(n + 1)$ inv $(\mathbb{Z}_q)$
13 SecWriteCol$([T], [V], [C])$; // 1 rnd, $n(m + 1)$ inv $(\mathbb{Z}_q)$
14 **foreach** $i \in [1..m + 1]$, $j \in [1..n + 1]$ **do parallel**
15 $\quad [T(i, j)] \leftarrow$ TruncPrN$([T(i, j)], 2f)$; // 1 rnd, $(m + 1)(n + 1)$ inv $(\mathbb{Z}_q)$
16 **return** $[T]$;

A variant that truncates once per fixed-point multiplication can work in a prime field with smaller modulus and only needs $m + n + 2$ additional truncations (in parallel). However, this also affects the protocol RecNR, increasing the number of rounds.

6.3 Tests and Performance Analysis

We tested the secure Simplex protocols using JSMC, the Java libraries of secure computation building blocks developed in Work Package 9. The prototypes follow the revised specifications of the Simplex protocols given in this chapter. The JSMC libraries used for this purpose include virtually all the building blocks presented in the deliverable.

The aim of the tests was to check the correctness and completeness of the improved Simplex protocol specifications and the new building blocks, and to analyze their performance. However, the Java code was gradually developed after protocol analysis and revision, leaving time only for preliminary tests, with linear programs of limited size.

The tests showed that the solutions provided by the secure Simplex protocols are virtually the same as those obtained by usual (non-secure) implementations of the same algorithms. Full tests (until termination of the Simplex algorithm) were performed with the ST-RP protocol for linear programs up to $m = 60$ and $n = 50$, and partial tests for larger sizes, $m, n < 200$. For the ST-IP protocol correctness tests were limited to small linear programs ($m = n = 10$), since accuracy is not an issue and the duration increases rapidly with the size of the linear program.

We summarize in the following the results of the performance measurements. We recall that the main goal of JSMC is to support the development and analysis of secure computation building blocks. JSMC follows the simple computation model described in Section 2.3. This simplifies the analysis of the contribution of different complexity metrics and building blocks and the effects of different optimization solutions. On the other hand, the scheduling and communication libraries are not fully optimized for multiprocessor computers and very large LP problems. A fully optimized implementation can achieve substantial performance gains with respect to the results presented below.

We measured the running time of the Simplex protocols for 5 parties, using the same test environment as for the arithmetic protocols (Section 5.5). Each party runs on a different PC, with full mesh interconnection topology. The PCs are equipped with Intel Pentium 4HT at 2.8 GHz or Intel Core Duo at 1.8 GHz. The experiments were carried out in an isolated network, for two settings: Ethernet LAN with 100 Mbps links and WAN with 10 Mbps links. The end-to-end delay of the WAN was 16 ms without any traffic and about 50 ms (average) during the computation.

Experiments with Simplex ST-RP. Table 14 gives the running time of an iteration of Protocol 6.7, STRPSimplex, and its main components. The table shows how the running time increases with the size of the linear programs, for $m = n = 10$, $m = n = 50$, $m = n = 100$, where m is the number of constraints and n is the number of variables. All tests used fixed-point numbers with $e = f = 48$ bits, encoded in $\mathbb{Z}_q$ with modulus length $\ell = 256$ bits and security parameter $\kappa = 56$ bits.

An iteration is split into a precomputation phase followed by the actual Simplex computation steps. The precomputation generates in parallel all the shared random bits necessary during the iteration, using the protocols PRandBitD and PRandBitL. Each iteration needs $(n+m)(e+f)+(m-1)(e+2f)+(e+f)$ double shared bits and $n+e+f$ bits shared in $\mathbb{Z}_q$, for comparisons and the reciprocal of the pivot. Figure 13 shows the running time after precomputation versus the total duration of an iteration. The precomputation takes about $40\%-50\%$ of the total time. An implementation that carries out the precomputation for iteration $i+1$ in parallel with the computation for iteration

LAN (100 Mbps), $\ell = 256$, $f = 48$	$m = n = 10$	$m = n = 50$	$m = n = 100$
Precomputation	1.02	4.24	7.81
Select pivot column	0.24	0.65	1.15
Select pivot row	0.53	1.53	2.65
Update tableau	0.32	1.57	4.84
Total iteration	2.10	7.99	16.45
After precomputation	1.09	3.75	8.64
WAN (10 Mbps), $\ell = 256$, $f = 48$	$m = n = 10$	$m = n = 50$	$m = n = 100$
Precomputation	2.82	13.41	26.49
Select pivot column	0.57	1.65	3.01
Select pivot row	1.45	4.31	7.20
Update tableau	1.20	8.06	28.77
Total iteration	6.04	27.42	65.46
After precomputation	3.22	14.02	38.97

Table 14: Running time (seconds) for Simplex ST-RP and its main components.

i can reduce the running time of an iteration to (roughly) the time we measured after precomputation.

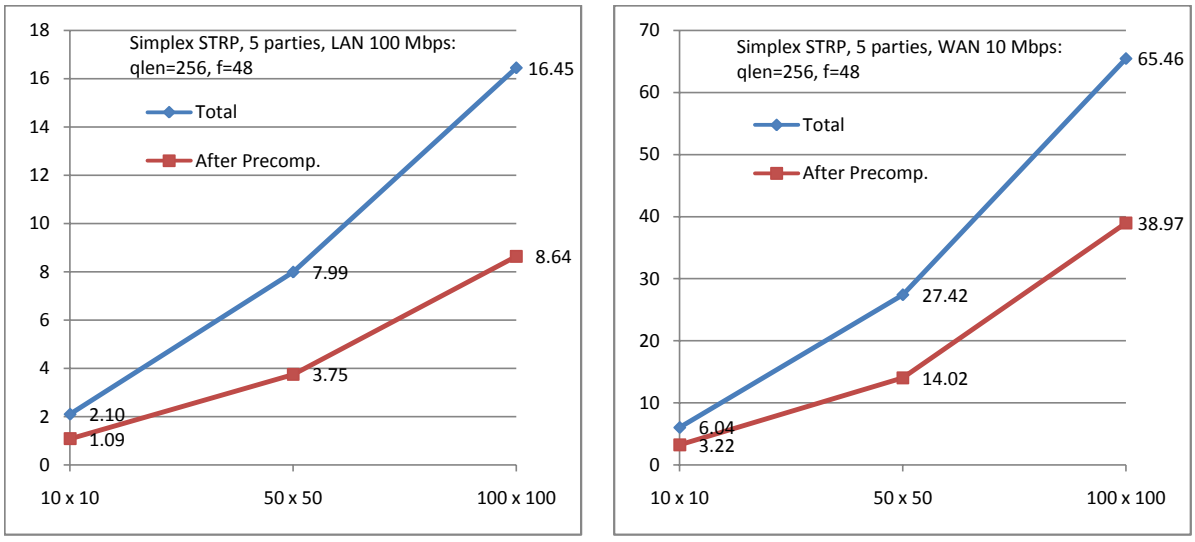

Figure 13: Running time (seconds) for the Simplex ST-RP protocol.

Simplex ST-RP versus Simplex ST-IP. Current tests show that ST-RP is much faster than ST-IP for the current comparison protocol, whose running time depends on the bit-length of the inputs (LTZF, Protocol 5.10). For the improved pivot selection protocols presented in Section 6.2.2, ST-RP computes all the comparisons with inputs on $k = e + f$ bits, and $k \leq 128$ is sufficient to obtain accurate solutions for large problems. On the other hand, k can reach several thousands of bits in ST-IP.

This handicap can be eliminated only by a comparison protocol whose complexity does not depend on the bit-length of the inputs. However, a protocol that meets this complexity requirement and at the same time satisfies standard security notions is not available. The

alternative is to find a comparison protocol with relaxed security requirements, sufficient for this particular application. However, if such a protocol is found, ST-RP and ST-IP will have similar running time for pivot selection, and the difference will come from the update of the tableau. Current tests suggest that ST-RP could still perform better, but the difference will depend on the network delay and bandwidth (the amount of data sent by ST-RP during the update of the tableau is about 10 times smaller, but it needs 30 rounds more for the reciprocal of the pivot).

6.4 Summary

Table 15 summarizes the complexity of protocols discussed in this chapter. In the table,t m is the number of constraints and n is the number of variables. Note that n does not denote the number of parties. The inputs and outputs are assumed to be as described in the protocols.

Protocol	Fld.	Rounds	Invocations	Exp.
SecRead	$\mathbb{Z}_q$	1	1	-
SecWrite	$\mathbb{Z}_q$	1	m	-
SecReadRow	$\mathbb{Z}_q$	1	$n+1$	-
SecReadCol	$\mathbb{Z}_q$	1	$m+1$	-
SecWriteRow	$\mathbb{Z}_q$	1	$m(n+1)$	-
SecWriteCol	$\mathbb{Z}_q$	1	$n(m+1)$	-
InitVar	$\mathbb{Z}_q$	-	-	-
UpdVar	$\mathbb{Z}_q$	2	$2m+2$	-
Null	$\mathbb{Z}_q$	1	1	-
GetPivCol	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	$1+\log n$ 2 $1+\log(k-1)$	$n+\frac{n}{2}\log n$ $2kn$ $n(2k-3)$	- kn -
GetPivRow	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	$2+\log m\log(32k-32)$ 2 $1+\log(k-1)$	$m(3k+7)$ $2km$ $m(2k-3)$	$m(k-1)$ km -
MinCons	$\mathbb{Z}_q$	$\log(32k-32)(1+\log\frac{m}{2})$	$\frac{m}{2}(9k+15)-3k-5$	$\frac{3m-2}{2}(k-1)$
CompCons	$\mathbb{Z}_q$	$3+\log(k-1)$	$3k+2$	$k-1$
FPCompCons	$\mathbb{Z}_q$	$3+\log f$	$3f+1+k$	$k+f$
STRPUpdTab†	$\mathbb{Z}_q$ $\mathbb{Z}_{q_1}$ $\mathbb{F}_{2^8}$	$11+3\alpha$ 2 $1+2\log(e+f)$	$5+3\alpha$ $4e+2f(4+\alpha)$ $\frac{e+f}{2}(2+3\log(e+f))$	- $f(3+\alpha)+e$ -

† $\alpha=\lceil\log\frac{f}{m}\rceil$

Table 15: Complexity of protocols in this chapter.

7 Conclusion

7.1 Summary

The protocols presented in Deliverable D3.1 [27] are based on standard cryptographic tools, methods, and models, and provide adequate security in the semi-honest model. On the other hand, they rely on general building blocks from Deliverable D9.1 [26] (found in the literature), that do not take into account the specific requirements of our applications. After reviewing D3.1 we found the following issues:

1. Many of the building blocks in D9.1 become bottlenecks when used in the complex protocols specified in D3.1, such as secure Simplex. We needed other design solutions for several building blocks, with better performance for our particular applications.

2. The main factors that degrade the performance of the protocols specified in D3.1 are the generation of shared random values, the data encoding, and the computation with bitwise-shared values. It was necessary to find more efficient solutions for data encoding and shared randoms, and to minimize binary computation.

3. The complexity metrics were not clearly defined. This led to a fuzzy analysis and difficulty in considering tradeoffs.

4. Specifications of some components referred to in D3.1 were missing.

Based on the above review, we searched for solutions that are better adapted to our applications and accomplished the following tasks:

1. Development of more efficient building blocks and techniques (based on solutions proposed in the literature and new protocols): methods for generating shared randoms with minimum interaction between parties, optimized data encoding and share conversions, improved protocols for operations with bitwise-shared values and for fixed-point arithmetic.

2. Theoretical foundation of the techniques used for constructing protocols with statistical security (Section 2.2.2).

3. Well-defined, abstract and meaningful complexity metrics (Section 2.3).

4. Complete and consistent specification of all the building blocks used in our applications, from primitives to secure arithmetic, with rigorous complexity and security analysis (Section 2.4, Chapters 3, 4, 5).

5. Complete specification of the secure Simplex ST-RP protocols (including the missing components and more efficient solutions), and analysis (Chapter 6).

6. Analysis of tradeoffs with respect to functional, security and precomputation aspects along with recommendations (e.g., Sections 5.5 and 6.3).

7. Performance measurements for the main building blocks and the secure Simplex protocols (Sections 5.5 and 6.3).

7.2 Security Analysis

Our protocols offer two categories of security: perfect and statistical (for threshold passive adversary). Many of them offer perfect security, i.e., no information is revealed about secret data. For other protocols, the security depends on a security parameter κ, which can be adjusted to a desired value. Roughly speaking, this translates to $\approx 2^{-\kappa}$ probability of the attacker guessing some information about the secret. For secure Simplex, $\kappa \in [32..64]$ is sufficient. Note that this also incurs κ bits of overhead. These are the RISS-based protocols of Chapter 3, the arithmetic protocols of Chapter 5, which hide secret values using 'one-time-pad encryption', and any application protocol using them.

We highlight several security properties of these protocols:

1. The protocols provide passive security for threshold adversary, i.e., they guarantee correctness and privacy if all parties follow the protocol and the number of dishonest parties is not larger than a specified threshold.

2. Honest majority is necessary. In particular, we require $t < n/2$, and $n \geq 3$, where n is the number of parties and t is the threshold.

3. The protocols guarantee correctness and statistical privacy for secret integers and fixed-point numbers within a specified range. Overflow is a topic of further investigation in WP3 and WP9.

7.3 Complexity and Performance Analysis

For communication complexity, we used a metric of an *invocation*, while for measuring communication time, we used a metric called *rounds*. Finally, we used a metric called *exponentiation* to measure local computation. These metrics, when defined with a corresponding field give an accurate view of the overhead in the protocol. This analysis is given in Tables 1, 2, 3, 4, 5, 6, 8, 10, 11, 12 and 15.

7.3.1 Tradeoffs

We considered the following tradeoffs in our design: gain in performance at the expense of (controlled) loss of one or more of the following: security, accuracy, correctness, and precomputation. Additionally, we considered tradeoffs related to data encoding and the general issue of alternative solutions that minimize either the number of rounds or the communication and computation complexity.

The security tradeoffs are evidenced by the category of security offered (perfect, statistical or weaker) and the corresponding complexities (i.e., invocations, rounds, etc) in different variants of the same protocol as illustrated in the tables mentioned in Section 7.3. Regarding accuracy and correctness, we made tests on the overall effect of tradeoffs in protocols for fixed-point arithmetic and reciprocal. See next section for a discussion on measurements. Following is a summary of several tradeoffs considered:

1. (Perfect versus statistical privacy) Many of the protocols presented in Chapter 5 are designed for statistical privacy in order to improve their efficiency. It is possible to construct variants of these protocols with perfect privacy, but they are substantially more complex.

2. (Precomputation) Secret random numbers can be precomputed in the beginning of the protocol to save rounds at the cost of storage (e.g., the effects of precomputation on the iterations of the secure Simplex protocol are discussed in Section 6.3).

3. (Data encoding) Encoding all the data types in the same field saves rounds but increases the communication complexity. Efficient encoding of the data types in different fields (e.g., bits and integers) reduces the communication complexity but requires additional rounds for conversions (the effects of data encoding are discussed in Section 5.5).

4. (Reciprocal) The protocol that computes the reciprocal of a fixed-point number is relatively complex. The protocol has a parameter that allows to improve its efficiency at the cost of reducing the accuracy of the output.

7.3.2 Measurements

We tested the building blocks and secure Simplex protocols using JSMC, the Java libraries of secure computation building blocks developed in Work Package 9 (Sections 5.5 and 6.3). The prototypes follow the revised specifications of protocols given in this document. The JSMC libraries used for this purpose include virtually all the building blocks presented in the deliverable.

The aim of the tests was to check the correctness and completeness of the improved protocols, and to analyze their performance. The tests showed agreement with theoretical complexity analysis. Furthermore, solutions provided by the secure Simplex protocols were in close agreement with those obtained via non-secure computation.

The comparison protocol remains an important bottleneck for secure Simplex. In the ST-RP variant the effects are considerably attenuated by precomputation and the protocol improvements presented in Section 6.2.2. On the other hand, the ST-IP variant becomes impractical for large problems.

7.4 Further Work

This deliverable has focused on establishing a foundation for the protocols developed in the project, by providing complete and consistent specifications and analysis for all building blocks, as well as the current secure Simplex protocols. A number of issues require further investigation.

The following building blocks need more efficient solutions in our current framework: (1) generation of secret random bit, (2) generation of secret random integer in range with uniform distribution, (3) comparison, and (4) reciprocal.

Secure comparison remains one of the main performance bottlenecks for secure Simplex. Further improvement of its efficiency seems to require relaxed security properties, adapted to the particular context of secure Simplex.

Our current protocols are secure in the semi-honest model, and cryptographic solutions for active security are not a realistic option for our complex applications. However, the security of these protocols does not vanish in case of (limited) protocol violations and it is necessary to clarify what are the effects of such actions on data privacy, in our applications.

Further issues to be addressed are related to the protocols with statistical privacy, e.g., the effects of overflow and the choice of the security parameter.

7.4.1 Non-Cryptographic Methods

Consider the passively secure protocols presented in this document. Assume that there is a correct implementation of these protocols under some passive adversary structure. This implementation follows a client-server approach, where the functionality of each party is implemented in a client interacting with the other clients via a trusted server whose only purpose is scheduling and synchronization of messages between the clients. A protocol implemented using this method provides passive security. In order to convert this protocol to provide active security under the same adversary structure, it is enough to ensure that the attacker is unable to alter the correct behavior of a client during the protocol execution. We discuss two non-cryptographic approaches to achieve this goal. First, note some important aspects of active and passive attacks:

1. Protection against active attacks is required only during protocol execution.
2. Observe that there is no way to ensure that the very first message of an active attacker is correct. However, assuming that the first message is correct, we can ensure that further messages sent by the attacker are consistent with prior messages (up to the first message). We call this technique *message chaining*.

The approaches we consider are given below:

1. **Tamper-resistant hardware:** The first approach is to use specialized tamper-resistant hardware that ensures only valid messages are accepted by the server, where validity is checked using the chaining technique discussed above. In order to do this, the hardware must be capable of maintaining internal state which keeps track of all outgoing messages since the first message. An emerging technology for this type of application is called *Trusted Computing* [9], where a tamper-resistant device called the trusted platform module authenticates the correct runtime behavior of a remote client using a method called *remote-entrusting* [20].
2. **Code-obfuscation:** In the second approach, we propose the use of code obfuscation.[8] The idea is to obfuscate the passively secure client to make the task of an active adversary harder - in order to launch an active attack, the adversary must now break the obfuscation. Although, it is not clear if perfectly secure obfuscators exist [3, 24], there are several commercial obfuscators available that claim to make programs significantly harder to reverse-engineer [10]. Therefore, even though obfuscators may not resist a reverse-engineering attack, they may be able to provide a time advantage. Assuming that the time between obtaining the obfuscation and time of a successful break is longer than the time for the entire computation, it is possible to do the computation with a reasonable guarantee of active security - if the attacker attempts to launch an active attack before a successful break he either risks detection or the inability to control the results of computation.

[8] Code obfuscation is best explained using the idea of a (code) obfuscator. An obfuscator $\mathcal{O}$ is a probabilistic 'compiler' that transforms a program P into $\mathcal{O}(P)$, which is a functionally equivalent version to P, yet hides certain internal details of P [3]. The idea is to make P hard to reverse-engineer.

References

[1] J. Algesheimer, J. Camenish, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO 2002*, volume 2442 of *LNCS*, pages 417–432. Springer-Verlag, 2002.

[2] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In *Proc. 8th annual ACM Symposium on Principles of distributed computing*, pages 201–209. ACM Press, 1989.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[4] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols. In *Proc. of the 22nd ACM Symposium on the Theory of Computing*, pages 503–515. Springer-Verlag, 1990.

[5] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1), 2000.

[6] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols (revised). http://eprint.iacr.org/2000/067, Dec. 2005.

[7] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In *Proc. of STOC*, pages 494–503, 2002.

[8] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[9] M. Chris. *Trusted computing*. IET, 2005.

[10] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735–746, 2002.

[11] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proc. of 2nd Theory of Cryptography Conference (TCC'05)*, pages 342–362, 2005.

[12] R. Cramer, I. Damgård, and U. Maurer. General Secure Multi-Party Computation from any Linear Secret-Sharing Scheme. In *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer-Verlag, 2000. http://www.iacr.org/archive/eurocrypt2000/1807/18070321-new.pdf.

[13] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. of 3rd Theory of Cryptography Conference (TCC'06)*, volume 3876 of *LNCS*, pages 285–304. Springer-Verlag, 2006.

[14] I. Damgård and R. Thorbek. Non-interactive Proofs for Integer Multiplication. In *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 412–429. Springer-Verlag, 2007.

[15] I. Damgard and R. Thorbek. Efficient conversion of secret-shared values between different fields. Cryptology ePrint Archive, Report 2008/221, 2008.

[16] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, August 1998.

[17] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[18] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multi-party computations with applications to threshold cryptography. In *Proc. of ACM Symposium on Principles of Distributed Computing (PODC'98)*, 1998.

[19] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[20] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[21] N. T. Masayuki Ito and S. Yajima. Efficient Initial Approximation for Multiplicative Division and Square Root by a Multiplication with Operand Modification. *IEEE Transactions on Computers*, 46(4), 1997.

[22] T. Nishide and K. Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer-Verlag, 2007.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[24] A. Saxena, B. Wyseur, and B. Preneel. Towards security notions in white-box cryptography. In *ISC'09: Proceedings of the 12th Information Security Conference*, 2009.

[25] B. Schoenmakers and P. Tuyls. Efficient Binary Conversion for Paillier Encryptions. In *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 522–537. Springer-Verlag, 2006.

[26] SecureSCM. Secure Computation Models and Frameworks. Deliverable D9.1, SecureSCM project, 2008.

[27] SecureSCM. Protocol Description V1. Deliverable D3.1, SecureSCM project, 2009.

[28] A. Shamir. How to share a secret. In *Communications of the ACM, 22(11)*, pages 612–613, 1979.

[29] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2009.

[30] T. Toft. *Primitives and Applications for Multi-party Computation*. PhD dissertation, University of Aarhus, Denmark, BRICS, Department of Computer Science, 2007.

[31] A. C. Yao. How to generate and exchange secrets. In *Proc. of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[32] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.