DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
8

TOOMAS KRIPS

Improving performance of secure real-number operations

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

8

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
8

TOOMAS KRIPS

Improving performance of secure real-number operations

UNIVERSITY OF TARTU
Press

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

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in informatics on April 17th, 2019 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Dr.	Jan Willemson Cybernetica AS
Prof. Dr.	Dominique Unruh University of Tartu

Opponents

Prof. Dr.	Claudio Orlandi Aarhus University
Prof. Dr.	Octavian Catrina Polytechnic University of Bucharest

The public defense will take place on June 12th, 2019 at 14:15 in J.Liivi 2, room 405.

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

ISSN 2613-5906
ISBN 978-9949-03-026-2 (print)
ISBN 978-9949-03-027-9 (PDF)

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

To my family and friends

ABSTRACT

Secure computation is a field that studies the problem of how to perform computations on data in such a way that the computing parties do not learn anything about the data. These computations have applications in a wide variety of areas such as medicinal research or outsourcing computation. Most of the focus concerning secure computation has been on integer-based data types. However, for many applications, real-number based data types are necessary. This thesis studies three approaches on making real-number based secure computation more efficient.

First we study an approach combining fixed-point numbers and floating point numbers, choosing the number types according to the situations where they give faster results. The efficiency of this method depends on the concrete function being computed as in some cases, switching between the data types is more expensive than the improvement we gain by using the more suitable data type for a specific operation.

Secondly, we study a method that uses the fact that parallel composition of operations is often orders of magnitude more efficient than sequential composition in some approaches for secure computation. This method improves monotone functions f for which there exists a more efficiently computable 'inverse' operation g, and we show that f can be evaluated by evaluating many instances of g in parallel. This can be more efficient than the traditional approach.

Thirdly, we introduce a new number type that we call the golden section numbers. A pair of signed integers (a,b) represents the real number $a-\varphi b$ where φ is the golden ratio. We show that we achieve results comparable to fixed-point numbers in the secure setting using this data type. As this data type is new, more improvements may be possible.

CONTENTS

LIST OF TABLES

1. INTRODUCTION

In the modern world, data is ubiquitous and analysis of existing data is highly useful. However, the owner(s) of the data and the parties that analyze data might not coincide exactly. For example, the owner of the data might have little computing capacity, and would want to outsource the computation to the cloud. He might not want to share the data with the computing agent(s), but would naturally still be interested in the result of the computation. Or there might be no single agent who possesses all the relevant data necessary for the data analysis. For instance, the relevant data could be held by several parties who are interested in the outcome of the analysis but do not want to share the data with the other parties. These requirements may sound contradictory at first, but it is in fact possible to satisfy both of these requirements and the field of solving problems in this way is called secure multiparty computation (SMC).

The field was started in 1982 by Yao, who posed the so-called Millionaire Problem [63] — two millionaires wish to know which of them has more money while leaking no more information about how much money they possess to the other party. The original problem is rather specific and its solution was very costly in both time and memory, however, after 1982 the field has become both more efficient and wider classes of problems have been studied.

Recently, several Turing-complete SMC frameworks have been developed with a variety of existing protocols and primitives. Such frameworks allow for building new desired protocols out of existing sub-protocols instead of tailoring a specific SMC solution for a problem from the beginning. Three popular SMC paradigms are secret-sharing based schemes, schemes built out of Yao's garbled circuits and schemes based on fully homomorphic cryptography. Examples of specific frameworks built on these paradigms include SPDZ [29] , Sharemind [11] , Viff [28], FairPlay [53], SCET [16], SMCR [15], SEPIA [18], TASTY [39], and $\Lambda \circ \lambda$ [27].

For these frameworks, some rather nontrivial primitives and protocols have been built. However, these primitives are usually based on either bit-typed or integer-typed values. In practice, more various types of values are needed for different types of problems. One example of such an important value type is real number types — they are so ubiquitously needed that often a special floating-point unit is built into the hardware. Recently, different real number models in SMC have been researched. This is also the focus of this thesis.

In regular, non-SMC models, floating-point numbers are often used to depict real numbers, as they provide flexibility and good precision. Floating-point numbers and protocols have been implemented in SMC settings, however, some basic operations such as addition have proven to be expensive on them [5, 47, 52].

As a different solution, fixed-point numbers have also been implemented in SMC, such as the works of Catrina and Saxena [24]. Fixed-point numbers are simpler than floating-point numbers and usually less expensive in the SMC setting, but they are quite inflexible and may not be sufficiently precise.

Thus there is no one best way for depicting real numbers in the SMC setting. The setting of SMC is rather different from the ordinary case and thus we should study how to perform various operations on real numbers there more efficiently. SMC operations are usually significantly more expensive than their nonsecure counterparts, and thus efficiency is highly important for the methods to be practically applicable.

While one strategy for constructing algorithms for SMC is taking existing algorithms and replacing the necessary primitives with corresponding privacy-preserving primitives, this strategy can often fail in that it can make the resulting SMC algorithm prohibitively expensive.

Thus various methods should be studied to find ways how to improve the efficiency of various operations on real-number computation in SMC, that utilize the special properties that SMC has. This thesis presents three such methods and they will be described in section 1.1.

1.1. Author's Contributions

In this thesis we present several techniques for improving secure computation over real numbers.

1. First, we present new techniques that use both fixed-point numbers and floating-point numbers to compute a function. This is based on the fact that fixed-point numbers and floating-point numbers have different effectiveness for different primitives and operations and thus it sometimes pays to convert the number to a different type to perform some computation. The author contributed the implementation and specifics for this method.
2. Second, we present the point-counting technique. This technique is based on the fact that in SMC, we greatly benefit from parallel composition. There are various functions for which there is an 'inverse function' that is much easier to compute than the original function, such as the pair $f(x) = \sqrt{x}, g(x) = x^2$. We present a method that is usable on fixed-point numbers in which we use many parallel instances of the 'inverse function' to compute one instance of the original function. We also show where this method is usable. The author contributed the initial idea, the analysis, the examples of its applicability, the idea how it can be used for generic polynomials and its analysis, and finally the implementation and benchmarking.
3. Third, we present a new number type, distinct from both fixed-point numbers and floating-point numbers. A pair of signed secret integers $(\llbracket a \rrbracket, \llbracket b \rrbracket)$ represents the number $a - \varphi \cdot b$ where $\varphi = 1.61\ldots$ is the golden ratio. We show that for this number type addition can be done locally, and on average, its multiplication is faster than the equivalent fixed-point number, thus making it a suitable candidate for uses in secure computation. The initial idea of using such a number type in the secure setting came from Vassil

Dimitrov, however, the author contributed the theory and practice of different normalization methods and their analysis, the analysis on why we chose φ, and the implementation of the protocols.

We claim that the methods described in this thesis can in certain models improve performance of computations based on real numbers in SMC.

2. STATE OF THE ART

Research has been conducted about secure real-valued operations. Here we give some overview on some of the related work.

2.1. Secure Fixed-Point Numbers

Fixed-point numbers have been a popular approach for representing real numbers in a secure setting. Fixed-point numbers are quite similar to integers and as many secure computation frameworks support integers, fixed-point numbers are a very natural match for secure real numbers.

More specifically, Catrina and Dragulin proposed a secure fixed-point framework in [22]. They used it to build a protocol for secure division. The secure division called a secure reciprocal as a subprotocol that used the Newton-Raphson method. This method, however, only worked for positive fixed-point numbers. Catrina and Saxena gave a framework that could support signed fixed-point numbers and a number of protocols in [24]. The proposed underlying integer primitives were improved in [21] by Catrina and Hoogh. Catrina and Hoogh also used these secure fixed-point numbers in [23] to solve linear programming problems securely. In 2012, de Hoogh defended his thesis [30] on secure linear programming which made use of secure fixed-point numbers. Additionally, Catrina introduced optimizations for fixed-point operations in [20] including a bit decomposition operation that takes not a logarithmic but constant (more precisely, 3) number of rounds.

Liedel applied that fixed-point framework to compute fixed-point square roots in [50]. Liedel also defended his PhD thesis [51] where he also uses this framework. Another example of existing secure fixed-point protocols comes from Ugwuoke et al., who in [62] proposed a method for dividing homomorphically encrypted fixed-point numbers.

A problem with the fixed-point approach is that if the order of magnitude of the data is not well known, there is the danger of either overflow or high imprecision. In essence, the problem is finding a suitable radix-point. Note that this can also leak information. Henecka et al. deal with this problem in [38]. They propose a method for mapping real numbers to integers in order to do secure computation. This is essentially a form of fixed-point numbers. They propose a method how two parties can agree on a suitable radix-point securely. However, it is a two-party protocol and thus somewhat limited in scope.

2.2. Secure Floating-Point Numbers

Floating-point numbers are more complicated than fixed-point numbers. However, they offer great precision and flexibility, and are the preferred method to represent real numbers in normal computation.

Kamm in her PhD thesis [46] used secure floating-point numbers. She constructed an R-like package for performing statistical analysis. She then used this package for the satellite collision problem.

In [48] Kerik, Laud, and Randmets presented optimizations for the Sharemind SMC engine that among other protocols improved floating-point protocols. We, in this thesis, base our secure floating-point numbers on this work.

Floating-point numbers have also been studied by other groups. In [6], Aliasgari et al. developed a framework for secure floating-point numbers that supported various operatinos, such as multiplication, addition, subtraction, comparison, and rounding. More complicated operations were also developed, such as the square root, exponentiation, and logarithm. In [3] Aliasgari and Blanton used the framework to develop privacy-preserving techniques for hidden Markov models and Gaussian mixture models. They also improved upon the efficiency of the exponential function. In [4], Aliasgari et al. give an approach for the malicious model. The results obtained were good, but this framework does not gain significant improvements in efficiency when the operation is carried out for larger vector sizes. That is, the amortized cost of operations per second does not improve significantly when we increase the number of operations carried out simultaneously.

In [35], Franz et al. implemented a secure floating-point number framework that implemented the IEEE 754 floating point standard. They used it for digitally filtering secure real-value signals. In [34], they applied these techniques to privacy-preserving sequence analysis. Franz also defended his PhD thesis [32] where he viewed secure real numbers, in particular, secure logarithmically encoded numbers and secure floating-point numbers.

Mohanty uses secure computation on floating-point numbers and fixed-point numbers for cloud-based image-processing in his PhD thesis [54]. He uses passively secure Shamir secret-sharing based approach. The operations used are those necessary for image processing. However, Shamir secret sharing and floating-point numbers appear to be difficult to combine as they are based on different arithmetic structures. Mohanty views two approaches of combining these two. The first approach is excluding the modular prime operation. This, however, introduces some security flaws, for example, it is unknown whether the scheme proposed is secure against side-channel attacks. Mohanty's second approach is converting the floating-point number to a fixed-point number. While this sounds on a superficial level similar to our Chapter 4, it is in fact not, as Mohanty does not use secret floating-point numbers. He merely uses secure fixed-point numbers to represent floating-point numbers. Thus the goals of Mohanty's thesis and this thesis are rather different in nature, as his goals are based on developing a framework that is specifically useful for secure image processing while we study the problem more generally.

In addition Bai et al. proposed secure floating-point numbers based on fully homomorphic encryption in [7] and Liu et al. proposed secure floating-point numbers dor secure data analysis in [52].

2.3. Other Approaches

As secure computation over reals is a relatively novel field, other real number types have also been used.

For example, Chung used continued fractions in her PhD thesis [25] in order to represent secure real numbers. Continued fractions are a method for representing real numbers that are not considered in this thesis.

In [33], Franz et al. used logarithmically scaled numbers to represent real numbers. We already noted that Franz used secure logarithmically encoded numbers in his thesis [32].

3. PRELIMINARIES

In this section we describe general notation, other necessary conventions, and preliminary results that are useful for better understanding this thesis.

A fundamental idea on which the field of SMC lays is the concept of *secret values*. We say that a value of a variable x is secret if it is somehow manipulated into some form in such a way that nothing about its value can be learned from this new form. We denote this new form with $[\![x]\!]$. There are various methods for making a value secret — for example encryption and secret sharing. There are also several possible definitions to what the phrase *'nothing about a value can be learned from* $[\![x]\!]$*'* means, we will later give several possible definitions to this that are more formal. We shall also use the words *secure value* and *private value* synonymously with the word *secret value*.

However, for the concept of secure computation it does not suffice for $[\![x]\!]$ to be private. We must be able to perform operations on secure values. Minimally, for the framework to be Turing-complete, we must be able to add and multiply secure values, i.e, we should have access to operations $\oplus$ and $\otimes$ such that $[\![x]\!] \oplus [\![y]\!] = [\![x+y]\!]$ and $[\![x]\!] \otimes [\![y]\!] = [\![x \cdot y]\!]$. Theoretically, it suffices to have constants, addition and multiplication for secure computation as a system with constants, addition and multiplication is Turing-complete. However, in practice, it is recommended to build more complicated operations (for example built from addition and multiplication) for efficiency and ease of writing protocols.

There are also various methods for building SMC settings where operations $\oplus$ and $\otimes$ with the abovementioned properties exist. We shall briefly explain fully homomorphic encryption, Yao schemes and secret-sharing based schemes,

3.1. Universal Composability

A basic foundation of cryptography are security proofs that show that a given protocol is secure, given certain assumptions. However, often these proofs are constructed as if the protocols were the end-products and not as if they were subproducts of some more complicated protocol. However, in practice, the protocols for which proofs are given are often just building blocks and thus the security of the more complex protocol should be proved by itself. This is very cumbersome and requires the prover to be well-versed in cryptography, which is undesired as it takes time and expert knowledge. It would be commendable if the security of a more complex protocol would follow from the security of its building blocks.

For this it would be necessary that a protocol would remain secure if placed in an *environment* where it interacts with other protocols, that are possibly controlled by malicious parties. More formally, a security of a protocol R is often defined by comparing the protocol to an ideal version F of the protocol. The ideal version is often modelled as a trusted third party. The security is shown by demonstrating that it is computationally difficult (i.e. success happens with a negligible probabil-

ity given a computationally bounded adversary) for the environment to distinguish the ideal case from the actual protocol. If the protocol leaked some data that the ideal case does not, then it would be possible to use that fact to distinguish between the ideal world case and real world case — which, provided that we have shown it to be practically infeasible, would be a contradiction.

We also need to add another layer between the environment and the ideal functionality. This is mainly for syntactic purposes. For example, in the multiparty case it is trivially possible to distinguish between the ideal functionality and the real-world multiparty case since the different parties pass messages to each other while the trusted third party does no such thing. Even if the messages are encrypted, they still leak something, for example, the length of the messages. Thus, we need to add a layer that makes the message-passing behaviour of the multiparty case and the ideal case indistinguishable. This layer is called a *simulator* S. It should be hard to distinguish between R and $S \circ F$. Here, by $S \circ F$ we mean that the distinguishing party only communicates with S. We have now in general terms introduced the concept of universal composability[19].

One main benefit of universally composable protocols is the UC theorem which states that the composition of UC protocols will be an UC protocol [26] Thus UC protocols make for good building blocks — given that we prove that a set of elementary protocols are universally composable, we can build much more complicated protocols out of them that will also be universally composable thanks to the theorem. This is the property the usefulness of which we noted at the beginning of this subsection. This fact allows this thesis to focus more on algorithm design and efficiency than proving the security of the schemes, as the security follows from the UC property of the building blocks.

3.2. Security Models

Security is an intuitive notion, but can be understood in several ways — in cryptography, there are several versions of security that are used. While more types exist, we shall focus on two types of security — passive and active security. We additionally talk about the assumption of how parties are allowed to collude.

First, there is the concept of passive security. In this security model, all the parties taking part of the protocol are presumed to behave as the protocol dictates, but may use the data to compute things that were not intended by the protocol, and yet should not learn anything more. This is also known as the honest-but-curious model.

We formalize this security concept in the following way in the UC framework. To do this, we introduce the concept of the adversary.

In the case of passive security, the adversary is allowed to passively corrupt some subsets of the parties. We define passive corruption via the ideal functionality F. The adversary can send F the message $(\text{passive}, i)$ which means that it corrupts the ith party. As a result F sends to the adversary the input messages

$F_{in,i}$ and the output messages $F_{out,i}$ of this party. The adversary is allowed to send these messages for fixed subsets of parties (for example, in the common t-threshold schemes, the adversary is allowed to corrupt subsets with no more than t parties).We now define the scheme to be passively secure if the environment cannot distinguish between R and $S \circ F$.

Second, there is the concept of the active security. In this security model, the integrity of the data is protected, even if malicious parties do not follow the protocol. Note that in a context where there is more than one party performing the secure computation, the protocol may be actively secure for some subset of parties but not for all subsets of parties. For example, [29] describes a model that is not secure when all parties are allowed to not follow the protocol, but is secure when there is at least one honest party that follows the protocol. This is generally formalized by describing an adversary who is allowed to control a certain subset of parties in the computation.

More formally, active security is formalized in the following way. The adversary is allowed to *actively* corrupt some parties. This means that the adversary can send F the message (active, i) which means that it corrupts the ith party. Now, F will ignore all incoming messages $F_{in,i}$. However, the adversary can send to F any messages x_i and F will accept these as legitimate input. F also sends to the adversary $F_{out,i}$. Similarly as before we now define the scheme to be actively secure if the environment cannot distinguish between R and $S \circ F$. This definition should capture the intuition that the parties are allowed to choose their input and thus it is not possible to protect against an attack that merely changes the input. However, the only thing an attacker should be able to do is to change the input and to learn the result.

There are more security models besides active and passive security, for example, the concepts of covert security and synchronous and asynchronous communication, and others. However, as this thesis focuses mainly on the algorithm design aspect of SMC, these finer notions of security are not crucial for understanding this thesis and thus we will omit them.

3.3. Methods for Secure Computation

In this section, we will describe three popular approaches to secure computation. These approaches are based on secret-sharing, on Yao's garbled circuits, and on fully homomorphic encryption.

3.3.1. Secret Sharing

Secret sharing was first proposed by Shamir [61] and Blakley [10] in 1979.

Secret sharing is a method for sharing a piece of data between k agents in such a way that some subsets of agents can reconstruct the data by collaborating and other subsets learn no new information from their shares when collaborating.

We will now explain additive and Shamir secret sharing after which we will discuss SMC based on secret sharing.

Shamir Secret Sharing. An important group of secret-sharing schemes are the so-called *threshold secret sharing schemes*. A (t,k)-threshold secret sharing scheme is a scheme where any subset of t parties can reconstruct the secret value but no smaller group can gain any information about it. One example of this is Shamir secret sharing [61]. The scheme works in the following way. Our underlying algebraic structure for this is a quotient field $\mathbb{Z}_p$. Suppose that we want to secret-share a value a. For this we choose t random values $b_1,\ldots,b_t$ from $\mathbb{Z}_p$ and construct the polynomial $f(x) = a + \sum_{i=1}^{t} b_i x^i$. We evaluate the polynomial f at k non-zero values $c_1,\ldots,c_k$ and give every party M_i the pair $(c_i, f(c_i))$. Now any subset of t players or more can find the coefficients of the polynomial using Lagrange interpolations, but any smaller subset possesses no information, even when they are computationally unbounded.

Additive Secret Sharing. We will now describe additive secret sharing.

In additive secret sharing, there are M independent parties. If we want to secretly store a value x that is an element of an abelian ring A, then we randomly pick elements $x_1,\ldots,x_{M-1}$ from A. We set $x_M := x - x_1 - x_2 - \ldots x_{M-1}$ in $\mathbb{Z}_N$. Every party P_i gets the value x_i. Now, in A, the sum of all the x_i is x, however, none of the parties learns no new information about the value of x from its share x_i — the values $x_1,\ldots,x_{M-1}$ were randomly chosen, and x_M is uniformly random.

Moreover we have the following theorem from [11]

Theorem 1. *For each secret value $s \in \mathbb{Z}_{2^{32}}$, any subset of $n-1$ shares of s is uniformly distributed and for any two secret values $u, v \in \mathbb{Z}_{2^{32}}$, their secret shared forms are indistinguishable for any coalition of parties holding up to $n-1$ shares.*

The result and the proof can be easily generalized from $\mathbb{Z}_{2^{32}}$ to any Abelian ring A. This guarantees the passive security of secret-sharing.

In [29], a version of additive secret sharing that is actively secure when all except one of the parties is corrupted was described.

Secure Multiparty Computation Based on Secret Sharing. Some secret sharing schemes also allow for methods to compute functions without the agents not learning anything about the data. This allows us to construct secure multiparty computation based on secret sharing.

For Shamir secret sharing, it is possible to manipulate the shares in a way to obtain addition and multiplication operations [8]. Examples of Shamir secret-sharing based SMC schemes are VIFF [28], SCET [16], and SEPIA [18].

This is also possible for additive secret sharing. Examples of secure computation schemes based on additive secret sharing include Sharemind [11] and SPDZ [29]. We used the Sharemind scheme in our implementations, with $M = 3$.

When designing protocols for SMC based on secret sharing, one should keep in mind the following heuristic principle. Namely in this setting, for computation

the agents generally need to exchange messages. As sending one large message is generally much faster than sending many small messages sequentially, parallel composition is often orders of magnitude more efficient than sequential composition [11].

3.3.2. Yao's Garbled Circuits

In [64], Yao described a method for securely evaluating binary circuits. Here the function that we wish to securely evaluate must be expressed as a binary circuit. This method provides secure evaluation of NOT-gates, AND-gates, and XOR-gates. It also allows for secure evaluation of compositions of these gates and is thus Turing complete. We will now shortly describe Yao's scheme in this subsection.

In this two-party scheme one party builds the binary circuit along with their inputs and sends the circuit to the other party who then evaluates it. For clarity, let the party that builds the circuit be called Alice and the party that evaluates the circuit be Bob. We assume that both Alice and Bob put inputs into the circuit. Note that if some internal value of the circuit depends only on the inputs of one party, then that party can compute it locally and thus more efficiently. Thus we assume without loss on generality that every gate evaluated depends on both the inputs of Alice and the inputs of Bob.

Alice takes the binary circuit C that Alice and Bob have agreed to evaluate together and compiles it into a garbled binary circuit C' that she sends to Bob to evaluate. How this garbled circuit is built and how Bob evaluates it are the key components of this scheme. Currently we shall describe a circuit that can only be evaluated once.

A circuit consists of wires and gates. For every wire w, Alice prepares two values w_0 and w_1 that respectively correspond to the values 0 and 1. Generally, though evaluation, Bob can obtain one of those values w_b which signifies that the bit carried by that wire in this evaluation is b. However, while Bob gets access to w_b, he does not learn whether it is w_0 or w_1. We now briefly explain how this is achieved.

The easiest case is when w is an input wire of Alice — if the corresponding value is b then Alice can simply send the corresponding value w_b to Bob.

The case when the value is an input wire of Bob is not much more complicated. In this case, Bob uses oblivious transfer on the pair (w_0, w_1) to obtain w_b where b is the corresponding input bit of Bob. The definition of oblivious transfer guarantees that Bob only learns one of the values and that Alice does not learn the value b.

Let us now consider the case where the wire w is the output wire of a gate. For clarity, let it be an AND-gate. The cases for the other binary gates are analogous.

Let two wires u and v be the input wires to the gate and let w be the output wire of the gate. Let the input bits be a and b, i.e, Bob has the values u_a and v_b. We want

that Bob would obtain w_{aANDb} as the result. This is achieved via a garbled gate. A garbled gate consists of four randomly permuted doubly-encrypted ciphertexts. The encryption used is a symmetric encryption. There are four possible pairs of values (u_a, v_b) that Bob can possess — he should be able to decrypt precisely one of the cyphertexts provided to him. The value that Bob can decrypt (e.g the value that is encrypted with the keys u_a and v_b) is w_{aANDb}

Thus, there are four ciphertexts $c_{a,b} = Enc_{u_a}(Enc_{v_b}((w_{aANDb})))$, for example, $c_{0,1} = Enc_{u_0}(Enc_{v_1}((w_0))$. Bob tries to decipher those four ciphertexts with u_a and v_b. He succeeds in decrypting one of them, namely w_{aANDb}. Since the outputs are permuted, Bob does not know which of the four ciphertexts he managed to decrypt. Thus he learns nothing new from those four ciphertexts (besides w_{aANDb}, of course).

Now consider the case when the wires hold the output bits. Essentially, there are two approaches to this — either first Bob learns the values of the output bits and sends them to Alice or vice versa. In the first case, the values of the output wires can actually correspond to the bits it carries – so the value of w_0 will be 0 and the value of w_1 will be 1. In the second case, w_0 and w_1 are encryptions of 0 and 1, respectively, and only Alice knows the decryption key.

Other gates can be constructed in a similar manner. Fairplay [53] is an example of a SMC system using Yao's circuits. There have been improvements on making Yao's circuits more effective and more secure, for example [55] and [45].

3.3.3. Fully Homomorphic Encryption

The problem of fully homomorphic encryption was originally proposed in 1978 [60], however, the first possible solution was not proposed until 2009 [36]. Fully homomorphic encryption is a method for secure computation with encryption , decryption, addition of plaintexts, addition of cyphertexts, multiplication of plaintexts and multiplication of cyphertexts that we respectively denote with $Enc, Dec, +, \cdot, \oplus$, and $\otimes$ that satisfy the properties $Enc(Dec(x)) = x$, $Enc(x+y) = Enc(x) \oplus Enc(y)$, and $Enc(x \cdot y) = Enc(x) \otimes Enc(y)$. As a system with addition and multiplication is Turing-complete, any computable function can theoretically be evaluated using fully homomorphic encryption.

Thus suppose that there is a set of parties who possess the values $x_0, \ldots, x_k$ and who wish to evaluate the computable function $f(x_0, \ldots, x_k)$. To do that, they could homomorphically encrypt the values $[\![x_i]\!]$ on which the computation will be run. They then send those values to the computing party P who then homomorphically computes $f([\![x_0]\!], \ldots, [\![x_k]\!])$ and obtains the result $[\![f(x_0, \ldots, x_k)]\!]$. Party P then sends $[\![f(x_0, \ldots, x_k)]\!]$ back to the output parties who can decrypt the answer. As P only sees encrypted values, it learns nothing about the input values $x_0, \ldots, x_k$ and because the system is Turing-complete, given enough time, P is able to compute $f([\![x_0]\!], \ldots, [\![x_k]\!])$.

While this scheme is very desirable, in practice, the existing schemes are currently too slow for real-world applications [2].

3.3.4. Secure Computation Based on Partially Homomorphic Encryption

While fully homomorphic encryption is yet not efficient enough for practical use, there are applications where full homomorphism is not needed. For some applications, it is sufficient that an encryption scheme is homomorphic in one operation. An encryption scheme that is homomorphic with regards to addition is one where there exists an operation $\oplus$ in the group of cyphertexts such that $Enc(x+y) = Enc(x) \oplus Enc(y)$. Homomorphism with regards to multiplication is analogously defined. ElGamal and Paillier encryption schemes are examples of somewhat homomorphic encryption schemes.

Somewhat homomorphic encryption schemes can have practical applications. For example, we can consider a binary vote where each participant votes either 'yes' or 'no', where 'yes' is encoded by 1 and 'no' by 0. Participants encrypt their votes and the cyphertexts are added together. Provided that the participants followed the protocol, the result will be an encryption of some value k, where k is the number of participants who voted 'yes'. We can decrypt this value and trivially obtain the result of the vote. Note that this does not leak who voted 'yes' and who voted 'no'.

Applications such as this are very useful and practical, however, their scope is limited, because we can use only one operation.

3.3.5. On Using Secure Computation Frameworks

Note that while having a number of universally composable primitives can make designing secure algorithms quite similar to regular programming, there are still some aspects that are different and need to be considered. First, one needs to consider that secure computation primitives are generally much slower than the respective primitives in the standard setting as generally either network communication or cryptographic operations are needed for secure computation, and these take a non-trivial amount of time. Second, the relative costs of different primitives can be different from their usual cost in non-SMC frameworks. These relative costs depend on the framework we use. For example, in the Sharemind framework [44] where we developed our implementations, bit extraction is an operation that is several times more expensive than integer multiplication, as we will see in the table 1 presented at the end of this chapter. Third, use of branching is heavily limited. Operations such as the following where $[\![b]\!]$ is a secure bit and A and B

are sub-programs, are not allowed.

```
1 if b == 1 then
2     [[x]] ← A
3 else
4     [[x]] ← B
```

If this was allowed the program flow would leak information about the value of $[\![b]\!]$. This, of course, is the case not only for secure bits, but for all secure values. Instead we can execute both branches and choose the result obliviously, such as in the following example.

```
1 [[x_A]] ← A
2 [[x_B]] ← B
3 [[x]] ← [[b]] · [[x_A]] + (1 − [[b]]) · [[x_B]]
```

Presuming that the operations used are valid, x will be set to the correct value. However, this can be costly in resources, and thus algorithms that rely heavily on branching become prohibitively expensive and are hence not practically usable in SMC.

3.3.6. Security Guarantees of Our Implementation

The exact security guarantees of the methods developed in the current thesis depend on the underlying framework. As stated above, we have implemented our protocols on Sharemind SMC engine which, in its default three-party setting, provides the following guarantees.

We start with the following definition (see, e.g. [14]).

Definition 1. We say that a share computing protocol is *perfectly simulatable* if there exists an efficient universal non-rewinding simulator $\mathscr{S}$ that can simulate all protocol messages to any real world adversary $\mathscr{A}$ so that for all input shares the output distributions of $\mathscr{A}$ and $\mathscr{S}(\mathscr{A})$ coincide.

In the original Sharemind paper [12], Bogdanov *et al.* show that a perfectly simulatable share computing protocol that ends with re-sharing of output shares is perfectly universally composable. In [14], Bogdanov *et al.* show perfect simulatability of a number of basic Sharemind primitives which we make use of. On the other hand, the author has not performed a review of Sharemind code base to make sure that all the primitive implementations really perform the re-sharing required for the composability proofs to work.

Another potential source of privacy violation is premature declassification of values; say, control flow bits (which is sometimes done to trade some privacy for efficiency). We have designed all our protocols so that no declassifications happen before the end result is computed.

Still, one would prefer to verify the security guarantees of high-level complex protocols formally. During the time research of this thesis was ongoing, such a

tool became available in the form of a compiler for a Domain Specific Language developed particularly for Sharemind protocols [59]. The author was able to make use of this tool for Chapter 6.

The above results hold for the standard passive honest-but-curious adversary model where the attacker is only able to observe values held by one of the computing parties. However, it has been recently shown by Laud and Pettai that privacy is also guaranteed in the active setting [56]. This means that Sharemind protocols tolerate one computing party that can arbitrarily deviate from the protocol, but is still unable to learn anything about the private inputs.

3.4. Notations and Conventions

In this section we will give certain notations and conventions that will be used throughout this thesis. We shall assume the following conventions.

- We use $\gg$ and $\ll$ to denote shifts to the right and left, respectively. That is, $x \gg k$ means the value x shifted to the right by k bits and $x \ll k$ means the value x shifted to the left by k bits. We also use $x << y$ (and $x >> y$) to denote that the value x is significantly smaller (larger) than y.
- If we say that $\{a_i\}_{i=0}^{n-1}$ are the bits of an n-bit unsigned integer a, then the bit a_{n-1} will refer to the most significant bit, the bit a_{n-2} to the next most significant and so on, with a_0 being the least significant bit.
- An integer type may be either unsigned, which means it takes only non-negative values, or signed, which means it can be either negative or positive. Usually we use some finite algebraic structure to store integers, such as a residue ring $\mathbb{Z}_k$. Note that the values of this finite ring can be interpreted as many elements of $\mathbb{Z}$. While usually there is a canonical interpretation, in some cases, there is potential for confusion, e.g in Subsection 6.1.3. To avoid confusion in those cases, we will define two functions. We shall use the function $z : \mathbb{Z}_{2^n} \to \mathbb{Z}$ that maps an element $c \in \mathbb{Z}_{2^n}$ to the member of the residue class of c that is in the interval $[0, 2^n - 1]$. Similarly, we define the function $t : \mathbb{Z}_{2^n} \to \mathbb{Z}$ that maps an element $c \in \mathbb{Z}_{2^n}$ to the member of the residue class of c that is in the interval $[-2^{n-1}, 2^{n-1} - 1]$. We will not use these functions before Subsection 6.1.3.

  For the sake of simplicity, if we are speaking about unsigned integers when the underlying ring is some $\mathbb{Z}_{2^n}$, we will use x instead of $z(x)$. Thus, we will use x to refer to both as a member of $\mathbb{Z}_{2^n}$ and $\mathbb{Z}$ without there being confusion about which integer we are speaking. Similarly, if we are speaking about signed integers when the underlying ring is some $\mathbb{Z}_{2^n}$, we will use x instead of $t(x)$. Here we can also use x to refer to both a member of $\mathbb{Z}_{2^n}$ and $\mathbb{Z}$.
- When not specified differently, integers refer to unsigned integers.
- When we speak of signed integers in this thesis, we mean the standard two's

complement. If we have a value a with bits $a_{n-1}, a_{n-2}, \ldots a_0$, then this represents the number $-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$.

- We use $\mathscr{F}_k$ to denote the kth Fibonacci number. Recall that $\mathscr{F}_0$ and $\mathscr{F}_1$ are defined to be equal to 1 and that for $k \geq 2$, Fibonacci numbers are defined recursively as $\mathscr{F}_k = \mathscr{F}_{k-1} + \mathscr{F}_{k-2}$.

We also assume that we have access to some pre-existing primitives and secure data types as they have been realized in various frameworks such as the frameworks noted previously.

3.4.1. Secure Bits and Integers

We presume that we have access to secure integers. *'Security'* can have several meanings, depending on the security model we use (as described in Section 3.2) and the method we use to achieve that security goal (as described in Section 3.3).

For all security models and methods, $[\![x]\!]$ denotes the structure that is stored and contains the integer value x in a secure fashion. For example, this can be either an encryption of x or a secret-sharing of x

3.4.2. Secure Fixed-Point Numbers

A fixed-point number is a data type that represents real numbers with a specific absolute precision. A fixed-point number has a specified number of bits after the radix-point and a specified number of bits before the radix-point.

More formally, we define a fixed-point number to be a triple (a,m,x) where $a \cdot 2^{-m} = x$ holds. Here a will be the integer value that is stored in the computer memory, x is the real number value that the number represents, and m is the positive 'radix-point' that describes how many bits of a should be thought to be 'after the radix point'.

For the sake of brevity, we denote such a fixed-point number with ${}^f x$. To refer to the integer a, we use $\widetilde{x}$. We shall also refer to $\widetilde{x}$ as the representative of the fixed-point number.

The precedence of the operator ^{f} will be weaker than exponentiation but stronger than multiplication and division. Thus ${}^f x^2$ will refer to the fixed-point number holding the value x^2, not the square of the fixed-point number ${}^f x$. Concerning multiplication and division, we will generally use parenthesis in order to avoid confusion. We will later introduce similar notations for floating-point numbers and a new number type we will call golden section numbers. Their precedence is analogous.

Note that generally, x must be a member of the set

$$\{a \cdot 2^{-m} | a \in \{0,1,\ldots,2^n - 1\}\}.$$

We shall, in some cases, write ${}^f y$ for some y that does not belong to that set (for example, ${}^f \pi$). In that case, ${}^f y$ signifies the triple (a,m,x) where $a \cdot 2^{-m} = x$ and x

is the member of $\{a \cdot 2^{-m} | a \in \{0, 1, \ldots, 2^n - 1\}\}$ that is closest to y. In the case of there being two such members of $\{a \cdot 2^{-m} | a \in \{0, 1, \ldots, 2^n - 1\}\}$ that are closest to y, we shall use the smaller value.

We also similarly define a secure fixed-point number as a triple $([\![a]\!], m, x)$ where $[\![a]\!]$ is the secure structure that is stored and contains a, and where likewise $a \cdot 2^{-m} = x$. We will call $[\![a]\!]$ the representative of the secure fixed-point number, and a the value of the representative. We will refer to this structure as $[\![^f x]\!]$.

Throughout this thesis we will use the letter m to signify the number of bits after the radix-point when speaking of fixed-point numbers. We will also refer to this value as the radix-point.

There are several possible variations of fixed-point numbers. The most important variable that we will need to consider is whether the fixed-point number is signed or not. Also, if the number is signed, we need to consider how this is achieved.

First, consider the case when we use unsigned fixed-point numbers. In this case we assume that all the numbers that we represent are non-negative. In that case, the underlying integers (i.e the elements a in the tuples $([\![a]\!], m, x)$) are also unsigned, and also represent only non-negative numbers. This is useful in cases where we know that the numbers that we represent will be non-negative.

Second, consider the case where we use signed fixed-point numbers that are achieved by using signed integers as the underlying data type. Here, the element a in the tuple (a, m, x) is a signed integer. We refer to these kinds of signed fixed-point numbers as three-field signed fixed-point numbers or three-field fixed-point numbers. By default, when we refer to signed fixed-point numbers, we assume them to be this type of signed fixed-point numbers.

Third, consider the case where we, instead of the triple (a, m, x) have a quadruple $([\![a]\!], [\![s]\!], m, x)$ where s is a sign bit and a is an unsigned integer, with $x = (-1)^s \cdot a \cdot 2^{-m}$. We will refer to these fixed-point numbers as four-field signed fixed-point numbers or four-field fixed-point numbers.

Here the sign bit represents whether the number is positive or negative and $a \cdot 2^{-m}$ is the absolute value of the number. In the secure version of this data type, if we wish to add or subtract, we need to perform both the addition and subtraction operations on the representatives and then

Also comparison of absolute values is needed. If we are given a signed fixed-point number of this type that represents x we will denote its sign with sgn_x and the unsigned integer with abs_x.

Note that this makes addition and subtraction more costly that three-field fixed-point numbers. However, multiplication can be be more efficient compared to the three-field signed fixed-point numbers. However, as addition is a more common operation than multiplication, we will generally prefer to use three-field fixed point numbers. This is the reason why we assume the default signed fixed-point numbers to be three-field signed fixed-point numbers. However, in Chapter 6, we

will also occasionally use four-field signed fixed-point numbers.

When we are talking about real numbers in a situation where they are represented only by fixed-point numbers, we will occasionally use the notation in the following way. We shall interpret the interval $[a,b)$ to be the same interval as $[a,b']$ where b' is the largest expressible fixed-point number smaller than b. Similarly, $(a',b]$ will be considered to be the same as the interval $[a',b]$ where a' is the smallest expressible fixed-point number larger than a. The reason for this is that if we want to express numbers in $[b',b)$, we have to either express them with b' or with a fixed-point number outside of $[a,b)$.

Secure Floating-Point Numbers. We will use floating-point numbers in this thesis as described in [46]. The IEEE 754 standard defines the single-precision floating-point number in the following way [1]: it is a 32 bit number, where the first bit refers to the sign s, the next 8 bits refer to the exponent E and the last 23 bits refer to the significand σ.

The format also has a fixed number q called bias and exception flags for the conditions such as division by zero or overflow.

The value of a number expressed by these values is

$$(-1)^s \cdot 2^{E-q} \cdot \sigma.$$

The floating-point standard analogously defines other precisions [1]. However, this format is generally not very suitable for the secure computation setting. Usually, accessing individual bits of a machine word is very cheap, however, in secure computing, it can be rather expensive. We will see examples of this at the end of this chapter. Thus, instead of packing all the three values into one machine word, we use three separate integer values. We use the triple of integers (s_x, E_x, σ_x) to represent the floating-point number with the value x.

Analogously to the case of the fixed-point number we say that the floating-point number ${}^F x$ is a 5-tuple $(s_x, E_x, \sigma_x, q, x)$ with the property that

$$x = (-1)^{1-s_x} \cdot 2^{E_x-q} \cdot \frac{\sigma_x}{2^n} \text{ and } \sigma_x \in [2^{n-1}, 2^n - 1], \tag{3.1}$$

where n is the number of bits in σ_x. We refer to the three values s_x, E_x, σ_x, as the sign, the exponent, and the significand, respectively. When we wish to refer to all three, we call them the representatives of ${}^F x$. We also will use e_x to denote $E_x - q$ for more intuitive understanding. Note that

$$|x| \in [2^{e_x-1}, 2^{e_x}) \tag{3.2}$$

holds for all $x \neq 0$. Likewise we will use τ_x to denote $\frac{\sigma_x}{2^n}$. The requirement that $\sigma_x \in [2^{n-1}, 2^n - 1]$ can also be thought of as $\tau_x \in \left[\frac{1}{2}, 1\right)$.

Like in the case of fixed-point numbers, most of the reals x cannot be represented as in Equation (3.1). Thus, if we use ${}^F y$ where y cannot be represented

in that way, it will be equal to ${}^F x$ where x is the closest real number that can be represented in such a way. If there are two closest numbers to y then we will use the one with the smaller absolute value. There is the special case of $x = 0$, where the 5-tuple $(s_0, 0, 0, q, 0)$ is also considered a legitimate floating-point number representing 0. The sign s_0 is either 1 or 0, representing whether it is a positive or a negative zero. Note that in protocols, we often use certain assumptions about floating-point numbers that do not necessary hold for zero. Thus one should always check whether the protocol works as intended in the case of zero.

Likewise, $[\![{}^F x]\!]$ shall denote a secret floating-point number, that is, a 5-tuple $([\![s_x]\!], [\![E_x]\!], [\![\sigma_x]\!], q, x)$ with $x = (-1)^{1-s_x} \cdot 2^{E_x - q} \cdot \frac{\sigma_x}{2^n}$ and $\sigma_x \in [2^{n-1}, 2^n - 1]$, and $[\![s_x]\!], [\![E_x]\!], [\![\sigma_x]\!]$ being securely stored.

Note that our floating-point numbers do not have any exception handling, because public exception handling would leak information about values. Alternatively, at every step we would have to obliviously check for overflow which would be very expensive. In this system, programs should be designed in such a way that illegal operations were never performed. If they are performed, there are no guarantees that the results would mean anything.

Note that we require that for all nonzero values of x, the first bit of σ_x is 1, thus, $\tau_x = \frac{\sigma_x}{2^n} \in [0.5, 1)$. This is due to the common practice of representing significands [1]. This helps in some of the protocols. Note that we can think of the significand as a fixed-point number with the radix-point n and the value in $[0.5, 1)$.

3.4.3. On Algorithm Notation

We now give a short overview of how we present our protocols. We use $[\![x]\!] \leftarrow y$ to denote that $[\![x]\!]$ stores the secure value that is equal to the public value y, i.e, here, $x = y$.

For fixed-point numbers, we use ${}^f x \xleftarrow{f} a$ to denote that the representative of ${}^f x$ will be a, i.e $\tilde{x} = a$. Likewise for secret values, $[\![{}^f x]\!] \xleftarrow{f} [\![a]\!]$ will denote that the protected representative of $[\![{}^f x]\!]$ will be $[\![a]\!]$. In the case that we use four-field fixed-point numbers we use the notation ${}^f x \xleftarrow{f} (s, a)$ to denote that s is the sign of the fixed-point number and a is the representative.

Similarly, for floating-point numbers, we use ${}^F x \xleftarrow{F} (s, E, \sigma)$ to denote that s is be the sign, E the exponent and σ the significand of ${}^F x$. Likewise for secret values, $[\![{}^F x]\!] \xleftarrow{F} ([\![s]\!], [\![E]\!], [\![\sigma]\!])$ denotes that the protected representatives of $[\![{}^F x]\!]$ are $[\![s]\!], [\![E]\!]$ and $[\![\sigma]\!]$.

As noted in section 3.3.5, secure computing tends to be remarkably slower than analogous computation in nonsecure setting and, thus, optimization is important. One way of optimizing is using parallel composition where possible. Besides the usual advantage gained from parallel composition, some SMC frameworks [11, p.74-75] benefit greatly from parallel composition.

Thus we need a way to denote in protocols and in their descriptions that some

operations are done in parallel composition. It is often natural to represent parallel composition with vector operations. Generally, if we have a protocol $F(a,b,\ldots x)$ that outputs y, then to denote that we want to compute $y_0 = F(a_0,b_0,\ldots,x_0)$, $y_1 = F(a_1,b_1,\ldots,x_1),\ldots,y_s = F(a_s,b_s,\ldots,x_s)$ in parallel composition we write

$$\{y_i\}_{i=0}^{s} \leftarrow F(\{a_i\}_{i=0}^{s},\{b_i\}_{i=0}^{s},\ldots,\{x_i\}_{i=0}^{s}).$$

In the case where some inputs c are common for all the parallel evaluation, we shall write c instead of $\{c_i\}_{i=0}^{s}$. Thus, for example, if we wish to evaluate $y_0 = F(a_0,b,c), y_1 = F(a_1,b,c),\ldots,y_s = F(a_s,b,c)$, we denote it with

$$\{y_i\}_{i=0}^{s} \leftarrow F(\{a_i\}_{i=0}^{s},b,c).$$

It might also be that the elements that we wish to call as the i-th inputs of a function in a parallel evaluation have different notations, such as a and b instead of a_0 and a_1. In that case we will simply denote it with the keyword *in parallel*, for example:

```
1 begin in parallel
2 |   [[c]] ← F([[a]])
3 |   [[d]] ← F([[b]])
```

We also use the notation $f(X)$ where f is some function and X is a set. This will mean the set $\{f(x)|x \in X\}$.

3.4.4. Existing Primitives

The framework-based approach depends on building more complex primitives out of existing ones. Thus, here we list the building blocks that will be used in the protocols that we present. Note that not all primitives are needed for all chapters. We will specify at the beginning of each chapter what primitives will be needed for that chapter. As the protocols that we present were implemented, the existing primitives described in this section had also been previously implemented. For some integer-related primitives, there are versions for both signed and unsigned integers. We will use the same notation for both of the cases. By default, unsigned integers are used. When signed integers are to be used, then it will be said so in context. Unsigned and signed integers will not be used together in the same context.

General Secure Primitives. This subsection gives an overview of the general previously-existing secure primitives.

- We have addition and subtraction of secret integers. We denote this with $[\![a]\!] + [\![b]\!]$ or $[\![a]\!] - [\![b]\!]$, respectively. The result of this operation is $[\![a+b]\!]$ or $[\![a-b]\!]$, respectively. We also have these operation for signed integers.
- We have multiplication of public and private integers. We denote this with $a[\![b]\!]$. The result of this operation is $[\![ab]\!]$. We also have this operation for signed integers.

- We have multiplication of private integers. We denote this with $[\![a]\!] \cdot [\![b]\!]$. The result of this operation is $[\![ab]\!]$. We also have this operation for signed integers.
- We have an operation that returns the last bit of a private integer $[\![a]\!]$. We denote this with $\mathsf{LastBit}([\![a]\!])$.
- We have an XOR-operation that, given two secret values $[\![a]\!], [\![b]\!]$, where $a, b \in \{0,1\}$, returns $[\![a \text{ XOR } b]\!]$. Note that if a and b are stored as bits, this is essentially addition of a and b but if a and b are stored in some larger integer format, the value must be computed as $[\![a]\!] + [\![b]\!] - 2[\![a]\!] \cdot [\![b]\!]$.
- We have an oblivious choice operation that takes a secure bit b and two values $[\![x]\!]$ and $[\![y]\!]$ as inputs. If $b = 1$, the output will be set to $[\![x]\!]$, and if $b = 0$, it will be set to $[\![y]\!]$. We refer to this operation as

  $$\mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![x]\!], [\![y]\!]).$$

  Note that this operation can easily be generalized to tuples. We take in a secure bit $[\![b]\!]$ and two secret tuples $([\![x_1]\!], [\![x_2]\!], \ldots, [\![x_k]\!])$ and $([\![y_1]\!], [\![y_2]\!], \ldots, [\![y_k]\!])$. We will output $([\![x_1]\!], [\![x_2]\!], \ldots, [\![x_k]\!])$, if $b = 1$, and $([\![y_1]\!], [\![y_2]\!], \ldots, [\![y_k]\!])$, if $b = 0$. This assumes that for all i, $[\![x_i]\!]$ and $[\![y_i]\!]$ have the same type. This operation may also be used to obliviously choose between more complicated structures, such as secure fixed-point numbers. We also have this operation for signed integers.
- We have an operation called $\mathsf{MSNZB}([\![x]\!])$ that returns a binary vector $\{[\![b_i]\!]\}_{i=0}^{n-1}$ that is the secret characteristic vector for the most significant nonzero bit of x. That is, $b_i = 1$ if and only if the ith bit is the most significant nonzero bit of x, otherwise $b_i = 0$.
- We we have the bit-extract operation that takes in a secret integer $[\![x]\!]$ and outputs the vector of n secret values $\{[\![u_i]\!]\}_{i=0}^{n-1}$ where each $u_i \in \{0,1\}$ and $u_{n-1}u_{n-2}\ldots u_0$ is the bitwise representation of $[\![x]\!]$. We refer to this as $\mathsf{BitExtraction}([\![x]\!])$. We also have this operation for signed integers.
- We have a comparison operation that takes in two secret integers $[\![x]\!]$ and $[\![y]\!]$ and outputs a secret bit b that is equal to 1 if $x \leq y$ and 0 if $x > y$. We refer to this operation as $\mathsf{LTEProtocol}([\![x]\!], [\![y]\!])$. We also have this operation for signed integers.
- We have a bit-shift operation that takes a secret shared integer $[\![x]\!]$ and a public integer k and outputs $[\![x \gg k]\!]$ where $x \gg k$ is equal to x shifted right by k bits. $x \gg k$ is equal to $\frac{x}{2^k}$ rounded down. We refer to this operation as $\mathsf{PublicBitShiftRightProtocol}([\![x]\!], k)$.
- We have a primitive for dividing a signed integer $[\![x]\!]$ by 2^k, rounded to the value with a lower absolute value. This can be achieved with performing in parallel the operations $\mathsf{PublicBitShiftRightProtocol}([\![x]\!], k)$ and $\mathsf{PublicBitShiftRightProtocol}([\![-x]\!], k)$ and obliviously choosing the correct

value based on the sign of x. We refer to this operation as $\mathsf{DivideBy2tok}(x,k)$. Note that we will use this operation also polymorphously for unsigned integers.

- We have an operation for converting a secure integer to a secure boolean. It takes in a secret-shared integer $[\![x]\!]$ where x is equal to either 0 or 1, and converts it to the corresponding boolean value shared over $\mathbb{Z}_2$. We refer to this operation as $\mathsf{ConvertToBoolean}([\![x]\!])$. We also have this operation for signed integers.
- We have a primitive for converting a secure boolean to a secure integer. It takes in a bit $[\![b]\!]$ secret-shared over $\mathbb{Z}_2$ and outputs a value $[\![x]\!]$ secret-shared over $\mathbb{Z}_n$, where x is equal to b as an integer. We refer to this operation as $\mathsf{ConvertBoolToInt}([\![b]\!])$.
- We have a primitive for converting a private integer $[\![x]\!]$ from a smaller ring $\mathbb{Z}_k$ to a larger ring $\mathbb{Z}_{k'}$. We will denote this with $\mathsf{ConvertUp}([\![x]\!],k')$. We also have this operation for signed integers.
- Similarly, we have a primitive for converting a private integer x from a larger ring $\mathbb{Z}_k$ to a smaller ring $\mathbb{Z}_{k'}$. We will denote this with $\mathsf{ConvertDown}([\![x]\!],k')$. We also have this operation for signed integers.
- We have a primitive that performs a more general version of oblivious choice. It takes in an array of secret integers $[\![x_0]\!],\ldots,[\![x_{k-1}]\!]$ and a secret index $[\![\ell]\!]$ where $\ell \in [0,k-1]$, and outputs the shared integer $[\![x_\ell]\!]$. We denote this primitive with $\mathsf{GenObliviousChoice}([\![x_0]\!],\ldots,[\![x_{k-1}]\!],[\![\ell]\!])$.
  We also have this operation for signed integers.
- We have a primitive that shifts a secret integer to the right by a secret number of bits. It takes a secret value $[\![x]\!]$ and a secret integer $[\![k]\!]$ and outputs $[\![x \gg k]\!]$ where $x \gg k$ is equal to x shifted right by k bits. When we apply this protocol to an n-bit secret integer $[\![x]\!]$ and k is not among $0,\ldots,n-1$, the result will be $[\![0]\!]$. We refer to this operation by $\mathsf{PrivateBitShiftRightProtocol}([\![x]\!],[\![k]\!])$.
- We have a primitive for adding two secure floating-point numbers. It takes in two secure floating-point numbers $[\![^F x]\!]$ and $[\![^F y]\!]$ and outputs the secure floating-point number $[\![^F x+y]\!]$. We denote this operation with $[\![^F x]\!]+[\![^F y]\!]$.
- We have a primitive for multiplying a secure and a public floating-point number [46]. It takes in a private float $[\![^F x]\!]$ and a public float ${}^F y$ and outputs a private float $[\![^F xy]\!]$. We denote this operation with $[\![^F x]\!]\cdot {}^F y$.
- We have a primitive for multiplying two secure floating-point numbers [46]. It takes in two secure floating-point numbers $[\![^F x]\!]$ and $[\![^F y]\!]$ and outputs the secure floating-point number $[\![^F xy]\!]$. We denote this with $[\![^F x]\!]\cdot[\![^F y]\!]$.
- We have a primitive for converting a secure integer into a secure floating point number. It takes in a secure integer $[\![x]\!]$ and outputs the secure floating-point number $[\![^F x]\!]$ [47].
- We have a secret-sharing specific primitive $\mathsf{ReshareToTwo}([\![x]\!])$. This prim-

itive takes a secret-shared integer $[\![x]\!]$ and reshares it in such a fashion that all but the first two parties P_0 and P_1 hold zeroes.

Primitives on Public Values. In this section we will describe some operations that function on public values that we will need. They are rather simple in nature, and their cost compared to private operations is negligible, but for cleaner notation in protocols, they are necessary.

- We have a primitive for converting a public integer a that is in the ring $\mathbb{Z}_n$ to a larger ring $\mathbb{Z}_{n'}$ where $n' > n$. We will denote this with $\mathsf{ConvertUp}(a, n')$. This naturally exists for both signed and unsigned integers.
- Similarly, we have a primitive for converting a public integer a that is in the ring $\mathbb{Z}_n$ to a smaller ring $\mathbb{Z}_{n'}$ where $n' < n$. We will denote this with $\mathsf{ConvertDown}(a, n')$. This naturally exists for both signed and unsigned integers.
- We have a primitive for obtaining the largest Fibonacci number that is not larger than the input integer a. We denote it with $\mathsf{PrevFibo}(a)$.

Fixed-Point Number Related Primitives. We shall now describe some fixed-point operations and their notations. Our fixed-point arithmetic follows the framework of Catrina and Saxena [24](for example, our multiplication protocol is based on that paper) but with several simplifications allowing for a more efficient software implementation.

We assume that our underlying algebraic structure is some ring $\mathbb{Z}_k$.

- The algorithm for fixed-point addition is presented in Algorithm 1. It takes in two secure fixed-point numbers $[\![^f x]\!]$ and $[\![^f y]\!]$ with the same radix-point m. Then it adds the representatives of $[\![^f x]\!]$ and $[\![^f y]\!]$. Then it sets the sum of those to be the representative of the secure fixed-point number $[\![^f z]\!]$, which it will return as the result. We represent this operation with $[\![^f x]\!] + [\![^f y]\!]$. The algorithm for fixed-point subtraction is analogous and will not thus be presented separately.

Algorithm 1: Addition of fixed-point numbers.

Data: $[\![^f x]\!], [\![^f y]\!]$
1 $[\![^f z]\!] \xleftarrow{f} [\![\tilde{x}]\!] + [\![\tilde{y}]\!]$
2 **return** $[\![^f z]\!]$

- We also specify an algorithm for fixed-point addition for four-field signed fixed-point numbers. This algorithm is formalized in algorithm 2.

  We first compute the sum and difference of the absolute values, denoting these with $[\![t]\!]$ and $[\![u]\!]$, respectively. Note that if $abs_y > abs_x$, then we want to obtain $[\![abs_y]\!] - [\![abs_x]\!]$ as the absolute value of the result. The integer in the underlying ring that carries this value is $-u$, so we can use that.

  Now, consider the possible cases. If the signs of $[\![^f x]\!]$ and $[\![^f y]\!]$ are the same, then we simply add the absolute values to obtain the absolute value of the

answer and set the sign of the answer to be one of the signs. Thus, in this case the answer will be $[\![^f z_0]\!] \leftarrow (sgn_x, abs_x + abs_y)$.
If the signs are different, however, mere subtraction or addition does not suffice for obtaining the answer. The sign and the absolute value of the answer also depend on which of abs_x and abs_y is greater.
Thus we set $[\![b]\!]$ to $\mathsf{LTEProtocol}([\![abs_x]\!], [\![abs_y]\!])$.
Now, in the case where $abs_x > abs_y$, the answer is $(sgn_x, [\![u]\!])$ and in the case $abs_x \leq abs_y$, the answer is $(sgn_y, -[\![u]\!])$. We can use the technique for oblivious choice to get that in this case the answer is

$$([\![b]\!]sgn_y + (1 - [\![b]\!])sgn_x, [\![b]\!][\![u]\!] + (1 - [\![b]\!])[\![-u]\!]).$$

We can simplify this part and obtain the answer

$$[\![^f z_0]\!] \leftarrow ([\![b]\!] + sgn_x - 2[\![b]\!]sgn_x, (2[\![b]\!] - 1) \cdot [\![u]\!]).$$

Note that whether the signs of the numbers are the same or different is equivalent to whether sgn_x XOR sgn_y is equal to 0 or 1.
Thus we set $[\![c]\!]$ to $[\![sgn_x]\!]$ XOR $[\![sgn_y]\!]$ and let the final answer be

$$\mathsf{ObliviousChoiceProtocol}([\![c]\!], [\![^f z_1]\!], [\![^f z_0]\!]).$$

The subtraction of four-field signed fixed-point numbers is analogous.

Algorithm 2: Addition of four-field fixed-point numbers.

Data: $[\![^f x]\!], [\![^f y]\!]$
1 $[\![t]\!] \xleftarrow{f} [\![abs_x]\!] + [\![abs_y]\!]$
2 $[\![u]\!] \xleftarrow{f} [\![abs_x]\!] - [\![abs_y]\!]$
3 $[\![b]\!] \leftarrow \mathsf{LTEProtocol}([\![abs_x]\!], [\![abs_y]\!])$
4 $[\![c]\!] \leftarrow [\![sgn_x]\!]$ XOR $[\![sgn_y]\!]$
5 $[\![^f z_0]\!] \leftarrow ([\![sgn_x]\!], [\![abs_x]\!] + [\![abs_y]\!])$
6 $[\![^f z_1]\!] \leftarrow ([\![b]\!] + [\![sgn_x]\!] - 2[\![b]\!][\![sgn_x]\!], (2[\![b]\!] - 1) \cdot [\![u]\!])$
7 $[\![^f z]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![c]\!], [\![^f z_1]\!], [\![^f z_0]\!])$
8 **return** $[\![^f z]\!] = ([\![sgn_z]\!], [\![abs_z]\!])$

- We have a primitive for multiplication of a fixed-point number and an integer. This primitive is formalized in Algorithm 3. Given an integer $[\![a]\!]$ and a fixed-point number $[\![^f x]\!]$, we compute $[\![^f ax]\!]$ by multiplying $[\![a]\!]$ with $[\![\widetilde{x}]\!]$, the representative of $[\![^f ax]\!]$ will be $[\![a\widetilde{x}]\!]$.

Algorithm 3: Multiplication of a private fixed-point number and a private integer.

Data: $[\![^f x]\!], [\![a]\!]$
1 $[\![y]\!] \leftarrow [\![a]\!] \cdot [\![\widetilde{x}]\!]$
2 $[\![^f z]\!] \xleftarrow{f} [\![y]\!]$
3 **return** $[\![^f z]\!]$

- We have a primitive for multiplication of a public and a private fixed-point number. This process is formalized in Algorithm 4.

  This is a somewhat more complicated protocol. It takes in a private fixed-point number $[\![^f x]\!]$ and a public fixed-point number ${}^f y$ and outputs a secret fixed-point number the value of which should be as close to xy as possible. Note that if we are given $[\![^f x]\!]$ and ${}^f y$, then the representative of $[\![^f xy]\!]$ should be $[\![xy2^m]\!]$, but multiplying the representatives of $[\![^f x]\!]$ and ${}^f y$ gives us $[\![xy2^{2m}]\!]$, and thus plain multiplication would give us a number that is 2^m times too large. Moreover, it can also happen that $xy2^{2m}$ is larger than k and causes overflow would happen, for example, in the case of $2^m \geq \sqrt{k}$ and $xy \geq 1$. Both of these conditions are rather natural as we often need to represent numbers larger than 1. Also, we often want to represent numbers with a fine enough granularity that we need to such a value for the radix-point that $2^m \geq \sqrt{k}$.

  The overflow concern can be solved by casting the inputs into a larger ring. We choose $\mathbb{Z}_{k^2}$ because we know that this ring will be able to hold the product of any two values of $\mathbb{Z}_k$. While converting the public fixed-point number to a larger ring can be thought of as a "free" operation, for converting the private fixed-point number up, we need to apply ConvertUp to the representative of $[\![^f x]\!]$ to convert it to $\mathbb{Z}_{k^2}$. We can multiply the cast-up versions of the representatives of $[\![^f x]\!]$ and ${}^f y$, obtaining $[\![s]\!]$, but it cannot be set to be the representative of the result yet, as it is too large by 2^m and not in the correct ring. To correct the first problem, we apply DivideBy2k to $[\![s]\!]$, shifting it to the right by m bits. After that we apply ConvertDown to the result to get back to the correct ring. We set the result of that process to be the representative of the result.

Algorithm 4: Multiplication of a private and a public fixed-point number.

Data: $[\![^f x]\!], {}^f y$
1 $[\![z]\!] \leftarrow \mathsf{ConvertUp}([\![\widetilde{x}]\!], k^2)$
2 $w \leftarrow \mathsf{ConvertUp}(\widetilde{y}, k^2)$
3 $[\![s]\!] \leftarrow [\![z]\!] \cdot w$
4 $[\![t]\!] \leftarrow \mathsf{DivideBy2tok}([\![s]\!], m)$
5 $[\![u]\!] \leftarrow \mathsf{ConvertDown}([\![t]\!], k)$
6 $[\![^f v]\!] \xleftarrow{f} [\![u]\!]$
7 **return** $[\![^f v]\!]$

- We have a primitive for multiplication of two private fixed-point numbers. This is formalized in Algorithm 5. It is very similar to the previous protocol, save for two differences. First, we need to convert both the inputs up in a secret fashion, which can be parallelized. Second, instead of multiplication of a private and public integer in the bigger ring, we have to perform a multiplication of secure integers.

In some applications, where one needs to compute the product of more than two secret fixed-point numbers, it might be sensible to cast the numbers to a bigger ring than $\mathbb{Z}_{k^2}$ so that one might be able to multiply them all in that large ring and then shift the result by the necessary number of bits and cast it back to the original ring. However, we did not use this option as at the time when we implemented those applications, using arbitrary ring sizes was not yet technically possible in the framework we were using for implementation.

Algorithm 5: Multiplication of two private fixed-point numbers.

Data: $[\![^f x]\!], [\![^f y]\!]$
begin in parallel
  $[\![z]\!] \leftarrow \mathsf{ConvertUp}([\![\widehat{x}]\!], k^2)$
  $[\![w]\!] \leftarrow \mathsf{ConvertUp}([\![\widehat{y}]\!], k^2)$
$[\![s]\!] \leftarrow [\![z]\!] \cdot [\![w]\!]$
$[\![t]\!] \leftarrow \mathsf{DivideBy2tok}([\![s]\!], m)$
$[\![u]\!] \leftarrow \mathsf{ConvertDown}([\![t]\!], k)$
$[\![^f v]\!] \xleftarrow{f} [\![u]\!]$
return $[\![^f v]\!]$

- We have a primitive for multiplication of two private four-field field fixed-point numbers.This is formalized in Algorithm 6. This is formally quite similar to the multiplication of three-field fixed point numbers, we only have to XOR the signs together in the end to obtain the sign of the result. However, converting up and down can be cheaper as we convert unsigned integers instead of signed integers.

Algorithm 6: Multiplication of four-field fixed-point numbers.

Data: $[\![^f x]\!], [\![^f y]\!]$
begin in parallel
  $[\![z]\!] \leftarrow \mathsf{ConvertUp}([\![abs_x]\!], k^2)$
  $[\![w]\!] \leftarrow \mathsf{ConvertUp}([\![abs_y]\!], k^2)$
$[\![s]\!] \leftarrow [\![z]\!] \cdot [\![w]\!]$
$[\![t]\!] \leftarrow \mathsf{PublicBitShiftRightProtocol}([\![s]\!], m)$
$[\![u]\!] \leftarrow \mathsf{ConvertDown}([\![t]\!], k)$
$[\![c]\!] \leftarrow [\![sgn_x]\!]$ XOR $[\![sgn_y]\!]$
return $([\![u]\!], [\![c]\!])$

- We have an operation for comparing fixed-point numbers. Essentially, this can be trivially done just by comparing the representatives. We will use $\mathsf{LTEProtocol}([\![^f x]\!], [\![^f y]\!])$ to denote the secure operation that compares secure fixed-point numbers.

Note 1. We note that for Algorithms 4 and 5, the result obtained might differ from

$x \cdot y$ by some very small amount, due to the DivideBy2k losing some data. Usually, small errors are natural when dealing with fixed-point numbers so it would be of no great concern. However, when we are dealing with unsigned fixed-point numbers and the values are close to zero, there might be concern to whether underflow might happen. If the error introduced is greater than the value of the number, then the result can be a small negative number in the signed case. In the unsigned case, however, there are no negative numbers, and, thus, that value would instead correspond to a very large positive value, which would be an unacceptable error.

We note that this error cannot happen. Namely, note that the error is introduced at the operation DivideBy2k. For that operation, as it was noted above, the result is the input divided by a power of two, rounded to the value with a lower absolute value. It is clear that an error that is introduced by rounding to a value with a lower absolute value can not change a non-negative number to a negative one.

Comparative Performance Cost of the Primitives in a Specific Implementation. For easier understanding, we present the relative costs of the given primitives in the framework where we implemented the techniques presented in this thesis. This was the Sharemind framework, which is based on three-party additive secret sharing. That paradigm greatly benefits from parallel composition and for that reason the most important characteristic describing a primitive is the number of rounds of communication. The inputs here presume that unless specified otherwise, the values given are n-bit integers. The fixed-point numbers given in operations are assumed to have n-bit integers as representatives. The floating-point numbers are assumed to have n-bit significands.

For primitives for which we could find the round complexity in existing literature, we added the reference. However, for some primitives, we could not find the round complexity and thus we derived our own estimates by analyzing the code or the pseudocode and verifying the results by running the PDSL optimizer tool. This should be presumed to be the case for any operation for which no reference is given.

Operation	Rounds
$[\![x]\!] + [\![y]\!]$	0 [11]
$[\![x]\!] \cdot y$	0 [11]
$[\![x]\!] \cdot [\![y]\!]$	1 [13]
$\mathsf{LastBit}([\![x]\!])$	0
$\mathsf{MSNZB}([\![x]\!])$	$\log n$
$\mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![x]\!], [\![y]\!])$, b is an n-bit integer	1
$\mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![x]\!], [\![y]\!])$, b is a bit	3
$\mathsf{BitExtraction}([\![x]\!])$	$\log n + 3$ [11]
$\mathsf{LTEProtocol}([\![x]\!], [\![y]\!])$, assumes the most significant bit to be 0	$\log n + 2$ [11]
$\mathsf{LTEProtocol}([\![x]\!], [\![y]\!])$	$\log n + 3$ [11]
$\mathsf{PublicBitShiftRightProtocol}([\![x]\!], k)$	$\log n + 3$ [11]
$\mathsf{ConvertToBoolean}([\![x]\!])$	0
$\mathsf{ConvertBoolToInt}([\![b]\!])$	2 [11]
$\mathsf{ConvertUp}([\![x]\!], k')$	$\log n + 3$
$\mathsf{ConvertDown}([\![x]\!], k')$	0
$\mathsf{GenObliviousChoice}([\![x_0]\!], \ldots, [\![x_{k-1}]\!], [\![\ell]\!])$	$\log n + \log \lfloor \log k \rfloor + 3$
$\mathsf{PrivateBitShiftRightProtocol}([\![x]\!], [\![k]\!])$	$\log n + 5$
cast an n-bit signed integer to float	$3 \log n + 8$
cast an n-bit unsigned integer to float	$2 \log n + 4$
$[\![^F x]\!] + [\![^F y]\!]$	$5 \log n + 24$
$[\![^F x]\!] \cdot {}^F y.$	$2 \log n + 14$
$[\![^F x]\!] \cdot [\![^F y]\!].$	$2 \log n + 16$
$[\![^f x]\!] + [\![^f y]\!]$	0
$[\![^f x]\!] - [\![^f y]\!]$	0
$[\![^f x]\!] \cdot {}^f y$	$2 \log n + 6$
$[\![^f x]\!] \cdot [\![^f y]\!]$	$2 \log n + 7$

Table 1: The round complexity of the presented primitives in the Sharemind 3 setting.

4. HYBRID MODEL

4.1. Introduction

There are two data types that are usually used for real numbers in computation, fixed-point numbers and floating-point numbers. This chapter is about how to combine and switch between these two data types in SMC for better performance.

Fixed-point numbers are essentially integers shifted to the right by a fixed number of bits. Thus fixed-point numbers translate rather naturally to the secure setting. Due to their similarity to integers, standard operations on them are relatively cheap. However, as is the case with normal computing, they usually lack either range, granularity, or require a large amount of memory.

Let us now consider floating-point numbers. Outside of SMC, floating-point numbers are ubiquitous as they are flexible and provide good relative precision. The fact that they also have the exponent variable makes computing functions that are related to powers easier. However, floating-point numbers are complicated by nature. In practice, the algorithms designed for them often contain a number of if-else statements and special cases, such as zero or overflow flags. In the SMC setting, complicated algorithms often translate into costly algorithms, as they tend to have a lot of branching based on special cases. For example, addition of floating-point numbers is very expensive in the Sharemind setting where we implemented our algorithms, as can be seen in Table 1, but this also holds in other frameworks.

In this thesis we study some ways of how to combine floating-point numbers with fixed-point numbers in computing a function. Intuitively, we would like to perform addition and multiplication on fixed-point numbers and evaluate functions that rely on manipulating the exponent on floating-point numbers. Moreover, we want to obtain the range, flexibility, and precision of the floating-point number in the result.

We shall look at four functions: the inverse $f(x) = \frac{1}{x}$, the square root $f(x) = \sqrt{x}$, the exponent $f(x) = e^x$ and the Gaussian error function $f(x) = \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. The reason why we have chosen these four functions is that these functions are necessary and sufficient for the satellite collision problem [47]. Kamm and Willemson provided algorithms for these four functions, however, they used only floating-point numbers, which led to inefficiencies. Often these algorithms had parts were the significand of an existing floating-point number was taken and converted into a floating-point number of its own and then polynomial evaluation was performed on it.

We note that the significand of a floating-point number is much more similar to a fixed-point number than a separate floating-point number. Due to this similarity we attempt to improve these algorithms by using fixed-point numbers instead of floating-point numbers for significands.

We also attempt to improve performance by using fixed-point numbers in some other contexts where this substitution might be beneficial. Note that while sometimes this approach improves results, it does not do so always because converting from floating-point values (or parts of floating-point values) to fixed-point values can be expensive, depending on the circumstances. These conversion costs are sometimes larger than our gains from using a more suitable type for an operation.

Our algorithms are based on the algorithms of Kamm and Willemson [47]. For the algorithms we will describe, we will need the following primitives:

- fixed-point addition, subtraction, and multiplication,
- floating-point addition, subtraction, and multiplication,
- $\mathsf{BitExtraction}(\llbracket x \rrbracket)$,
- $\mathsf{PublicBitShiftRightProtocol}(\llbracket x \rrbracket, k)$,
- $\mathsf{LTEProtocol}(\llbracket x \rrbracket, \llbracket y \rrbracket)$,
- $\mathsf{ObliviousChoiceProtocol}(\llbracket b \rrbracket, \llbracket x \rrbracket, \llbracket y \rrbracket)$,
- $\mathsf{ConvertToBoolean}(\llbracket x \rrbracket)$,
- $\mathsf{ConvertBoolToInt}(\llbracket b \rrbracket)$,
- $\mathsf{GenObliviousChoice}(\llbracket x_1 \rrbracket, \ldots, \llbracket x_k \rrbracket, \llbracket \ell \rrbracket)$,
- $\mathsf{PrivateBitShiftRightProtocol}(\llbracket x \rrbracket, \llbracket k \rrbracket)$,
- $\mathsf{ConvertToFloat}(\llbracket x \rrbracket)$,
- $\mathsf{LastBit}(\llbracket x \rrbracket)$.

The more detailed descriptions of the algorithms can be found in Section 3.4.4.

4.2. Fixed-Point Numbers

Even though we will occasionally need to cope with negative values, we will only represent non-negative fixed-point numbers. This is because fixed-point numbers will be used only for polynomial evaluation, where both the input and the result are guaranteed to be non-negative, making unsigned fixed-point numbers sufficient. Note that we will also use fixed-point numbers in a context where both the inputs and outputs will be relatively small. We, thus, assume that inputs will not be larger than 4 and that outputs will not be larger than 2. This allows us to use fixed-point numbers with a relatively large radix-point. We will, however, have intermediate fixed-point values that are several orders of magnitude larger, and, thus, our fixed-point numbers must be able to hold these values as well. The specific value for the radix-point will depend on the specific algorithm that we use.

Also note that while we generally assume that the underlying ring for the fixed-point numbers is some general $\mathbb{Z}_k$, in this section we will assume that the underlying ring is more specifically $\mathbb{Z}_{2^n}$ for some $n \in \mathbb{Z}_+$. We assume this, because the significand of a floating-point number is a member of a ring $\mathbb{Z}_{2^n}$ and conversion between fixed-point numbers and floating-point numbers is easier if the signifi-

cand of the floating-point number has the same type as the representative of the fixed-point number.

4.2.1. Polynomial Evaluation on Fixed-Point Numbers

As it was noted above, the main difference between this chapter and the article of Kamm and Willemson [47] is that instead of floating-point numbers as it is done in [47], polynomial evaluation will be performed on fixed-point numbers. Thus, it is important to have an algorithm for polynomial evaluation on fixed-point numbers. As mentioned above, since the evaluations will be performed on values which will be small and positive, we will use small unsigned fixed-point numbers. In this section we will describe an algorithm for evaluating a fixed-point polynomial $\sum_{i=0}^{k}(-1)^{s_i}c_i x^i$ at x where x is stored as a private fixed-point value $[\![{}^f x]\!]$. This is formalized in Algorithm 7. A similar algorithm for floating-point numbers has also been described by Kamm and Willemson [47]. Note that the polynomials used are approximation polynomials and thus the final result contains an error that depends on the polynomial used and the errors introduced through fixed-point multiplication.

The non-negative fixed-point coefficients c_i and the sign bits s_i are public. $[\![{}^f x]\!]$ is a private non-negative fixed-point number.

First we need to evaluate $[\![{}^f x^2]\!], [\![{}^f x^3]\!], \ldots, [\![{}^f x^k]\!]$. It is trivial to do this with $k-1$ rounds of multiplications, however, we shall do it in $\lceil \log k \rceil$ rounds because parallel composition and round-efficiency are important to us. In every round we compute the values $[\![{}^f x^{2^i+1}]\!], [\![{}^f x^{2^i+2}]\!], \ldots, [\![{}^f x^{2^{i+1}}]\!]$ by multiplying $[\![{}^f x^{2^i}]\!]$ with $[\![{}^f x^1]\!], [\![{}^f x^2]\!], \ldots, [\![{}^f x^{2^i}]\!]$, respectively (line 4).

Following that, on line 7 we can multiply the powers of x with the respective coefficients ${}^f c_i$ with one round of multiplication, obtaining the values $[\![{}^f y_1]\!] = {}^f c_1 \cdot [\![{}^f x]\!], [\![{}^f y_2]\!] = {}^f c_2 \cdot [\![{}^f x^2]\!], \ldots, [\![{}^f y_k]\!] = {}^f c_k \cdot [\![{}^f x^k]\!]$. We also set $[\![{}^f y_0]\!]$ to the public fixed-point number ${}^f c_0$.

After that we can compute the sums $[\![{}^f y']\!] = \sum_{i:s_i=1} [\![{}^f c_i x^i]\!]$ and $[\![{}^f y'']\!] = \sum_{i:s_i=-1} [\![{}^f c_i x^i]\!]$, respectively, on lines 11 and 13 and find the final result $[\![{}^f y]\!] = [\![{}^f y']\!] - [\![{}^f y'']\!]$ on line 13. We return $[\![{}^f y]\!]$.

We must pay some attention to the fact that we use unsigned fixed-point numbers here. Namely, we should be certain that the final result and the intermediate results are non-negative fixed-point numbers. Note that due to Note 1, it follows that if we multiply two non-negative fixed-point numbers, the results will also be non-negative. Addition of fixed-point numbers is an exact operation, so adding non-negative fixed-point numbers will also result in a non-negative result. From this it follows that the y_i are non-negative numbers.

Thus also y' and y'' are non-negative numbers, as they are sums of non-negative numbers. Now the only possible value that might be negative is $y = y' - y''$. For

this to be negative, it is necessary that y' is smaller than y''.

The polynomials that we use will be such that the output is non-negative on the input we provide. However, multiplication may introduce small errors. Hence the result might be wrong by some small ε. Thus it is necessary to require that the output of the polynomial is larger than this ε in the domain that we evaluate it. We will only use such polynomials that have this property.

Concerning the polynomials, while it may seem at first that approximation polynomials with more terms will automatically give more accurate results than approximation polynomials with less terms, this is not necessarily the case. Consider that, depending on the function, more coefficients may also mean that the coefficients are larger. There are two problems with large coefficients. First, if we want to be able to store larger coefficients, we must decrease the value of the radix-point and, thus, lose granularity and precision. Secondly, the operations may have some small errors. So, if the polynomial has a term that is larger than 10^k and it is multiplied with a value that has an error of ε, then this can result in a total error of $10^k\varepsilon$, which significantly decreases the accuracy.

One should note that this type of error is not due to the original value x being wrong by some ε so that we compute $\sum_{i=0}^{k}(-1)^{s_i}c_i(x+\varepsilon)^i$ instead of $\sum_{i=0}^{k}(-1)^{s_i}c_ix^i$. This would not be a big problem since the polynomial is continuous and, thus, this would not change the result significantly. This error is due to a small error possibly happening every multiplication and thus the result would be $\sum_{i=0}^{k}(-1)^{s_i}c_i(x^i+\varepsilon_i)$ which can have a maximal error of $\sum_{i=0}^{k}|c_i\varepsilon_i|$ in the worst case. Thus, if c_i are small, it might be that we get better results.

We can estimate the error of this algorithm with the following claim.

Claim 1. *Suppose that the fixed-point multiplication algorithm we use has an error with an absolute value no greater than ε. Algorithm 7, when called at value $[\![^f x]\!]$ where $0 < x \leq 1-\varepsilon$ for the polynomial with coefficients $\{(-1)^{s_i}\cdot {}^f c_i\}_{i=0}^{k}$, will return the secret value $[\![^f y]\!]$ where y differs from $\sum_{i=0}^{k}(-1)^{s_i}\cdot c_i x$ by at most $\sum_{i=0}^{k}|c_i|(i-1)\varepsilon$.*

Proof. We first show by induction over i that a term $[\![^f x^{2^i}]\!]$ has a multiplication error at most $(2^i-1)\varepsilon$.

The case for $i=1$ is trivial — one multiplication is performed, thus the result can differ by no more than ε.

Suppose that the multiplication error for $i=k$ is no greater than $(2^i-1)\varepsilon$. Now $[\![^f x^{2^{k+1}}]\!]$ is computed by squaring $[\![^f x^{2^k}]\!]$ and the computed value of $[\![^f x^{2^k}]\!]$

Algorithm 7: Computation of a polynomial on fixed-point numbers.

Data: Takes in a secret fixed point number $[\![^f x]\!]$ where the radix-point is m, the number of bits in the representative of the fixed-point number n and the coefficients $\{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^{k}$ for the approximation polynomial. The function is called as $\mathsf{Poly}([\![^f x]\!], m, n, \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^{k})$.

Result: Outputs a secret fixed-point number $[\![^f y]\!]$ that is the value of the approximation polynomial at point x.

1 $[\![^f x^1]\!] \leftarrow [\![^f x]\!]$
2 **for** $j \leftarrow 0$ **to** $\lceil \log_2(k) \rceil$ **do**
3 **for** $i \leftarrow 1$ **to** 2^j **do** in parallel
4 $[\![^f x^{i+2^j}]\!] \leftarrow [\![^f x^{2^j}]\!] \cdot [\![^f x^i]\!]$
5 $[\![^f y_0]\!] \leftarrow [\![^f c_0]\!]$
6 **for** $i \leftarrow 1$ **to** k **do** in parallel
7 $[\![^f y_i]\!] \leftarrow {}^f c_i \cdot [\![^f x^i]\!]$
8 $[\![^f y']\!], [\![^f y'']\!] \leftarrow {}^f 0$
9 **for** $i \leftarrow 0$ **to** k **do** in parallel
10 **if** $s_i == 0$ **then**
11 $[\![^f y']\!] += [\![^f y_i]\!]$
12 **if** $s_i == 1$ **then**
13 $[\![^f y'']\!] += [\![^f y_i]\!]$
14 $[\![^f y]\!] \leftarrow [\![^f y']\!] - [\![^f y'']\!]$
15 **return** $[\![^f y]\!]$

differs from the real result by no more than $(2^i - 1)\varepsilon$, that is, we might have $x^{2^k} + (2^k - 1)\varepsilon$ instead of x^{2^k}.

Thus the error of $[\![^f x^{2^{k+1}}]\!]$ is no greater than $2(2^k - 1)\varepsilon x^{2^k} + \varepsilon^2 + \varepsilon \leq (2^{k+1} - 1)\varepsilon$.

We now show that a term $[\![^f x^j]\!]$ has a multiplication error at most $(j-1)\varepsilon$.

We likewise do it with induction. Suppose that the claim holds for $j < 2^i$, we shall show it for $j \in (2^i, 2^{i+1})$.

Take some $j \in (2^i, 2^{i+1})$. Let $j = 2^i + j'$ where $j' < 2^i$. The value $[\![^f x^j]\!]$ is computed by multiplying $[\![^f x^{2^i}]\!]$ by $[\![^f x^{j'}]\!]$. The error of this is bounded by $(j' - 1)\varepsilon x^{2^i} + 2^i x^{2^{j'}} \varepsilon + \varepsilon^2 + \varepsilon \leq j\varepsilon$.

From this the claim easily follows.

□

4.2.2. Helper Functions

We shall need a few common functionalities for computing the four functions that we will give algorithms for. Namely, as we need to alter between floating-point numbers and fixed-point numbers, we require functions that help us with converting between the two cases. We will be able to make some assumptions about our inputs — for example, if we take a significand as an input, we know that the respective fixed-point number lies in $[0.5, 1)$. Based on these assumptions and the nature of the operations, we can also make some assumptions about the outputs which can be helpful when we want to convert fixed-point outputs back to floating-point outputs. However, for different functions, these assumptions can be different. Thus we need some variability in these helper functions.

Range Correction. In the case of the inverse, the square root and the exponential, we shall obtain, as the result of fixed-point calculation, a number about which we know that it is approximately in some interval $[2^t, 2^{t+1})$. For example, when we evaluate the inverse function on fixed-point numbers, we start with $[\![^f x]\!]$ about which we know that x is in $[0.5, 1)$. Thus, $\frac{1}{x}$ ought to be in $(1, 2]$.

This property would make it easy and cheap to convert the fixed-point result $[\![^f x]\!]$ back to a floating-point number. We would set $t+1$ as the exponent and $[\![\widehat{x}]\!] \ll n - m - 1$ as the significand and would thus get a very fast conversion. However, the problem here is that the result is *approximately* in $[2^t, 2^{t+1})$. As we will see from the algorithms we will use, the error of the algorithm that we use for converting the fixed-point number back to a floating-point number is not continuous at 2^t or 2^{t+1}.

Namely, if we assume for ${}^f x$ that $x \in [2^t, 2^{t+1})$ but the true value of x is $2^{t+1} + \varepsilon$, for example, then we will obtain either a notably different floating-point number at best or something that is not even a properly formatted floating-point number at worst. Even in the case where $x = 2^{t+1}$, where the value fails to be in the correct interval by the least possible amount, shifting x to the left by $n - m - t$ bits (as is

done to obtain a significand from a fixed-point number) will result in 0, which is not a proper significand for our case.

Thus we need a method for guaranteeing that the result is in $[2^t, 2^{t+1})$. For this purpose, we present Algorithm 8. The idea is quite simple. We have a fixed-point number $[\![^f y]\!]$ and we know that y is approximately in $[2^t, 2^{t+1})$.

We interpret this rather loosely. We shall assume that y is either in $[0, 2^t)$, $[2^t, 2^{t+1})$ or $[2^{t+1}, 2^{t+1} + 2^t)$. In the first case we want to give $[\![^f 2^t]\!]$ as the answer, in the second case we want to give $[\![^f y]\!]$ as it is, without changing it and in the third case we want to return $[\![^f (2^{t+1} - 2^{-m})]\!]$, i.e. representing the largest number that we can represent that is smaller than 2^{t+1} (this is also the largest element in the set $[2^t, 2^{t+1})$ that we can represent).

This gives us a result that is in $[2^t, 2^{t+1})$. Note that this operation will not lose precision, on the contrary, it is possible that it corrects some error of the polynomial. For example, if computing $\frac{1}{x}$ where $x \in [0.5, 1)$, it is possible for the polynomial to give an answer that is smaller than 1. If we replace that answer with 1, it is bound to be closer to the true answer. The only exception to this is when 2^t would be the correct answer and we return instead $2^t - 2^{-m}$ but that is a very small error.

We assumed that y is in either $[0, 2^t)$, $[2^t, 2^{t+1})$ or $[2^{t+1}, 2^{t+1} + 2^t)$. Note that the fact which interval y is in depends on precisely two bits of y. Let the bits of $\widetilde{y}$ be $\{y_i\}_{i=0}^{n-1}$.

The most significant nonzero bit is either y_{t+m+1}, y_{t+m} or some less significant bit, depending on in which of the three abovementioned intervals y is. Also note that if $y_{t+m+1} = 1$, then $y_{t+m} = 0$, because otherwise ${}^f y \geq 2^{t+1} + 2^t$.

Thus, there are three possibilities: either $y_{t+m+1} = 1$ and $y_{t+m} = 0$, $y_{t+m+1} = 0$ and $y_{t+m} = 1$ or $y_{t+m+1} = 0$, and $y_{t+m} = 0$. These correspond to the respective three intervals to which y can belong. Thus, if $y_{t+m+1} = 1$, then we ought to return $[\![^f (2^{t+1} - 2^{-m})]\!]$, if $y_{t+m} = 1$, then we ought to return $[\![^f y]\!]$ and otherwise we should return $[\![^f 2^t]\!]$. Thus, we need to extract the bits y_{t+m+1} and y_{t+m} and then perform two oblivious choices based on their values.

It might be that we can rule out one or two of the three possibilities. We can give upper and lower bounds to the values of the polynomial evaluation algorithm, learning that the result of an input will be in $[2^t - \varepsilon_0, 2^{t+1} + \varepsilon_1)$. If we know that $\varepsilon_0 \leq 0$, then we know that the output of the algorithm cannot be in $[0, 2^t)$. Likewise, if we know that if $\varepsilon_1 < 0$, then the output of the algorithm cannot be in $[2^{t+1}, 2^{t+1} + 2^t)$. If both of these conditions hold, then we do not have to use the correction algorithm. If one of them holds, then we can only perform a part of the correction algorithm. We will specify public flags to signify what parts will be performed. These flags only depend on the function that we evaluate and the polynomial used so they will not leak information about the data.

We shall now describe the range-correcting protocol which is formalized in Algorithm 8.

The inputs are the following: the fixed-point number $[\![^f y]\!]$ that is the input, the number of bits n of the representative type, the radix-point m, and three public parametres t, b_0 and b_1 that describe the potential range where $[\![^f y]\!]$ could be. We note that depending on the context, we might not have to perform both of the checks. As discussed above, it might be possible that we know that the input will certainly be no smaller than 2^t or that it will certainly be smaller than 2^{t+1}. Thus, we shall give the algorithm two public flags b_0 and b_1 that describe it: we know that y is in $[2^t - b_0 2^t, 2^{t+1} + b_1 2^t)$ where $b_0, b_1 \in \{0,1\}$.

We first extract the bits $\{[\![u_i]\!]\}_{i=0}^{n-1}$ from the representative of $[\![^f y]\!]$ on line 1. What will be done next depends on the values of the public flags b_0 and b_1. If $b_0 = 0$ and $b_1 = 0$ then there is no need for this algorithm, thus, we shall assume that at least one of the flags must be equal to 1.

If $b_0 = 1$ and $b_1 = 0$, then we know that y is in $[0, 2^{t+1})$, and hence the bit $[\![u_{t+m}]\!]$ describes whether y is in $[0, 2^t)$ or $[2^t, 2^{t+1})$. If $u_{t+m} =$ is equal to 0 then y is in $[0, 2^t)$ and if it is equal to 1, then y is in $[2^t, 2^{t+1})$. Thus we have to obliviously choose between $[\![^f y]\!]$ and $[\![^f 2^t]\!]$ based on the bit $[\![u_{t+m}]\!]$. On line 3 we set the answer to be equal to

$$\mathsf{ObliviousChoiceProtocol}([\![u_{t+m}]\!], [\![^f z]\!], [\![^f 2^t]\!]).$$

If $b_0 = 0$ and $b_1 = 1$, then we know that y is in $[2^t, 2^{t+1} + 2^t)$, then the bit $[\![u_{t+m+1}]\!]$ describes whether y is in $[2^t, 2^{t+1})$ or $[2^{t+1}, 2^{t+1} + 2^t)$ — if $u_{t+m+1} = 0$, then it is in $[2^t, 2^{t+1})$ and if $u_{t+m+1} = 1$, then it is in $[2^{t+1}, 2^{t+1} + 2^t)$. Thus we have to obliviously choose between $[\![^f 2^{t+1} - 2^{-m}]\!]$ and $[\![^f y]\!]$ based on the bit $[\![u_{t+m+1}]\!]$ — on line 5 we set the answer to be equal to

$$\mathsf{ObliviousChoiceProtocol}([\![u_{t+m+1}]\!], [\![^f 2^{t+1} - 2^{-m}]\!], [\![^f y]\!]).$$

If $b_0 = 1$ and $b_1 = 1$, then we know that y is in $[0, 2^{t+1} + 2^t)$, and hence we need to run both checks. We first run the check based on the value of u_{t+m}. Thus, we set first $[\![^f y]\!]$ to $\mathsf{ObliviousChoiceProtocol}([\![u_{t+m}]\!], [\![^f y]\!], [\![^f 2^t]\!])$ and then $[\![^f y]\!]$ to $\mathsf{ObliviousChoiceProtocol}([\![u_{t+m+1}]\!], [\![^f 2^{t+1} - 2^{-m}]\!], [\![^f z]\!])$. It is easy to verify that in all three possible cases, the function returns the respectively correct result.

Note that it is important in which order the checks are performed. Suppose that we performed the check based on u_{t+m+1} before the check based on u_{t+m} and that our input was in $[2^{t+1}, 2^{t+1} + 2^t)$. Then $u_{t+m+1} = 1$ and $u_{t+m} = 0$. We would first set the answer to $[\![^f 2^{t+1} - 2^{-m}]\!]$ as it ought to be, but after then, the next check would set the answer to $[\![^f 2^t]\!]$, which would be the wrong result. Thus the order of the checks is important.

We shall refer to this algorithm as Correction.

Note that a polynomial in a given setting gives us could give a result where $\varepsilon_0 \leq 0$, $\varepsilon_1 < 0$, or both of these conditions hold. However, even if neither of them holds, then we can make a small modification to make one of them to hold. More specifically, if the constant member of the polynomial that was used to obtain

Algorithm 8: Correcting the range of a fixed-point number.

Data: $[\![^f y]\!], t, m, n, b_0, b_1$

Result: Takes in a secret fixed-point number $[\![^f y]\!]$, the number of bits n, the position of the radix-point$m$, and integer t and two bits b_0 and b_1 such that we know from prior data that $y \in [2^t - b_0 2^t, 2^{t+1} + b_1 2^t)$. Outputs a fixed-point number that is equal to ${}^f y$ if $y \in [2^t, 2^{t+1})$, equal to ${}^f 2^t$ if $y \in [0, 2^t)$, and equal to ${}^f 2^{t+1} - 2^{-m}$ if $y \in [2^{t+1}, 2^{t+1} + 2^t)$.

$\{[\![u_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtraction}([\![\hat{y}]\!])$

if $b_0 == 1$ **then**

  $[\![^f y]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![u_{t+m}]\!], [\![^f y]\!], [\![^f 2^t]\!])$

if $b_1 == 1$ **then**

  $[\![^f y]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![u_{t+m+1}]\!], [\![^f 2^{t+1} - 2^{-m}]\!], [\![^f y]\!])$

return $[\![^f y]\!]$

the result is ${}^f c$, then we can replace it either with ${}^f(c + \varepsilon_0)$ or ${}^f(c - \varepsilon_1 - 2^{-m})$. Modifying the constant member of the polynomial does not alter any other errors and thus it is safe to assume that the result will be either no smaller than 2^t, or smaller than 2^{t+1}, respectively. However, note that this can affect the precision of the result, and the efficiency gain tends to be rather small. Thus the question whether to add or deduce the respective ε from the constant term of the polynomial is a question of speed-precision tradeoff and we will leave it to be decided by the implementer.

Thus we can summarize the result of this subsection with the following claim.

Claim 2. *Algorithm 8, when given a secret fixed-point number $[\![^f y]\!]$, the number of bits n, the position of the radix-pointm, and integer t and two bits b_0 and b_1 so that $y \in [2^t - b_0 2^t, 2^{t+1} + b_1 2^t)$, outputs a fixed-point number that is equal to ${}^f y$ if $y \in [2^t, 2^{t+1})$, equal to ${}^f 2^t$ if $y \in [0, 2^t)$, and equal to ${}^f 2^{t+1} - 2^{-m}$ if $y \in [2^{t+1}, 2^{t+1} + 2^t)$.*

4.2.3. Converting a Fixed-Point Number to a Floating-Point Number

We will need a specific functionality for the Gaussian error function. Namely, unless in the other cases where the evaluation of the polynomial gives an answer that fits into an interval $[2^t, 2^{t+1})$, the case of the error function is more complicated and, thus, polynomial evaluation gives us fixed-point numbers about which we know that they belong to some interval $[2^{t-1}, 2^{t+1})$.

Thus, in this subsection we shall present Algorithm 9 for changing a fixed-point number to a floating-point number, on the condition that we know that it belongs to $[2^{t-1}, 2^{t+1})$. We will to refer it as FixToFloatConversion.

Essentially, we need to convert the fixed-point number $[\![^f y]\!]$ to the significand by shifting it to the left by $n - m - t + 1$ bits (if it is in $[2^{t-1}, 2^t)$) or by $n - m - t$

Algorithm 9: Converting a fixed-point number to a floating-point number.

Data: Takes in a secret positive fixed-point number $[\![^f y]\!]$, the number of bits n, the position of the radix-point m, and integer t such that we know from prior data that $y \in [2^{t-1}, 2^{t+1})$. We call this function $\mathsf{FixToFloatConversion}([\![^f y]\!], t, m, n)$

Result: Outputs a floating-point number $[\![^F x]\!]$ that represents approximately the same number as $[\![^f y]\!]$

1 $\{[\![u_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtraction}([\![\widehat{y}]\!])$
2 $[\![\sigma]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![u_{t+m}]\!], [\![\widehat{y}]\!] \cdot 2^{n-t-m-1}, [\![\widehat{y}]\!] \cdot 2^{n-t-m})$
3 $[\![E]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![u_{t+m}]\!], [\![t+q+1]\!], [\![t+q]\!])$
4 $[\![^F x]\!] \leftarrow ([\![1]\!], [\![E]\!], [\![\sigma]\!])$
5 **return** $[\![^F x]\!]$

bits (if it is in $[2^t, 2^{t+1})$). We also need to set the exponent to either $q+t+1$ or $q+t$.

This choice depends on whether ${}^f y \geq 2^t$ or not. This information is contained in the $(t+m)$-th bit of y — if it is 0, then ${}^f y < 2^t$ and if it is 1, then ${}^f y \geq 2^t$.

We start by extracting the bits of $[\![\widehat{y}]\!]$ on line 1. Let the bits be $\{[\![u_i]\!]\}_{i=0}^{n-1}$.

We then use only the $(t+m)$-th bit $[\![u_{t+m}]\!]$ to first perform oblivious choice on both the candidates for the exponent (either $[\![t+q+1]\!]$ or $[\![t+q]\!]$) and the candidates for the significand (either $[\![\widehat{y}]\!] \cdot 2^{n-t-m-1}$ or $[\![\widehat{y}]\!] \cdot 2^{n-t-m}$). We then set the sign of the resulting floating-point number to 1 (because we are using non-negative fixes as a default) and, thus, obtain the result.

We can summarize this subsection with the following claim.

Claim 3. *Algorithm 9, when given a fixed-point number $[\![^f y]\!]$ and integer t such that $y \in [2^{t-1}, 2^{t+1})$, outputs the secure floating-point number $[\![^F y]\!]$.*

4.3. Inverse

We shall describe four algorithms using both floating-point numbers and fixed-point numbers. We shall start with the inverse function as the method works most easily on it. This is because computing the inverse breaks very naturally into two distinct parts, the first of which is manipulating the exponent and the second is evaluating a polynomial on the significand.

We will not include here the concrete polynomials that we use due to several reasons. First, different polynomials should be used for different precision levels. Second, even for a single given precision level, there are several possible polynomials that one can use. Third, we believe that the particular polynomials used do not give any particular insight and are not interesting for the reader. The polynomials we used for testing can be found in Chapter 7. For similar reasons, we also did not include the concrete polynomials used for the other protocols in this

chapter that use some kind of approximation polynomial. Those polynomials can likewise be found in Chapter 7.

In this section we will discuss computing the inverse of a secure floating-point number. This is formalized in Algorithm 10.

Let us first note that while usually, in the case of floating-point algorithms, extra attention must be paid to zero, because constraints that apply to ordinary floating-point numbers do not apply here, in the case of the inverse, $\frac{1}{0}$ does not make any sense. Thus, it is assumed for this algorithm that the input is not zero.

We know that our input is $x = (-1)^{1-s_x} \cdot 2^{e_x} \cdot \tau_x$ and we want to compute the floating-point number representing $\frac{1}{x}$ or a suitably close approximation of it. Obviously $\frac{1}{x} = ((-1)^{1-s_x})^{-1} \cdot (2^{e_x})^{-1} \cdot (\tau_x)^{-1}$. Because the inverse operation does not change the sign, we can take $[\![s_{\frac{1}{x}}]\!] = [\![s_x]\!]$. Finding the exponent of the result is similarly easy — $(2^{e_x})^{-1} = 2^{-e_x}$. We will show later that we can take $[\![E_{\frac{1}{x}}]\!] = [\![2q - E_x + 1]\!]$.

The difficult part is computing $[\![^f(\tau_x)^{-1}]\!]$. Because $\tau_x \in \left[\frac{1}{2}, 1\right)$ which is a rather small interval, computing its inverse can be done using polynomial evaluation. The specific polynomials we use will be specified in Section 7.2. Hence, given some approximation polynomial P, we would get that $(\tau_x)^{-1} = P(\tau_x)$ and thus that $\frac{1}{x} = (-1)^{1-s_x} \cdot 2^{-e_x} \cdot P(\tau_x)$. This is, in fact, the core idea for our inverse function algorithm, but some modifications are necessary.

First, we may not be able to simply give $-e_x + q$ and $P(\tau_x)$ as the exponent and the significand of the result as the significand of the result should represent a number in $\left[\frac{1}{2}, 1\right)$, i.e $\tau_{\frac{1}{x}}$ should be in $\left[\frac{1}{2}, 1\right)$. If we apply a polynomial P to τ_x, then we obtain a value that is approximately equal to $\frac{1}{\tau_x}$. However, as $\tau_x \in [\frac{1}{2}, 1)$, then $P(\tau_x) \approx \frac{1}{\tau_x} \in (1, 2]$.

The interval $(1, 2]$ is not the desired $\left[\frac{1}{2}, 1\right)$, however, it is rather simple to obtain $\left[\frac{1}{2}, 1\right)$ from $(1, 2]$. We note that $2^{-e_x} \cdot P(\tau_x) = 2^{-e_x+1} \cdot \frac{P(\tau_x)}{2}$ and that $\frac{P(\tau_x)}{2}$ is approximately in $\left(\frac{1}{2}, 1\right]$. It is problematic that $\frac{P(\tau_x)}{2}$ is *approximately* in the interval as well that this interval contains the point 1. However, we can get solve both of these by applying the Correction protocol (Algorithm 8).

One additional problem is how to apply the polynomial. While a significand is, in essence, a fixed-point number with a radix-point equal to its number of bits, the value of the radix-point means that we can only represent numbers in

Algorithm 10: Inverse of a floating-point number.

Data: Takes in a secret floating-point number $[\![^F x]\!] = ([\![s_x]\!], [\![E_x]\!], [\![\sigma_x]\!])$, the bias of the exponent q and the radix-pointof the corresponding fixed-point number m, the coefficients $\{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k$ for computing the fixed-point polynomial, the number of bits of the fixed-point number n and flags b_0 and b_1 that detail how the Correction algorithm will be performed. Assumes that $x \neq 0$.

Result: Outputs a secret floating-point number $[\![^F y]\!]$ so that $y \approx \frac{1}{x}$.

1 $[\![^f \sigma]\!] \xleftarrow{f} \mathsf{PublicBitShiftRightProtocol}([\![\sigma_x]\!], n-m)$
2 $[\![^f t]\!] \leftarrow \mathsf{Poly}([\![^f \sigma]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$
3 $[\![^f t]\!] \leftarrow \mathsf{Correction}([\![^f t]\!], 0, m, n, b_0, b_1)$
4 **return** $[\![^F y]\!] = ([\![s_x]\!], [\![2q - E_x + 1]\!], \widehat{[\![t]\!]} \cdot 2^{n-m-1})$

$[0,1)$. This would mean that we could not use coefficients, intermediate values, or results larger than 1. This would be a restriction that would make polynomial approximation infeasible.

The solution is to convert the significand to the fixed-point format with a different radix-point for performing the polynomial evaluation.

Let m be such a radix-point that we would not run into overflow errors when evaluating the respective polynomial. That is, we should be able to represent the coefficients, the results, and the intermediate values when evaluating the polynomial with values from $[0.5,1)$ with these radix-point numbers. Note that the value of this m depends on the specific polynomial that we use, but we generally assume that $m < n$. We thus arrive to Algorithm 10.

Converting from a fixed-point format with a radix-point n to a fixed-point format with radix-point m is essentially dividing by 2^{n-m}. This, in turn, can be closely approximated by shifting the secret integer to the right by $n-m$ bits. Thus, what happens is that on line 1 we apply $\mathsf{PublicBitShiftRightProtocol}([\![\sigma_x]\!], n-m)$ and obtain a fixed-point number that we denote with $[\![^f \sigma]\!]$.

We shall now apply the polynomial P to the $[\![^f \sigma]\!]$ by calling the protocol $\mathsf{Poly}([\![^f \sigma]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$, obtaining the fixed-point number $[\![^f t]\!]$ where $t \approx \frac{1}{\tau_x}$.

This is done on line 2. After that we apply Correction on $[\![^f t]\!]$ in order to be certain that t belongs to $[1,2)$. This is done on line 3.

Note that we should now divide t by 2 and then convert it up to the significand format where $m = n$. However, as converting is essentially multiplying by 2^{n-m}, we can do both of these operations by simply multiplying $\widetilde{t}$ by 2^{n-m-1}.

We can summarize this subsection with the following claim.

Claim 4. *Algorithm 10, when given a secret floating-point number $[\![^F x]\!]$, outputs*

a secret floating-point number $[\![^F y]\!]$ *where* $y \approx \frac{1}{x}$.

4.4. Square Root

In this section we shall present an algorithm for computing the square root of a secure floating-point number $[\![^F x]\!]$. This will be presented in Algorithm 11.

First we note that since the square root of a negative number is not defined for real numbers, we assume that the floating-point number x is non-negative. Thus, we do not use the sign bit $[\![s_x]\!]$ at all in our computations because we assume that it is 1. In effect, the function we compute is $\sqrt{|x|}$.

Square root is also a function that naturally splits into manipulations of the exponent and manipulations of the significand.

If $x = 2^{e_x}\tau_x$, then $\sqrt{x} = 2^{\frac{e_x}{2}}\sqrt{\tau_x}$. , where $\frac{e_x}{2}$ and $\sqrt{\tau_x}$ can be computed independently.

Concerning $\sqrt{\tau_x}$, this is essentially computing the square root of a fixed-point number about which we know that it belongs to $[0.5, 1)$. This can be naturally evaluated using approximation polynomials. However, as noted in the previous section, the significand is a fixed-point number with 0 bits before the radix point and n bits after it, which is suboptimal as we cannot use values that are larger than 1. We would rather use a fixed-point number with m bits after the radix point and $n-m$ bits before it, for a suitably-picked m. This requires shifting $[\![\sigma_x]\!]$ to the right by $n-m$ bits.

We start the Algorithm 11 by calling PublicBitShiftRightProtocol($[\![\sigma_x]\!], n-m$) on line 1 of our algorithm and setting the result to be the representative of the fixed-point number $[\![^f \sigma]\!]$. After that we perform polynomial evaluation, i.e. we will call Poly($[\![^f \sigma]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^{k}, m, n$) on line 5. The polynomial we will use will be specified in Section 7.2.

Now let us consider what we must do with the exponent. We get as input $[\![E_x]\!] = [\![e_x + q]\!]$, and we would like to return a value that is close to $[\![\frac{e_x}{2} + q]\!]$. Seeing that $\frac{e_x}{2} + q = \frac{E_x}{2} + \frac{q}{2}$ and that using the function PublicBitShiftRightProtocol is very close to dividing by two, a natural approach would be first computing PublicBitShiftRightProtocol($[\![E_x]\!], 1$) and then adding $\frac{q}{2}$ to the result. However, there are problems with this approach. First, as q is odd, $\frac{q}{2}$ is not an integer. Second, if E_x is odd, then $E_x \gg 1$ is not equal to $\frac{E_x}{2}$ but to $\frac{E_x}{2} - 0.5$. In the case of the exponent, simply rounding those values off would result in a significant relative error.

Thus, we slightly need to modify our algorithm. Let b be the last bit of E_x. Then we can say that

$$\frac{e_x}{2} + q = E_x \gg 1 + 0.5b + q \gg 1 + 0.5 = E_x \gg 1 + q \gg 1 + 1 + 0.5(b-1).$$

Now, if $b = 1$, then the value we want is $E_x \gg 1 + q \gg 1 + 1$, which can be easily computed and represented. However, if $b = 0$, then the result would be $E_x \gg 1 + q \gg 1 + 0.5$, which is not an integer and, thus, storing it accurately is a problem.

Suppose that, using the approximation polynomial, we have computed the fixed-point number $[\![^f t]\!]$. Here $t \approx \sqrt{\tau_x}$. Note that approximately, $t \in \left[\sqrt{\frac{1}{2}}, 1\right)$. We now note that

$$t \cdot 2^{E_x \gg 1+q \gg 1+0.5} = t\sqrt{\frac{1}{2}} \cdot 2^{E_x \gg 1+q \gg 1+1},$$

where $t\sqrt{\frac{1}{2}}$ is approximately in $\left[\frac{1}{2}, \sqrt{\frac{1}{2}}\right)$ and $E_x \gg 1 + q \gg 1 + 1$ is an integer, making them potential valid candidates for the significand and the exponent of the result. Thus, in the case where $b = 0$, we can multiply the significand with $\sqrt{\frac{1}{2}}$ and set the exponent to $E_x \gg 1 + q \gg 1 + 1$. By doing this we will get a result that accurately represents the correct square root, provided that we shift the result to the left by $n - m$ bits and that the Correction algorithm is applied. The choice between the two cases can be performed with oblivious choice.

In Section 4.2.2 we noted that the question about which flags to use in Correction and whether to add or subtract a respective ε from the constant member of the polynomial is left up to the implementation. However, we will not leave it undecided for the square root, as the case is slightly different here.Namely, even if we know that $t \in \left[\sqrt{\frac{1}{2}} - \varepsilon_1, 1 + \varepsilon_2\right)$ and, thus, use $t + \varepsilon_1$ instead of t, it is not guaranteed that $\sqrt{\frac{1}{2}} \cdot (t + \varepsilon_1)$ will be larger than or equal to 0.5, as secure fixed-point multiplication can introduce additional errors.

However, if we modify the polynomial by using $t - \varepsilon_2$ instead of t, then on the one hand, we can be sure that $t - \varepsilon_2 \in \left[\frac{1}{2}, 1\right)$. On the other hand, when we multiply $t - \varepsilon_2$ by $\sqrt{\frac{1}{2}}$, then the result may be smaller than 0.5 but certainly cannot be larger than 1. Thus, we will apply Correction to t' with $b_0 = 1$ and $b_1 = 0$. We will assume that the polynomial will be chosen in such a way that $t < 1$. We will also in parallel apply Correction to t with $b_0 = 1$ and $b_1 = 0$ because of the case when $x = 0$. We will explain later why this is necessary. Now

that we have considered the necessary pieces of the algorithm, we will present it in Algorithm 11.

We get as input the floating-point number $[\![^F x]\!]$, the bias q, the radix-point m of the fixed-point numbers that we will use, n that describes how many bits are in the significand of x and the evaluation polynomial $\{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k$.

We start by shifting the significand σ_x to the right by m bits on line 1 to get a secure fixed-point number $[\![^f \sigma]\!]$ that represents the same value as τ_x. We then evaluate the polynomial P at σ by calling $\mathsf{Poly}([\![^f \sigma]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$ at line 5 and setting the result to $[\![^f t]\!]$. On line 6, we multiply $[\![^f t]\!]$ with ${}^f\left(\frac{\sqrt{2}}{2}\right)$ and denote the product with $[\![^f t']\!]$.

We then compute the last bit $[\![b]\!]$ of E_x on line 2 by using the LastBit protocol. We compute $[\![E_x \gg 1]\!]$ by calling $\mathsf{PublicBitShiftRightProtocol}([\![E_x]\!], 1)$ on line 3 and denoting the result with $[\![E]\!]$. Now note that since the value of the exponent is the same for both possible values of b, on line 4 we thus compute the exponent of the result by adding $1 + q \gg 1$ to $[\![E]\!]$. We will denote this with $[\![E']\!]$.

Then, on line 8 we set $[\![^f t']\!]$ to $\mathsf{Correction}([\![^f t']\!], -1, m, n, 1, 0)$. Next, on line 10 we choose obliviously between the $[\![\widehat{t}]\!]$ and corrected $[\![\widetilde{t'}]\!]$ using the bit $[\![b]\!]$ and denote the result with $[\![t'']\!]$. We then multiply $[\![t'']\!]$ with 2^{n-m} on line 11 to make it into a valid significand, obtaining $[\![\sigma']\!]$.

We finish the algorithm by setting $[\![^F x']\!]$ to $([\![1]\!], [\![E']\!], [\![\sigma']\!])$ which is the value that we return.

Let us consider what happens if $x = 0$. We shall see that in that case the answer will be very close to 0. In this case, $\sigma_x = 0$ and $e_x = 0$.

The exponent of the result will be set to $\frac{q}{2}$ which is not the exponent for zero, however, as we use $q = 2^{14} - 1$, if $z \in [2^{\frac{q}{2}}, 2^{\frac{q}{2}+1})$, then $z \approx 10^{-2500}$ which is negligible. Thus, if the significand is in the correct format, the result will be a floating-point practically equal to zero.

We now have to consider whether the first bit of the significand is 1 for the number to be a correctly-formed floating-point number.

We take σ_x as a fixed-point number and since it is equal to 0, we will obtain ${}^f 0$ if we shift it to the right by $n - m$ bits. We apply the approximation polynomial to the fixed-point number 0 and obtain the constant member of that specific polynomial. A number of useful approximation polynomials that we use have constant members in $[0, 0.5)$. Thus it is necessary that Correction is applied to the result.

Note that the last bit of $E_x = e_x + q = q$ is 1 and, thus, oblivious choice will give us $[\![^f t]\!]$. Thus we see why Correction must also be applied to $[\![^f t]\!]$. However, this is necessary due to our choice of polynomials. If we used polynomials where the constant member was larger than 1, another flag would be necessary for the Correction algorithm. If we used a polynomial with the constant member in $[0.5, 1)$, we would not need to use the Correction algorithm at all.

Now we can be sure that we obtain a legal floating-point number when the

Algorithm 11: Square root of a floating point number.

Data: Takes in a a secret floating-point number $[\![^F x]\!] = ([\![s_x]\!], [\![E_x]\!], [\![\sigma_x]\!])$, the bias of the exponent q and the radix-point of the corresponding fixed-point number m, the coefficients of the approximation polynomia $\{^f c_i\}_{i=0}^k$ in fixed-point format and the number of bits of the fixed-point number n. Assumes $x \geq 0$.

Result: Outputs a secret floating-point number with a value that is approximately equal to $\sqrt{x}$.

1 $[\![^f \sigma]\!] \leftarrow \mathsf{PublicBitShiftRightProtocol}([\![\sigma_x]\!], n-m)$
2 $[\![b]\!] \leftarrow \mathsf{LastBit}([\![E_x]\!])$
3 $[\![E]\!] \leftarrow \mathsf{PublicBitShiftRightProtocol}([\![E_x]\!], 1)$
4 $[\![E']\!] \leftarrow [\![E]\!] + q \gg 1 + 1$
5 $[\![^f t]\!] \leftarrow \mathsf{Poly}([\![^f \sigma]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$
6 $[\![^f t']\!] \leftarrow [\![^f t_1]\!] \cdot {}^f \sqrt{\frac{1}{2}}$
7 **begin** in parallel
8 $\quad [\![^f t]\!] \leftarrow \mathsf{Correction}([\![^f t_2]\!], -1, m, n, 1, 0)$
9 $\quad [\![^f t']\!] \leftarrow \mathsf{Correction}([\![^f t_2]\!], -1, m, n, 1, 0)$
10 $[\![t'']\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![b]\!], [\![^f t]\!], [\![^f t']\!])$
11 $[\![\sigma']\!] \leftarrow [\![t']\!] \cdot 2^{n-m}$
12 **return** $[\![^F \sqrt{x}]\!] = ([\![1]\!], [\![E']\!], [\![\sigma']\!])$

input is 0.

We can summarize this subsection with the following claim.

Claim 5. *Algorithm 11, when given a secret floating-point number $[\![^F x]\!]$, outputs a secret floating-point number $[\![^F y]\!]$ where $y \approx \sqrt{x}$.*

4.5. Exponential

Computing the exponential function is somewhat more difficult as this problem does not partition as neatly into a part where we evaluate a polynomial on fixed-point numbers and a part where we manipulate the exponent as we do in the case of the inverse function and the square root. We shall see that it is possible, albeit potentially expensive.

We shall now describe how to compute the exponent of a floating-point number as it will be given in Algorithm 12. We are given $[\![^F x]\!]$ and we have to compute $[\![^F e^x]\!]$.

We note that $e^x = 2^{\log_2 e \cdot x}$ so we can instead evaluate 2^y where $y = \log_2 e \cdot x$. This should be more natural to obtain, as computing a specific power of 2 is close to manipulating exponents. Thus, on line 1, we multiply $[\![^F x]\!]$ with ${}^F \log_2 e$, obtaining $[\![^F y]\!]$.

It is possible to split y into an exponent-part and significand-part, however, this is not as easy as in the case of inverse and square root.

Namely, let $y = [y] + \{y\}$ where $[y] \in \mathbb{Z}$ and $\{y\} \in [0,1)$. Note that $2^{\{y\}} \in [2^0, 2^1)$ and that thus $2^y = 2^{[y]} \cdot 2^{\{y\}} = 2^{[y]+1} \cdot \frac{2^{\{y\}}}{2}$ where $\frac{2^{\{y\}}}{2} \in \left[\frac{1}{2}, 1\right)$. Also note that $[0,1)$ is a relatively small interval which means that polynomial evaluation can be used in that interval. Thus we could set s_{e^x} to 1, E_{e^x} to $[y]+1+q$ and τ_{e^x} to $\frac{2^{\{y\}}}{2}$ where $\frac{2^{\{y\}}}{2}$ is computed with the respective polynomial.

We will use this idea of splitting a number in this way as the main idea behind our algorithm, however, the problem is that splitting the floating-point number y into an integer part and a fractional part is nontrivial and potentially expensive. Also, this requires conversions between different types in a secure setting, which can in turn be expensive.

Concerning how to obtain the integer part, we note that if $y \geq 0$ and $e_y > 0$, then the first e_y bits of σ_y correspond to the bits of $[y]$ (the integer part of y) and the last $n - e_y$ bits of the s_y correspond to $\{y\}$ (the fractional part of y). If $y \geq 0$ and $e_y \leq q$, then the integer part $[y]$ is zero. Thus, if $y \geq 0$, both in the case where $E_y \geq q$ and in the case where $E_y < q$, shifting σ_y to the left by $n - e_y$ bits gives us $[y]$. This is done on line 2. If $y < 0$, then, analogously, similar properties will hold. We will discuss that case later.

Now we have obtained the integer part of y. However, in order to obtain the fractional part of y we need to deduct $[y]$ from y and for that $[y]$ and y must have the same type. Thus we shall convert the integer $[y]$ into a floating-point number $[\![^F [y]]\!]$ by using $\mathsf{ConvertToFloat}([\![[y]]\!])$ on line 3. We can then compute $[\![^F \{y\}]\!]$ by deducting $[\![^F [y]]\!]$ from $[\![^F y]\!]$ on line 4.

However, now $[\![^F \{y\}]\!]$ has the wrong type. We have established that it is preferrable to perform polynomial evaluation on fixed-point numbers and thus we have to convert $[\![^F \{y\}]\!]$ to a fixed-point number $[\![^f \{y\}]\!]$.

This can be done by using the $\mathsf{PrivateBitShiftRightProtocol}$. We shall shift $\sigma_{\{y\}}$ to the right by $e_y + n - m$ bits, we obtain the fixed-point number $[\![^f \{y\}]\!]$, on which we can perform polynomial evaluation. This is done on line 5 and we denote the result with $[\![^f \{y\}]\!]$. Note that although we mentionaed before that we want to set τ_{e^x} to $\frac{2^{\{y\}}}{2}$, the result will be shifted left by $n-m$ bits after that. We can equivalently evaluate $2^{\{y\}}$ and shift the result to the left by $n-m-1$ bits instead of $n-m$ bits. Here we choose the second option and evaluate the polynomial to compute $2^{\{y\}}$ on line 7 by using $[\![^f \{y\}]\!]$ as input. The specific polynomials used will be specified in Section 7.2. We denote the result as $[\![^f v]\!]$.

Let us now consider what happens if $y < 0$. The problem is that the method for splitting y into an integer part and a fractional part does not work here as intended. Note that we did not use the sign of y while computing $[y]$ and $\{y\}$. Thus, essentially, if one performs those steps on a negative y, one obtains $[y]$ and

$\{y\}$ such that $[y]+\{y\}=|y|=-y$.

However, one can modify these results to obtain numbers $[y]'$ and $\{y\}'$ that satisfy $y=[y]'+\{y\}'$ and where $[y]'\in\mathbb{Z}$ and $\{y\}'\in(0,1]$. Namely, $y=-[y]-\{y\}=(-[y]-1)+(-\{y\}+1)$ and, thus, we can take $\{y\}':=-\{y\}+1\in(0,1]$ and $[y]':=-[y]-1\in\mathbb{Z}$.

Thus, we do not have to split y into an integer and fractional part twice, for the positive and negative case. However, what we must do twice, is polynomial evaluation, as $-\{y\}+1$ and $\{y\}$ are different numbers. Thus, on line 8, we evaluate the polynomial for computing $2^{1-\{y\}}$. We denote the result with $[\![^f v']\!]$. This, of course, can be performed in parallel with polynomial evaluation for computing $2^{\{y\}}$.

Likewise, correcting the $[\![^f v]\!]$ and $[\![^f v']\!]$ must also be done twice. Thus on lines 10 and 11, we call the Correction protocol on $[\![^f v]\!]$ and $[\![^f v']\!]$. Which flags b_0 and b_1 to use will be left decided by the implementation.

Finally, we have to obliviously choose between the negative and positive cases. Thus on line 13 we use s_x to choose between the two possible exponents for the result — $[\![[y]]\!]+1+q$ and $[\![-[y]]\!]+q$ and on line 14 we use s_x to choose between the two possible significands for the result — $[\![\tilde{v}]\!]\cdot 2^{n-m-1}$ and $[\![\widetilde{v'}]\!]\cdot 2^{n-m-1}$.

Consider what will happen if $x=0$. In that case, also $y=0$ and $\sigma_y=0$. Thus also $[y]=0$ and likewise $\{y\}=0$. Now we will get that $v'\approx 1$ and $v''\approx 2$, but still $v',v''\in[1,2)$ due to the Correction protocol. If $s_x=1$, then we will obtain the floating-point number that is approximately $([\![1]\!],[\![q+1]\!],[\![1\cdot 2^m\cdot 2^{n-m-1}]\!])$ which represents the number $2^1\cdot\frac{2^{n-1}}{2^n}\approx 1$. If $s_x=0$, then we will obtain the floating-point number that is approximately $([\![1]\!],[\![q]\!],[\![2\cdot 2^m\cdot 2^{n-m-1}]\!])$ which represents the number $2^0\cdot\frac{2^n}{2^n}\approx 1$. Thus the function works as intended for $x=0$.

We can summarize this subsection with the following claim.

Claim 6. *Algorithm 12, when given a secret floating-point number $[\![^F x]\!]$, outputs a secret floating-point number $[\![^F y]\!]$ where $y\approx e^x$.*

4.6. Error Function

This section discusses the algorithm for securely evaluating the Gaussian error function which is formalized in Algorithm 13. The Gaussian error function is defined by $\operatorname{erf}(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. It is an antisymmetric function, i.e, $\operatorname{erf}(-x)=-\operatorname{erf}(x)$. Thus, we can evaluate the function only depending on the exponent and the significand, and in the end, set the sign of the output to be the sign of the input. Thus, for the sake of simplicity, we will assume that our input is non-negative.

The Gaussian error function, however, is a more difficult function than the other three functions we have discussed. It does not partition naturally into a part where we manipulate exponents and a part where we evaluate a fixed-point polynomial.

Algorithm 12: Power of e of a floating-point number.

Data: Takes in a secret floating-point number $[\![^F x]\!] = ([\![s_x]\!], [\![E_x]\!], [\![\sigma_x]\!])$, the bias of the exponent q and the radix-point of the corresponding fixed-point number m, coefficients $\{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k$ for computing the fixed-point polynomial, public flags b_0 and b_1 that specify how the correction will be applied and the number of bits of the fixed-point number n.

Result: Outputs a secret floating-point number that is approximately equal to e^x.

$[\![^F y]\!] = ([\![s_y]\!], [\![E_y]\!], [\![\sigma_y]\!]) \leftarrow {}^F\log_2 e \cdot [\![^F x]\!]$
$[\![y]\!] \leftarrow \mathsf{PrivateBitShiftRightProtocol}([\![\sigma_y]\!], [\![n - (E_y - q)]\!])$
$[\![^F y]\!] \leftarrow \mathsf{ConvertToFloat}([\![y]\!])$
$[\![^F \{y\}]\!] \leftarrow [\![^F y]\!] - [\![^F y]\!]$
$[\![^f \{y\}]\!] \leftarrow \mathsf{PrivateBitShiftRightProtocol}([\![\sigma_{\{y\}}]\!], [\![-E_{\{y\}} + q + n - m]\!])$
begin in parallel
  $[\![^f v]\!] \leftarrow \mathsf{Poly}([\![^f \{y\}]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$
  $[\![^f v']\!] \leftarrow \mathsf{Poly}([\![^f (1 - \{y\})]\!], \{(-1)^{s_i} \cdot {}^f c_i\}_{i=0}^k, m, n)$
begin in parallel
  $[\![^f v]\!] \leftarrow \mathsf{Correction}([\![^f v]\!], 0, m, n, b_0, b_1)$
  $[\![^f v']\!] \leftarrow \mathsf{Correction}([\![^f v']\!], 0, m, n, b_0, b_1)$
begin in parallel
  $[\![E]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![s_x]\!], [\![y] + 1 + q]\!], [\![-[y] + q]\!])$
  $[\![\sigma]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![s_x]\!], [\![\widetilde{v}]\!] \cdot 2^{n-m-1}, [\![\widetilde{v'}]\!] \cdot 2^{n-m-1})$
$[\![^F e^x]\!] \leftarrow ([\![1]\!], [\![E]\!], [\![\sigma]\!])$
return $[\![^F e^x]\!]$

The good aspect of the error function is that it is difficult to evaluate only in a small range, as the function starts approximating 1 very soon, for example, when the input x is larger than 4, then $1-\operatorname{erf}(x) < 1.5 \cdot 10^{-8}$. Thus, if $x \geq 4$, we can approximate $\operatorname{erf}(x)$ with 1. Note that $\operatorname{erf}(x)$ approaches 1 superexponentially as x tends to infinity, as $1-\operatorname{erf}(x) = \Theta(e^{-x^2})$ [40]. On the other hand, when the input x is small, then $\operatorname{erf}(x)$ behaves practically linearly, namely, $\operatorname{erf}(x) \approx \frac{2}{\sqrt{\pi}}x$

This can be explained by observing the McLaurin series of the error function which is $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{i=0}^{\infty} \frac{(-1)^n}{n!\,(2n+1)} x^{2n+1}$. Note that

$$\begin{aligned} \left| \operatorname{erf}(x) - \frac{2}{\sqrt{\pi}}x \right| &= \left| \frac{2}{\sqrt{\pi}} \sum_{i=1}^{\infty} \frac{(-1)^n}{n!\,(2n+1)} x^{2n+1} \right| < \frac{2}{\sqrt{\pi}} \sum_{i=1}^{\infty} \frac{1}{n!\,(2n+1)} x^{2n+1} < \\ &< \frac{2}{\sqrt{\pi}} x \frac{1}{1!\,(2\cdot 1+1)} \sum_{i=1}^{\infty} x^{2n} = \frac{2}{3\cdot\sqrt{\pi}} \frac{x^3}{1-x^2}. \end{aligned}$$

If x is small enough, then $\frac{2}{3\cdot\sqrt{\pi}} \frac{x^3}{1-x^2}$ becomes sufficiently small to be ignored.

Hence, we need a method for evaluating $\operatorname{erf}(x)$ if $x \in [2^{-v}, 2^2)$ where v is suitably picked, depending on what speed and precision we wish to achieve. This is the range for which we need polynomial interpolation.

A single polynomial cannot be used for the whole range as it is too large. We shall, thus, use four specific polynomials p_0, p_1, p_2, p_3 so that $p_i(y) \approx \operatorname{erf} y$ in $[i, i+1)$ for every $i \in \{0,1,2,3\}$. The specific polynomials will be presented in Section 7.2.

We now need to cast the number into a suitable fixed-point number for the polynomial evaluation to work. As it was noted before, there are several possibilities for x — either $x < 2^{-v}$, $x \geq 4$ or $x \in [2^{-v}, 2^{-v+1}), x \in [2^{-v+1}, 2^{-v+2}), \ldots, x \in [2^1, 2^2)$. The information about which of the cases is true is contained solely in the exponent of x. Therefore we compute $\operatorname{erf}(x)$ for all the cases, later obliviously choosing between the cases. As it was noted before, the first two cases are easy. We shall now consider the other cases.

We want to cast our floating-point number into a fixed-point number for polynomial evaluation. If $x \in [2^j, 2^{j+1})$ and our fixed-point type has m bits after the radix point, then we have to shift the significand of x to the right by $n-m-j-1$ bits. Namely, given a fixed-point number ${}^f a$ with radix-point m, if $a \in [2^{j-1}, 2^j)$, then the most significant nonzero bit of $\widetilde{a}$ is the $(n-m+j+1)$-st bit. In the significand, the most significant nonzero bit is the first bit.

In the previous protocol, when we did not know in which interval $[2^{j-1}, 2^j)$ lies and we used PrivateBitShiftRightProtocol to obtain a fixed-point number on which to perform polynomial evaluation. However, in this case, the algoritm will be slightly different depending in which interval x lies. Thus we have to run

all these possibilites. As it was noted above, if x lies in $[2^{j-1}, 2^j)$, we have to shift σ_x to the right by $n-m-j$ bits to obtain the representative of ${}^f x$. Thus let us the compute fixed-point numbers $[\![{}^f x_j]\!] \xleftarrow{f} \mathsf{PublicBitShiftRightProtocol}([\![\sigma_x]\!], n-m-j)$ for $j \in \{-v+1, \ldots, 2\}$. This is done on line 1.

Now we can apply suitable polynomials to $[\![{}^f x_j]\!]$. For $j \in \{-v+1, \ldots, 0\}$ we will apply the polynomial p_0 as in all those cases, $x \in [0,1)$. This is done on lines 3 to 6 and we denote the result of applying p_0 to $[\![{}^f x_j]\!]$ with $[\![{}^f y_j]\!]$. If $j = 1$, we apply p_1, because in this case $x \in [1,2)$. This is done on line 7, and likewise, we denote the result with $[\![{}^f y_1]\!]$. In the case of $j = 2$, we know that $x \in [2,4)$ and, thus, we ought to apply either p_2 or p_3. To find which of these should be applied we examinine the second most significand bit of $\widetilde{x_2}$, however, there is one additional problem with this case.

Namely, note that if $x \in [2,4)$ and we want to use a polynomial with a degree t, then we have to be able to express numbers as large as $4^t = 2^{2t}$. Thus, when our fixed-point number has n bits, this leaves only $n-2t$ bits after the radix-point. This can result in low granularity and, thus, poor precision. Also note that we do not use fixed-point numbers with values as high as 4^t anywhere else in this functionality. Thus we would have to use such a radix-point only for the sake of being able to represent this one value.

Thus, instead of evaluating $\sum_{i=0}^{t} a_i [\![{}^f x_2]\!]^i$, we evaluate $\sum_{i=0}^{t} 2^i a_i [\![{}^f x_1]\!]^i$, because $x_2 \approx 2x_1$. In this case, the numbers $[\![{}^f x_1]\!]^i$ are all in $[0,1]$ and thus we can still obtain good granularity. Observe that in approximating polynomials, the coefficients a_i tend to diminish exponentially as i grows and thus generally $2^i a_i$ tends to be smaller than 1 and we do not lose granularity from this side either. On line 8 we set $[\![{}^f y_{2,0}]\!]$ to $\mathsf{Poly}([\![{}^f x_1]\!], m, n, \{(-1)^{s_{i,2}} \cdot {}^f(2^i c_{i,2})\}_{i=0}^{t_2})$ where $\{(-1)^{s_{i,2}} c_{i,2}\}$ are the coefficients of p_2 and on line 9 we set $[\![{}^f y_{2,1}]\!]$ to $\mathsf{Poly}([\![{}^f x_1]\!], m, n, \{(-1)^{s_{i,3}} \cdot {}^f(2^i c_{i,3})\}_{i=0}^{t_3})$ where $\{(-1)^{s_{i,3}} c_{i,3}\}$ are the coefficients of p_3.

As mentioned above, the choices between most of the cases shall be based on the exponent, but the choice between the cases where $x \in [2,3)$ or $x \in [3,4)$ must be made based on the significand. If $x \in [2,4)$ and the second most significant bit of the signifcand is 0, then $x \in [2,3)$, otherwise, $x \in [3,4)$. Thus we obliviously choose between $[\![{}^f y_{2,0}]\!]$ and $[\![{}^f y_{2,1}]\!]$ using the second bit of σ_x. Thus, on line 10 we first extract the bits of σ_x and denote them with $\{[\![u_i]\!]\}_{i=0}^{n-1}$, and then, on line 11, we use u_{n-2} to obliviously choose between $[\![{}^f y_{2,1}]\!]$ and $[\![{}^f y_{2,0}]\!]$. We denote the result of this by $[\![{}^f y_2]\!]$.

Now that we have obtained the correct fixed-point values for the different values of the exponent, we shall change them back to floating-point values. Note that for x larger than approximately 0.5, $\mathrm{erf}(x) \in (0.5, 1)$ and that for $x \in [0, 0.5)$, erf is rather linear, with $\frac{\mathrm{erf}(x)}{x} \in [1, 1.15)$ for $x \in [0, 0.5)$. This was experimentally verified. Thus, if $x \in [2^{j-1}, 2^j)$ for a $j < 1$, then $\mathrm{erf}(x) \in [2^{j-1}, 2^{j+1})$. If $x \in [2^{j-1}, 2^j)$

for a $j \geq 1$, then $\text{erf}(x) \in [0.5, 1)$.

However, due to approximation errors, it might happen that y_1 or y_2 are greater than 1. Thus we assume that if $j \geq 1$, then $y_j \in [2^{-1}, 2^1)$.

Now the prerequisites for the FixToFloatConversion are satisfied and we can apply it to the fixed-point numbers $[\![^f y_i]\!]$ to obtain the corresponding floating-point values $[\![^F y_i]\!]$. This is done on line 16, the respective parametres were set on the lines 12 to 14. On line 17 we additionally set $[\![^f y_{-v}]\!]$ to be $^F \frac{2}{\sqrt{\pi}} \cdot [\![^F x]\!]$. As noted, if x is small, then $\text{erf}(x)$ can be very well approximated with $\frac{2}{\sqrt{\pi}}[\![x]\!]$. Similarly, on line 18 we set $[\![^F y_3]\!]$ to $[\![^F 1]\!]$ for a similar reason. If $x \geq 4$, then $\text{erf}(x)$ is very close to 1.

Now, we simply must choose the correct result among the $[\![^F y_{-v}]\!], \ldots, [\![^F y_3]\!]$. We will do this by using the exponent of x and the GeneralizedObliviousChoice protocol to choose obliviously between the various $[\![^F y_i]\!]$ — intuitively, we want to return$[\![^F y_{e_x}]\!]$. However, at the moment this cannot yet be done because currently, e_x can have values outside of $\{-v, \ldots, 3\}$. We note, however, that if $e_x < -v$, then we want to return $[\![^F y_{-v}]\!]$, just as in the case when $e_x = [\![^F y_{-v}]\!]$. Thus, if $e_x \leq -v$, then we shall replace it with $-v$. Likewise, if $e_x \geq 3$, then we shall replace it with 3.

Replacing e_x with $-v$ if $e_x \leq -v$ is done in the following way: we compare E_x with $q - v$ using LTEProtocol and set the resulting bit to $[\![b_0]\!]$ on line 20. After that we use $[\![b_0]\!]$ on line 22 to obliviously choose between E_x and $q - v$ for the new value of E_x.

Replacing e_x with 3 if $e_x \geq 3$ on lines 21 and 23 is analogous and note that the two comparisons can be run in parallel.

Now it is finally possible to run the GeneralizedObliviousChoice on lines 24 and 25 and obtain the result.

Note that if $x = 0$, then $E_x < -v$ and the result will be set to $[\![^F y_{-v}]\!] = \frac{2}{\sqrt{\pi}} \cdot [\![^F 0]\!] = [\![^F 0]\!]$ which is the correct answer.

We can summarize this subsection with the following claim.

Claim 7. *Algorithm 13, when given a secret floating-point number $[\![^F x]\!]$, outputs a secret floating-point number $[\![^F y]\!]$ where $y \approx \text{erf} x$.*

4.6.1. Conclusion

In this section we presented a method for combining fixed-point and floating-point numbers in the context of evaluating certain functions securely. We implemented the protocols and benchmarked the results. The specific benchmarks can be found in Chapter 7. From the benchmarks we can see that the methods we propose in this section are much more efficient for larger vector sizes. Considering the amortized cost, our solution greatly outperforms the approaches that were built for computing only a single operation. For inverse and square root functions, it also

Algorithm 13: Gaussian error function of a floating-point number.

Data: $[\![^F x]\!], q, m, n, v, \{\{(-1)^{s_{i,j}} \cdot {}^f c_{i,j}\}_{i=0}^{t_j}\}_{j=0}^{3}$

Result: Takes in a a secret floating-point number $[\![^F x]\!]$, the bias of the exponent q and the radix-point of the corresponding fixed-point number m, coefficients $\{s_{i,j}{}^f c_{i,j}\}_{i=0}^{l}$ for computing the fixed-point values that are accurate in $[j, j+1)$ and an integer v so that we evaluate the function with a polynomial, if $2^{-v} \leq x < 4$. Outputs a secret floating-point number that is approximately equal to $\mathrm{erf}(x)$.

$\{[\![^f x_j]\!]\}_{j=-v+1}^{2} \xleftarrow{f} \mathsf{PublicBitShiftRightProtocol}([\![\sigma_x]\!], \{n-m-j\}_{j=-v+1}^{2})$

begin in parallel

  $[\![^f y_{-v+1}]\!] \leftarrow \mathsf{Poly}([\![^f x_{-v+1}]\!], m, n, \{(-1)^{s_{i,0}} \cdot {}^f c_{i,0}\}_{i=0}^{t_0})$

  $[\![^f y_{-v+2}]\!] \leftarrow \mathsf{Poly}([\![^f x_{-v+2}]\!], m, n, \{(-1)^{s_{i,0}} \cdot {}^f c_{i,0}\}_{i=0}^{t_0})$

  ...

  $[\![^f y_0]\!] \leftarrow \mathsf{Poly}([\![^f x_0]\!], m, n, \{(-1)^{s_{i,0}} \cdot {}^f c_{i,0}\}_{i=0}^{t_0})$

  $[\![^f y_1]\!] \leftarrow \mathsf{Poly}([\![^f x_1]\!], m, n, \{(-1)^{s_{i,1}} \cdot {}^f c_{i,1}\}_{i=0}^{t_1})$

  $[\![^f y_{2,0}]\!] \leftarrow \mathsf{Poly}([\![^f x_1]\!], m, n, \{(-1)^{s_{i,2}} \cdot {}^f (2^i c_{i,2})\}_{i=0}^{t_2})$

  $[\![^f y_{2,1}]\!] \leftarrow \mathsf{Poly}([\![^f x_1]\!], m, n, \{(-1)^{s_{i,3}} \cdot {}^f (2^i c_{i,3})\}_{i=0}^{t_3})$

$\{[\![u_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtraction}([\![\sigma_x]\!])$

$[\![^f y_2]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![u_{n-2}]\!], [\![^f y_{2,1}]\!], [\![^f y_{2,0}]\!])$

for $j \leftarrow -v+1$ **to** 0 **do**

  $t_j \leftarrow j$

$t_1, t_2 \leftarrow 0$

for $j \leftarrow -v+1$ **to** 2 **do** in parallel

  $[\![^F y_j]\!] \leftarrow \mathsf{FixToFloatConversion}([\![^f y_j]\!], t_j, m, n)$

$[\![^F y_{-v}]\!] \leftarrow {}^F \dfrac{2}{\sqrt{\pi}} \cdot [\![^F x]\!]$

$[\![^F y_3]\!] \leftarrow {}^F 1$

begin in parallel

  $[\![b_0]\!] \leftarrow \mathsf{LTEProtocol}([\![E_x]\!], [\![q-v]\!])$

  $[\![b_1]\!] \leftarrow \mathsf{LTEProtocol}([\![q+3]\!], [\![E_x]\!])$

$[\![E_x]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![b_0]\!], [\![q-v]\!], [\![E_x]\!])$

$[\![E_x]\!] \leftarrow \mathsf{ObliviousChoiceProtocol}([\![b_1]\!], [\![q+3]\!], [\![E_x]\!])$

$[\![E]\!] \leftarrow \mathsf{GenObliviousChoice}([\![E_{y_{-v}}]\!], \ldots, [\![E_{y_3}]\!], [\![E_x - q + v]\!])$

$[\![\sigma]\!] \leftarrow \mathsf{GenObliviousChoice}([\![\sigma_{y_{-v}}]\!], \ldots, [\![\sigma_{y_3}]\!], [\![E_x - q + v]\!])$

return $[\![^F \mathrm{erf}(x)]\!] = ([\![s_x]\!], [\![E]\!], [\![\sigma]\!])$

outperforms the results of Kamm and Willemson. As this technique was based on improving the results of that paper by replacing a floating-point number with a fixed-point number, this shows that for the inverse and square root the technique gives the expected benefit.

Considering the exponent and the Gaussian error function, our approach still often outperforms the results of Kamm and Willemson, however, the improvement is not as remarkable as in the case of the inverse and square root functions.

Indeed, for the 64-bit exponential function with batch size 10000 the amortized cost of the current result is even slightly outperformed by both the corresponding result by Kamm and Willemson and the amortized cost of the 64-bit exponential function with batch size 1000.

The reason behind why the inverse and square root functions gain more from the optimization presented in this chapter is that it is possible to utilize the fixed-point polynomial evaluation in a more contained, modular sense. For both the exponential and the Gaussian error function more corrections have to be made to ensure that the inputs and outputs belong to the correct intervals. However, even in those cases, there is generally a notable improvement in efficiency.

5. POINT-COUNTING

In this chapter we shall describe another technique that is suitable for improvements in secure computing for a certain class of functions. Although it can be used also for integer-typed computations, its main application is for real-value typed secret values. More specifically, it is suitable for fixed-point numbers. It presumes the existence of relatively few pre-existing primitives, and can theoretically be used for various secure computation settings. However, its main optimization gain is that it achieves lower round cost by performing more operations and, thus, is only reasonable in settings where parallel composition is highly preferrable to sequential composition. Therefore it is a good fit for the secure computation setting based on secret-sharing, as the architecture benefits from parallel composition very much [11, p. 74-75]. It must be noted that this requires a relatively sophisticated implementation that is able to make use of the resources. [1]

5.1. Introduction

Like in the previous chapter, we want to obtain methods that improve performance of functions on secure real-valued input. There are several desirable qualities that such methods should have — we would like them to improve precision and be more efficient in that they take less time, but there are also other desirable qualities. For example, two such qualities are applicability to a large number of functions and the ability to have precision as an input parameter not as a built-in constant of the technique.

In this chapter we will describe a method that can take precision as an input parameter and is applicable to a wider class of functions than the hybrid method. As can be seen in Chapter 7, in some cases it outperforms some other comparable methods with regards to precision and efficiency.

The method described in this chapter was inspired by the Monte Carlo methods — to evaluate a function $f(x)$, we run a test on a large amount of randomly picked points and count the number of points where the test passes. The test depends on both the function f and the value x. The number of points where the test passes can be used to more efficiently evaluate the function f at value x. In this chapter we are naturally interested in the case where we want to evaluate f at a secret

[1] Note that the term 'parallel composition' does not necessarily mean that this needs to be performed on parallel infrastucture. For example, in the secret-sharing paradigm, computation is performed by the parties sending messages to each other. "Parallel composition' can here mean that instead of sending several messages sequentially that contain the respective data about the variables, we send a larger message than contains the data about all the variables. Note that for this case, the capabilities of communication channels are an important factor in this case. If the network channel is unsaturated, then the running time grows sublinearly in the number of inputs. This can be considered ineffective [46, p. 20].

value $[\![x]\!]$. Unlike the real Monte Carlo methods, it turned out that it was more efficient to use equidistributed values instead of random values.

Also, the technique relies on the assumption that for a function f that we wish to compute there exists an 'inverse function' g that is easy to compute. We can thus run many instances of g on different inputs as tests and the proportion that passes gives us the answer.

We will be working on fixed-point numbers in this chapter. The fixed-point numbers used in this chapter can mean either unsigned or signed fixed-point numbers, depending on whether we can be sure that we will have negative values or not.

The reason why we use fixed-point numbers is that this method is built on the possibility of easily obtaining sets of equidistributed points. While this is also possible for floating-point numbers with the extra constraint that the points all belong to some $[2^t, 2^{t+1})$, this makes the method more complicated and thus we do not consider floating-point numbers. Whether this method can be generalized to other number types is a subject of further research.

The technique that we shall describe shall require relatively few operations and data types. Now let us consider what operations are necessary. There are three operations that are generally needed for the method:

- addition of private fixed-point numbers;
- multiplication of public fixed-point numbers with private integers;
- comparison of private fixed-point numbers. Since fixed-point numbers are stored and ordered as integers, this reduces to comparison of integers.

In addition to these, we also need a special operation g that depends on the function f that we want to compute. We can intuitively think of g as the inverse operation of f or verification of f. In order for the method to be fast and therefore applicable, g should be easy to compute. The function g should also be monotone and gf should be a simple function. In our examples, gf is either the identity function or the constant function that outputs 1. We formalize this by requiring that we must have access to functions g and h so that $g(x, f(x)) \equiv h(x)$ and that g is monotone in the first argument in the range where we know $f(x)$ to lie.

In our examples, g can be usually be computed by one or a few multiplication operations. Thus, g can in practice be reduced to the ability of multiplying fixed-point numbers.

5.2. The Scalar Pick Function

The main method this chapter relies on performing many operations in parallel, with the aim of reducing the round complexity of communication. However, we first give an example of another, fairly trivial method. The aim of this is to build intuition about the main method, as its construction is rather similar to it. We will also use this example later as a subprotocol in a more complicated protocol. For

this example we shall need secret integer comparison, multiplication of a secret fixed-point number with a public fixed-point number, and the addition of private fixed-point numbers. The process is specified in Algorithm 14.

Suppose that we want to compute some function f on a secret input $[\![^f x]\!]$ where $x \in [a,b)$ and where f is twice differentiable in $[a,b)$. We then choose a small Δ and, thus, obtain a large set of equidistributed points $\{a_i\}_{i=0}^{\ell-1} := \{a + i \cdot \Delta | i \in \mathbb{N}, a + i \cdot \Delta \in [a,b)\}$. Let j be such an integer that $x \in [a_j, a_{j+1})$. Because f is twice differentiable, we can expect its values not to fluctuate too much, thus the value of $f(x)$ should be close to $f(a_j)$. We would like to obtain the value of that index j.

For that we compute in parallel $[\![c_i]\!] \leftarrow {}^f a_i \stackrel{?}{\leq} [\![^f x]\!]$ for all the ${}^f a_i$ by using the secure comparison operator LTEProtocol. Note that the vector $\{c_i\}_{i=0}^{\ell-1}$ looks like $(1,1,\ldots,1,1,0,0,\ldots,0)$ with $c_0 = \cdots = c_j = 1$ and $c_{j+1} = \cdots = c_{\ell-1} = 0$.

We thus let $[\![d_i]\!] := [\![c_i]\!] - [\![c_{i+1}]\!]$ if $i \leq \ell - 1$ and $[\![d_{\ell-1}]\!] := [\![c_{\ell-1}]\!]$. Now we note that $d_j = 1$ and that $d_i = 0$ for all $i \neq j$. Essentially the vector $\{[\![d_i]\!]\}_{i=0}^{\ell}$ is the private characteristic vector of j.

It easily follows that $\sum_{i=0}^{\ell-1} f(a_i) \cdot d_i = f(a_j)$. As it was noted above, $f(a_j)$ is a good approximation for $f(x)$. Thus we can return $\sum_i {}^f f(a_i) \cdot [\![d_i]\!]$ as the answer.

Let us briefly consider the precision of this method. We assumed that f has first and second derivatives in $[a,b)$. Let $c_1 := \max_{y \in [a,b)} |f'(y)|$ and $c_2 := \max_{y \in [a,b)} |f''(y)|$. Then, according to Taylor's theorem, $|f(x) - f(a_j)| \leq c_1 \Delta + \frac{c_2 \Delta^2}{6}$. We can also add the error resulting from the inaccuracy of the fixed-point representation, but the error will be dominated by $c_1 h$.

This method may of course be used for functions that are not twice differentiable in the given interval but where the value of the function simply does not change much, however, then we need different error estimation methods.

5.3. The Point-Counting Method

Now we shall consider the main idea of this chapter.

While the intuition-building idea described in the previous section is very round-efficient, there are two main problems with it. First, that it needs a very large number of operations to be performed in parallel if we want to achieve reasonable precision. Second, it requires either a large amount of memory or a large amount of public computation, because all of the $f(a_i)$ must be either stored or recomputed. We will solve both of these problems with the cost of a more limited scope of application. We start by considering a solution that deals with the second problem but not the first.

This method can be applied to functions that have easily-computable "inverse

Algorithm 14: Scalar Pick

Data: Takes in a secret number $[\![^f x]\!]$ such that $x \in [a,b)$, and numbers $b_i \approx f(a+i\cdot\Delta)$ where f is the function we evaluate. Here Δ is a small constant. This function is called with $\mathsf{ScalarPick}([\![^f x]\!], a, b, \{^f b_i\}_{i=0}^{\ell-1})$

Result: Outputs $[\![z]\!]$ where z approximately equal to $f(x)$.

for $i = 0, i < \ell, i{+}{+}$ **do**
  $^f a_i \leftarrow {}^f a + i \cdot {}^f \Delta$
$\{[\![c_i]\!]\}_{i=0}^{\ell-1} \leftarrow \mathsf{LTEProtocol}(\{^f a_i\}_{i=0}^{\ell-1}, \{[\![^f x]\!]\}_{i=0}^{\ell-1})$
for $i = 0, i < \ell - 1, i{+}{+}$ **do**
  $[\![d_i]\!] \leftarrow [\![c_i]\!] - [\![c_{i+1}]\!]$
$[\![d_{\ell-1}]\!] \leftarrow [\![c_{\ell-1}]\!]$
$[\![z]\!] = \sum_{i=0}^{\ell-1} [\![d_i]\!] \cdot {}^f b_i$
return $[\![z]\!]$

functions". This definition is somewhat informal in order to be intuitively understandable. For example, consider the function $f(x) = \sqrt[k]{x}$ in some interval. This function could be computed using polynomial interpolation, which uses many rounds and is only accurate in a small interval. However, computing x^k takes only $\log k$ rounds of multiplications and its accuracy does not depend on the interval. However, for some settings, x^k might not fit into the respective fixed-point type for larger values of x and k and, thus, for those cases, one must be careful when designing protocols.

Let us now consider an alternative approach. Unlike in Section 5.2, here we do not consider a large set of points in the domain of the function, but a large set of points in the range of the function.

We start with the knowledge that $\sqrt[k]{x}$ belongs to some interval $[a,b)$ where the endpoints a and b might be either private or public. Note that this does not constrain us too much, as when we know nothing about the possible value of $\sqrt[k]{x}$, we can take a and b to respectively be the smallest and largest fixed-point numbers that we can represent.

Let now the interval $[a,b)$ be covered in equidistributed points $a+i\cdot\Delta$ where Δ is relatively small. Note now that if j is such that $a + j\cdot\Delta \leq \sqrt[k]{x} < a + (j+1)\cdot\Delta$, then $(a+j\cdot\Delta)^k \leq x < (a+(j+1)\cdot\Delta)^k$. Thus, by computing $(a+i\cdot\Delta)^k$ for all the i and performing a trick similar to the one performed in the previous section, we can obtain $[\![^f(a+j\cdot\Delta)]\!]$ where $a+j\cdot\Delta$ is an approximation to $\sqrt[k]{x}$. Note that here we do not need to store the $f(a_j)$.

Thus, we have a method that essentially computes $\sqrt[k]{x}$ by computing many instances of x^k in parallel. Hence here the "inverse function" is thus $g(x,y) = y^k$. However, when we apply the same method, for example, to $f(x) = \dfrac{1}{x}$, the 'inverse

function' is $g(x,y) = y \cdot x$.

We will now describe the method more formally, as it is presented in Algorithm 15. We want to compute a function $f(x)$ where the input x is secret, but about which we know that $f(x)$ belongs to the interval $[a, a+2^k)$ where a can be private or public. Additionally there should be easily computable functions g and h where $g(f(x),x) \equiv h(x)$ where $g(\cdot,x)$ is also monotonous in the interval $[a,a+2^k)$. For example, if $f(x) = \frac{1}{\sqrt{x}}$, then $g(x,y) = x^2 \cdot y$ and $h(x) = 1$.

We thus start with a value a about which we know that the output $f(x)$ will be in some $[a,a+2^k)$. Let the precision that we want to achieve be 2^{k-s}. We start by computing $h([\![^f x]\!])$ on line 1 and set it to $[\![^f w]\!]$. Then, on line 2, we establish the values $a_i \leftarrow a + i \cdot 2^{k-s}$ for every $i \in \{1,\ldots,2^s-1\}$. On line 3, we compute $g(a_i,x)$. We can now get an estimate for the value of $f([\![^f x]\!])$ with the proportion of $g(a_i,x)$ that are smaller than $[\![^f w]\!]$.

Thus, on lines 5 and 7 we perform secret comparisons by setting $c_i := g(a_i,x) \overset{?}{\leq} h(x)$ if g is increasing and $c_i := g(a_i,x) \overset{?}{\geq} h(x)$ if g is decreasing, respectively. Finally, on lines 8 and 9, we set the representative of the result to be $a + 2^{k-s+m} \cdot \left(\sum_{i=1}^{2^s-1} c_i \right)$.

Intuitively, this gives a correct answer because for one $j \in \{0,\ldots,2^s-1\}$, a_j is approximately equal to $f(x)$. If g is increasing, we can measure the position of this j in $\{0,\ldots,2^s-1\}$ by testing whether $g(a_i,x) \overset{?}{\leq} h(x)$ for all i. Due to monotonicity, for all i smaller than j, $g(a_i,x) \leq h(x)$ but for all i greater than j, $g(a_i,x) > h(x)$. Thus the number of i that 'pass the test' is proportional to the position of j in $\{0,\ldots,2^s-1\}$. A similar argument holds when g is decreasing.

The following theorem shows why the approach works.

Theorem 2. *Let f be a function. Let g, g_x and h be functions such that $g(f(x),x) \equiv h(x)$, $g_x(y) := g(y,x)$ and g_x is strictly monotonous in $[a,a+2^k)$ for all $x \in [a,a+2^k)$. Let $x \in X$ be such that $f(x) \in [a,a+2^k)$, $y_0, y_1, \ldots, y_{2^s}$ be such that $y_i := a + i \cdot 2^{k-s}$, and $Y := \{y_1,\ldots,y_{2^s-1}\}$. Additionally let $Z := g_x(Y)$, $j := |\{y_i \in Y | g_x(y_i) \leq h(x)\}|$ and $j' := |\{y_i \in Y | g_x(y_i) \geq h(x)\}|$. Then $f(x) \in [y_j, y_{j+1})$ if g is monotonously increasing and $f(x) \in [y_{j'}, y_{j'+1})$ if it is monotonously decreasing.*

Proof. We will give a proof for a monotonously increasing g_x without the loss of generality. First note that

$$g_x(y_1) < g_x(y_2) < \cdots < g_x(y_{2^s-1}),$$

because g_x is monotonously increasing. We know that $f(x) \in [y_r, y_{r+1})$ for some r. This is equivalent to $g_x(f(x)) \in [g_x(y_r), g_x(y_{r+1}))$, because g_x is monotone. Rewriting this gives us $h(x) \in [g_x(y_r), g_x(y_{r+1}))$. Because

$$g_x(y_1) < g_x(y_2) < \cdots < g_x(y_{2^s-1}),$$

this is equivalent to

$$h(x) \geq g_x(y_1), \ldots, g_x(y_r) \text{ and } h(x) < g_x(y_{r+1}), \ldots, g_x(y_{2^s-1}),$$

which in turn means that

$$|\{g(y_i) \in Z | h(x) \geq g(y_i)\}| = r.$$

Because the function g is one-to-one, we get that

$$|\{y_i \in Y | h(x) \geq g_x(y_i)\}| = r,$$

which gives us $r = j$ which is what we wanted to show. □

Algorithm 15: Computing a function with an easily computable inverse.

Data: Takes in $[\![^f x]\!], [\![^f a]\!], k, n, s$ and sign. We know that $f(x) \in [a, a+2^k)$. We assume that f is either monotonously increasing or decreasing in the interval and that we know which of the two options is true. sign is a public flag that is set to 1 if the function is increasing in the interval $[a, a+2^k)$, and 0 if it is decreasing in the interval $[a, a+2^k)$.

Result: Computes one round of function f of $[\![^f x]\!]$ where we already know that $f(x) \in [a, a+2^k)$. Here g and h are functions so that $g(f(x), x) \equiv h(x)$. The public flag sign is 1 if the function is increasing and 0 if it is decreasing in $[a, a+2^k)$. We use $2^s - 1$ test points and we work on n-bit data types.

$[\![^f w]\!] \leftarrow h([\![^f x]\!])$
$\{[\![^f a_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![^f a]\!] + i \cdot {}^f 2^{k-s}\}_{i=1}^{2^s-1}$
$\{[\![^f b_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{g([\![^f a_i]\!], [\![^f x]\!])\}_{i=1}^{2^s-1}$
if sign $== 1$ **then**
  $\{[\![c_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f b_i]\!]\}_{i=1}^{2^s-1}, \{[\![^f w]\!]\}_{i=1}^{2^s-1})$
else
  $\{[\![c_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f w]\!]\}_{i=1}^{2^s-1}, \{[\![^f b_i]\!]\}_{i=1}^{2^s-1})$
$[\![^f c]\!] \overset{f}{\leftarrow} \sum_{i=1}^{2^s-1} 2^{k-s+m} [\![c_i]\!]$
return $[\![^f a]\!] + [\![^f c]\!]$

The theorem also holds for slightly weaker assumptions which will be specified in the following note. We did not include these assumptions in the theorem for the sake of simplicity, however, it is easy to see that the theorem also holds for these weaker assumptions. Namely, one can replace the given assumptions with the weaker assumptions provided and follow through with the proof with only very minor modifications needed, the addition of which is trivial.

Note 2. We also note that it is not strictly necessary for g_x to be monotonous in $[a, a+2^k)$. It suffices for it to be monotonous in $[a+\alpha, a+2^k-\beta)$ provided that $x \in [a+\alpha, a+2^k-\beta)$ and that $\alpha, \beta < 2^{k-s}$.

We can summarize the results of this subsection with the following claim.

Claim 8. *Suppose that Algorithm 15 is given* $[\![^f x]\!], [\![^f a]\!]$*, k, s,* α*,* β*, functions f, g, and h so that* $f(x) \in [a+\alpha, a+2^k-\beta)$ *and* $g(f(x),x) \equiv h(x)$*, where* g_x *is monotonous in* $[a+\alpha, a+2^k-\beta)$ *and* $\alpha, \beta < 2^{k-s}$*. Then Algorithm 15 outputs* $[\![^f y]\!]$ *so that* $f(x) \in [a', a'+2^{k-s})$*.*

5.3.1. Iteration

While the method described in the previous section gives us a good round-efficiency, we still need a large amount of memory to store all of the $[\![c_i]\!]$ and $[\![d_i]\!]$. We also have to perform 2^s-2 secure evaluations function g and 2^s-2 secure evaluations of the comparison operator. If we want to obtain reasonable precision, then 2^s-2 must be rather large.

Thus, in a way, this can be thought of as just a version of the introductory idea, albeit one that takes slightly less memory and requires less precomputation. However, the method works only on a specific class of functions so it would seem, practically speaking, like a strictly worse version of the example-building idea, and, thus, not very useful. In this section, we describe how to improve on this method.

Besides the question of memory, we have also established that we operate in a setting where one of the most important parameters is round-efficiency and where we, thus, want to use parallel composition as much as possible.

However, in any specific setting, there is an upper limit to how much parallel processing power is available. This holds also for secret-sharing based settings, as there the parallel processing ability is limited by the network capabilities.

Thus, there is a practical saturation point that describes the maximal amount of parallel processing that we are able to do in a given setting. Past that point, even if the algorithm specifies that more operations should be performed in parallel composition, practically, some of them will have to be performed after others, i.e. in sequential composition [46, p.22].

Thus, in a specific setting, for every operation f there is, roughly speaking, some number ρ_f so that when we perform operations f more than ρ_f times, we start observing diminishing returns with increasing vector size. For example, increasing the vector size from k to αk for some $\alpha > 1$ improves the amortized speed by approximately a factor of α if $\alpha k < \rho_f$ but the improvement is strictly less than α if $\alpha k < \rho_f$. This is of course, a somewhat idealized model.

For the sake of simplicity, we let 2^p be an approximation for both $\rho_{\mathsf{LTEProtocol}}$ and ρ_g. In the following it shall signify the maximal power of two of either comparisons or operations g (whichever is larger) for which parallel composition gives an advantage. Thus we know that both 2^p secure comparison operations can be performed in parallel, as can 2^p operations g, but this does not hold for 2^{p+1}.

Thus, up to 2^{p-1}, doubling the number of tests should increase the overall computation time by a factor that is strictly smaller than 2. Past that point, however,

doubling the number of tests will double the performance time.

Thus, if we want to perform more than 2^p tests, the number of rounds will increase. If we want to perform $k2^p$ tests, then we essentially increase the round complexity k times. Thus, if we had a way of modifying our method by performing significantly fewer tests by performing slightly more rounds, it would be a gain in efficiency.

It happens to be that we have such a way for modifying our method. The idea is, in essence, 2^p-ary search.

We note that in the beginning of Algorithm 15 we start with values $[\![^f x]\!]$ and $[\![^f a]\!]$ where $f(x) \in [a, a+2^k)$ and running the protocol we obtain $[\![^f a']\!]$ so that $f(x) \in [a', a'+2^{k'})$ where k' is smaller than k. Thus, we can apply Algorithm 15 again, using a smaller h. This can be repeated until we obtain the desired precision.

More precisely, suppose that we want to compute v instances of some function f in parallel with accuracy of t bits and so that we beforehand know that $f(x_i) \in [a_i, a_i+2^k)$ for every $i \in \{1,\ldots,v\}$. Suppose also that our system can perform at most approximately 2^p comparison operations or operations g in parallel and we want to achieve precision 2^t.

Then we have to perform approximately $\dfrac{(k-t)\lceil \log v \rceil}{p}$ rounds of Algorithm 15 where in every round the total number of operations performed is smaller than or equal to 2^p.

The value $\dfrac{(k-t)\lceil \log v \rceil}{p}$ is derived in the following way. When $v = 1$, then it takes about $\dfrac{(k-t)}{p}$ rounds of the Algorithm 15 to reach from k to t—after the first round, the length of the interval goes from 2^k to 2^{k-p}, after the second round to 2^{k-2p}, and so on, until we reach 2^t.

However, if v is larger, then after the first round, the length of the interval goes from 2^k to only $2^{k-\lfloor p-\log v \rfloor}$, provided that $v \leq 2^p$. We will specify the case when $v > 2^p$ later. Now, using easy arithmetic, we obtain the value $\dfrac{(k-t)\lceil \log v \rceil}{p}$.

However, if the number of operations we wish to do in parallel is too large, then we must perform more than 2^p operations in one round. In this case, we shall compute only one extra bit each round because that is the smallest possible amount. The resulting procedure is presented as Algorithm 16, where the Round

subroutine refers to Algorithm 15.

Algorithm 16: Computing f using iteration

Data: $v, \{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![a_i]\!]\}_{i=0}^{v-1}, k, p, n, t$
Result: Computes function f of values $\{[\![x_i]\!]\}_{i=0}^{v-1}$. We know that $f(x_i) \in [a_i, a_i + 2^k)$ for all $i \in \{0, \ldots, v-1\}$. We can perform at most 2^p comparison or g operations in parallel. We work on n-bit data types and we wish to achieve precision 2^t

$s \leftarrow \max\{\lfloor p - \log v \rfloor, 1\}$
$r \leftarrow \lfloor \frac{k-t}{s} \rfloor$
$s' \leftarrow k - t - s \cdot r$
if $s' == 0$ **then**
    $s' \leftarrow s$
    $r \leftarrow r - 1$
$\{[\![y_{0,i}]\!]\}_{i=0}^{v-1} \leftarrow \mathsf{Round}(\{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![a_i]\!]\}_{i=0}^{v-1}, k, n, s')$
for $j = 1; j \leq r; j{+}{+}$ **do**
    $\{[\![y_{j,i}]\!]\}_{i=0}^{v-1} \leftarrow \mathsf{Round}(\{[\![x_i]\!]\}_{i=0}^{v-1}, \{[\![y_{j-1,i}]\!]\}_{i=0}^{v-1}, k - s' - (j-1)s, n, s)$
return $[\![y_r]\!]$

We can summarize the results of this subsection with the following claim.

Claim 9. *Suppose that Algorithm 15 is given $[\![^f x]\!], [\![^f a]\!]$, k, s, α, β t, functions f, g, and h so that $f(x) \in [a + \alpha, a + 2^k - \beta)$ and $g(f(x), x) \equiv h(x)$, where g_x is monotonous in $[a + \alpha, a + 2^k - \beta)$ and $\alpha, \beta < 2^{k-s}$. Then Algorithm 16 outputs $[\![^f y]\!]$ so that $f(x) \in [a', a' + 2^t)$.*

5.4. Applications of the Method

The class of functions f described in Theorem 2 is rather abstract and not easy to interpret. This section studies this class more closely.

The functions described by Theorem 2 are perhaps better understood when considering the possible options for the easily computable functions g, h and $\tilde{h}$. Which functions exactly are easy to compute depends on the underlying implementation of the secure computation engine. Typically such functions would include constants, additions, subtractions, multiplications and their compositions. However, depending on the system, other operations such as bit decompositions, shifts or other functions might be cheap and, thus, different functions might belong into that class. Note that in the case of the fixed-point multiplication detailed above in section 3.4.4, the operation calls converting up and down and shifting to the right as sub-operations and, thus, it might be reasonable to also consider these operations easily computable. However, for now, we shall not consider them for the sake of simplicity.

The compositions of constants, additions, subtractions, and multiplications are

polynomials. Thus, we are looking for functions $f(x)$ for which there exist polynomials p_1 and p_2 so that $p_1(f(x),x) = p_2(x)$ with the additional constraint that p_1 must be strictly monotonous in the first argument for any fixed value of the second argument.

Note that by defining the polynomial $p_3(y,x) := p_1(y,x) - p_2(x)$ we obtain that for a function f there must exist a polynomial p_3 so that $p_3(f(x),x) \equiv 0$ and that $p_3(\cdot,x)$ is strictly monotonous in the interval for any fixed x.

This means that we can think of f as a function that returns the root of p_3 in a given interval — the inverse function of that polynomial. Due to the monotonicity of p_3, an inverse function exists.

Thus, in this case, we can rephrase our class of functions as the set of functions that can be represented as problems for finding the roots of polynomials with secret coefficients in an interval where the named polynomials have precisely one root. Formally we can represent the problems in this class as

$$\begin{aligned} &\text{"given } \{[\![^f a_i]\!]\}\text{, find } x \ \in [a,b) \text{ so that } \sum_i [\![^f a_i]\!] x^i \ \approx 0, \\ &\text{provided that } \sum_i [\![^f a_i]\!] x^i \text{ has precisely one root in } [a,b)\text{".} \end{aligned} \tag{5.1}$$

For example:

- computing $\dfrac{1}{[\![^f a]\!]}$ is equivalent to finding a root of $[\![^f a]\!]x - 1 = 0$;
- computing $\sqrt{[\![^f a]\!]}$ is equivalent to finding a root of $x^2 - [\![^f a]\!] = 0$;
- computing $\dfrac{1}{\sqrt{[\![^f a]\!]}}$ is equivalent to finding a root of $[\![^f a]\!]x^2 - 1 = 0$;
- computing $\dfrac{[\![^f a]\!]}{[\![^f b]\!]}$ is equivalent to finding a root of $[\![^f b]\!]x - [\![^f a]\!] = 0$.

This class of problems can also be extended to finding the roots of polynomials with secret coefficients in general, whether they are injective in an area or not, and this is done in Section 5.4.1. Later, in Section 5.4.2, we will present the computation routine for the binary logarithm.

5.4.1. Finding Roots of Polynomials

We saw that finding the roots of injective polynomials with secret coefficients is a large subclass of problems that can be solved using the point-counting method described above.

We will now present a method for making point-counting applicable also for polynomials that are not injective in the given interval. Denoting the rank of the polynomial by k, the extended method will possibly be up to k^2 times slower, hence, it should be used only for polynomials with a suitably small rank.

The key observation for the extended method is the fact that we can still use the point-counting method if we can divide $[a,b)$ into intervals $[a,c_1),[c_1,c_2),..,[c_k,b)$

so that the polynomial $p(x)$ is monotone in all those intervals. We can then separately use the point-counting method in all those intervals.

The polynomial is monotone in an interval if the derivative of the polynomial does not change signs there. Since the derivative of a polynomial is a continuous function, it can change signs only at points where it is equal to zero. Thus we can find the points $c_1, \ldots, c_k$ by computing the roots of $p'(x)$. Now we again must find the roots of a polynomial, but that polynomial has a smaller rank than the original one. This, rather naturally, gives us a recursive algorithm. If $p'(x)$ is an injective function, we can directly use the point-counting algorithm. If it is not, we can compute its roots recursively.

Thus, if the rank of the polynomial $p(x)$ is k, then its $(k-1)$st derivative is a linear function and therefore injective, which means that the recursion has no more than $k-1$ steps.

We will now give the algorithm in more detail. We presume that we have access to the following three functions. First, we naturally assume that we have access to the function that evaluates the polynomial. We denote with

$$\mathsf{p}([\![^f a_0]\!], \ldots, [\![^f a_n]\!], [\![^f x_0]\!])$$

the function that evaluates the polynomial $\sum_{i=0}^{n} a_i x^i$ at x_0. This only requires multiplication and addition, which we already have access to.

Second, we assume access to the function $\mathsf{Der}([\![^f a_0]\!], \ldots, [\![^f a_n]\!])$ that takes in the coefficients of a polynomial and returns the coefficients of its derivative. This requires only multiplication of a secret fixed-point number with a public integer. We have access to this operation.

Third, we presume that we have access to a version of Algorithm 16 that, in an interval where a polynomial is injective, returns a root of the polynomial if it has one or an endpoint of the interval if it does not. The requirements for this function have been specified earlier in this chapter.

However, we need to modify the function Round that Algorithm 16 calls. We

shall replace Algorithm 15 with Algorithm 17 which differs from it in two ways.

Algorithm 17: Computing a function with an easily computable inverse in a secret interval. Function may be either increasing or decreasing.

Data: Gets in the values $[\![^f x]\!], [\![^f a]\!], [\![^f b]\!], k, n, s$. Here $[\![^f x]\!]$ is the input of the function. $[a,b)$ is the interval in which we know the answer to be. We know that $b-a < 2^k$. The value n refers to the number of bits in the underlying integer type of the fixed-point numbers and the number of test points we are using is $2^s - 1$.

Result: Computes one round of function f of $[\![^f x]\!]$ where we already know that $f(x) \in [[\![^f a]\!], [\![^f b]\!])$. Here g and h are functions so that $g(f(x),x) \equiv h(x)$. We know that the function is monotone but not whether it is increasing or decreasing.

$[\![^f w]\!] \leftarrow h([\![^f x]\!])$

$\{[\![^f a_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![^f a]\!] + i \cdot {}^f(2^{k-s})\}_{i=1}^{2^s-1}$

$\{[\![^f b_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![^f g(a_i,x)]\!]\}_{i=1}^{2^s-1}$

$\{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f b_i]\!]\}_{i=1}^{2^s-1}, \{[\![^f w]\!]\}_{i=1}^{2^s-1})$

$\{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f w]\!]\}_{i=1}^{2^s-1}, \{[\![^f b_i]\!]\}_{i=1}^{2^s-1})$

$\{[\![c'_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f b_i]\!]\}_{i=1}^{2^s-1}, \{[\![^f b]\!]\}_{i=1}^{2^s-1})$

$\{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![c_{i,0}]\!]\}_{i=1}^{2^s-1} \cdot \{[\![c'_i]\!]\}_{i=1}^{2^s-1}$

$[\![^f c_0]\!] = {}^f(2^{k-s}) \cdot \sum_{i=1}^{2^s-1} [\![c_{i,0}]\!]$

$\{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![c_{i,1}]\!]\}_{i=1}^{2^s-1} \cdot \{[\![c'_i]\!]\}_{i=1}^{2^s-1}$

$[\![^f c_1]\!] = {}^f(2^{k-s}) \cdot \sum_{i=1}^{2^s-1} [\![c_{i,1}]\!]$

$[\![^f z_a]\!] \leftarrow \mathsf{p}([\![^f a_0]\!], \ldots, [\![^f a_n]\!], [\![^f a]\!])$

$[\![^f z_b]\!] \leftarrow \mathsf{p}([\![^f a_0]\!], \ldots, [\![^f a_n]\!], [\![^f b]\!])$

$[\![z]\!] \leftarrow \mathsf{LTEProtocol}([\![^f z_a]\!], [\![^f z_b]\!])$

$[\![^f c]\!] \leftarrow [\![z]\!] \cdot [\![^f c_0]\!] + (1 - [\![z]\!]) \cdot [\![^f c_1]\!]$

return $[\![^f a]\!] + [\![^f c]\!]$

First, unlike in Algorithm 15, we do not know the length of the interval where our result may be. It might even happen that the interval has length zero. We solve this problem in the following way.

Suppose that we have $[\![^f a]\!]$ and $[\![^f b]\!]$ so that our solution is in $[a,b)$. It might happen that when we let $c_i := g(a + i \cdot h, x) \overset{?}{\leq} h(x)$ for all $i \in [0, 2^k]$ that the value $a + 2^{k-s+m} \cdot (\sum_{i=1}^{2^s-1} c_i)$ would fall out of $[a,b)$. This happens if for instance f is not monotonous in $[a, a+h2^k]$ and thus we can not use the Theorem 2.

We would still like to get a result that is in $[a,b)$ and that the test points that fall out of $[a,b)$ would not affect the outcome. We do this by essentially setting

the value c_i to 0 if the respective a_i does not belong to $[a,b)$.

We therefore do the following: for an increasing function (the algorithm is analogous for a decreasing function) we compute the values $[\![^f b_i]\!]$ by applying g_x to $[\![^f a_i]\!]$ on line 3 as usual in the interval $[a,a+2^k)$ where 2^k is such a number such that $b \leq a+2^k$. We also set $\{[\![c_i]\!]\}$ to $\mathsf{LTEProtocol}([\![^f b_i]\!],[\![^f h(x)]\!])$ on line 4.

However, now we also securely compare every point $[\![^f a_i]\!]$ to $[\![^f b]\!]$ on line 6 obtaining the comparison vector $\{[\![c_i']\!]\}$. Here c_i' is zero if and only if $a_i > b$, which means that it is out of $[a,b)$.

We now compute $[\![c_i]\!] = [\![c_i]\!] \cdot [\![c_i']\!]$ for every i on line 7. We then proceed as usual. After this procedure we can be certain that the result is in $[[\![^f a]\!],[\![^f b]\!])$.

Additionally Algorithm 17 differs from Algorithm 15 as we do not know whether the polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ is increasing or decreasing in the interval $[[\![a]\!],[\![b]\!])$ where it is injective. Hence we execute the algorithm for both cases and then compute $p(a) \overset{?}{\leq} p(b)$ to perform oblivious choice between the two options on lines 11, 12, and 13.

Because p is injective in the interval, the only case when it can happen that $p([\![a]\!]) = p([\![b]\!])$ is when $[\![a]\!] = [\![b]\!]$, but then the output of the function is always $[\![a]\!]$ and does not depend on whether we use the increasing or decreasing version of the algorithm. In other cases comparing $p([\![a]\!])$ and $p([\![b]\!])$ will determine whether the function is increasing or decreasing in that interval and, thus, the correct output. Thus we obtain the correct output by performing an oblivious choice on line 14. Thus we obtain Algorithm 17.

If we replace the call to Algorithm 15 with a call to Algorithm 17 in Algorithm 16, we shall obtain the function $\mathsf{injecRoot}([\![^f a_0]\!],\ldots,[\![^f a_n]\!],[\![^f a]\!],[\![^f b]\!])$ that takes in the secret coefficients $[\![^f a_0]\!],\ldots,[\![^f a_n]\!]$ of a polynomial and the secret interval $[[\![^f a]\!],[\![^f b]\!])$ such that the polynomial has at most one root in $[[\![^f a]\!],[\![^f b]\!])$. The function outputs the root of the polynomial in $[a,b)$, if it exists. If it does not exist, the function outputs the point $[\![^f a]\!]$ if the function has only positive values and is increasing in the interval or has only negative values and is decreasing in the interval. In other cases, it outputs the fixed-point number with the smallest representable absolute value in $[[\![a]\!],[\![b]\!])$.

We shall now present Algorithm 18 that returns n values, in increasing order, among which are all the real roots of the polynomial. We call this function with $\mathsf{polyRoot}([\![^f a_0]\!],\ldots,[\![^f a_n]\!],[\![^f a]\!],[\![^f b]\!],t)$. By saying that a root is at point c we mean that the root is in the interval $[c,c+2^{-m})$.

It is important to note that not all the returned values might be the roots of the polynomial — this is because the polynomial might not have n real roots as some of them might be complex. Generally, the algorithm returns the value with the smallest absolute value. If there is no root in the given interval, it returns the endpoint with a smaller absolute value.

We note that the behaviour of these "false roots" in the recursive algorithm that

we will present is interesting, but out of the scope of this thesis.

First the algorithm finds the polynomial that is the derivative of the polynomial $\sum_{i=0}^{n} [\![a_i]\!] x^i$. If that is a linear function, it applies the function injecRoot to it to obtain its root if it has one. If the derivative has a higher order, it recursively calls polyRoot to obtain $n-1$ values $c_1, \ldots, c_{n-1}$, in increasing order, among which are all the real roots of the derivative.

We then set $[\![c_0]\!] = [\![a]\!]$ and $[\![c_n]\!] = [\![b]\!]$. We then apply the function injecRoot to the original polynomial in the intervals $[[\![c_i]\!] + 2^t, [\![c_{i+1}]\!]]$ where 2^t is the precision of the function injecRoot. We return the outputs of injecRoot, ordered.

Algorithm 18: Computing roots of a polynomial

Data: Gets as input the values $[\![^f a_0]\!], \ldots, [\![^f a_n]\!], [\![^f a]\!], [\![^f b]\!], t$. Here the $[\![^f a_0]\!], \ldots, [\![^f a_n]\!]$ are the polynomial coefficients and all the points that interest us are in (a,b). The value m is the radix-point of the fixed-point number.

Result: Returns n values, in an increasing order, among which are all the real roots of the polynomial. Has precision 2^{-m}.

if $n > 1$ **then**
  $[\![^f b_0]\!], \ldots, [\![^f b_{n-1}]\!] \leftarrow \mathsf{Der}([\![^f a_0]\!], \ldots, [\![^f a_n]\!])$
  $[\![^f c_1]\!], \ldots, [\![^f c_{n-1}]\!] \leftarrow \mathsf{polyRoot}([\![^f b_0]\!], \ldots, [\![^f b_{n-1}]\!], [\![^f a]\!], [\![^f b]\!], t)$
  $[\![^f c_0]\!] \leftarrow [\![^f a]\!]$
  $[\![^f c_n]\!] \leftarrow [\![^f a]\!]$
  for $i = 0, i < n, i++$ **do**
    $[\![^f d_i]\!] \leftarrow \mathsf{injecRoot}([\![^f a_0]\!], \ldots, [\![^f a_n]\!], [\![^f (c_i + 2^{-m})]\!], [\![^f c_{i+1}]\!])$
  return $[\![^f d_0]\!], \ldots, [\![^f d_{n-1}]\!]$
else
  return $\mathsf{injecRoot}([\![^f a_0]\!], \ldots, [\![^f a_1]\!], [\![^f a]\!], [\![^f b]\!])$

Based on Theorem 2, we note that each step gives correct answers provided that the function under question is injective in the intervals where it is called. Based on properties of the derivative of a polynomial we conclude the correctness of the algorithm.

The reason why we chose the specific intervals for injecRoot as $[[\![c_i]\!] + 2^t, [\![c_{i+1}]\!]]$ is the following. We know that when we call the function on the derivative, all the zeroes are in the intervals $[c_i, c_i + 2^{-m})$. Thus, when taking an interval $[c_i + 2^{-m}, c_{i+1})$, we know that the derivative has no zeroes there and thus the function is one-to-one, hence we can apply the method.

5.4.2. Logarithm

In this Section we show how the point-counting method can be applied to computing binary logarithms.

As it was noted by Aliasgari *et al.* [5], an approximation of the exponential

function can be computed using the bits of the input to obliviously choose between 2^{2^i} and 1 and then computing the product over all bits — we can use this for the function g. At first, it may seem that this requires us to perform bit-decomposition and many multiplications for computing the function g. However, we will later see that it can be done in a manner where we only need a multiplication of a private and a public value to compute g.

Let us have input $[\![^f x]\!]$ and suppose that we want to compute $[\![^f \log x]\!]$.

Let us have n-bit three-field signed fixed-point numbers with radix-point m as input.

We assume that we have $[\![^f a]\!]$ and u such that $\log x \in [a, a+2^u)$. We also assume that we have the value $[\![^f 2^a]\!]$. We will later show how this $[\![^f 2^a]\!]$ can be obtained.

Our test points will be thus $a_i = a + i\Delta$. We let $\Delta = 2^{u'}$ where $u' < u$ and will be our new precision level.

Here $f(x) = \log x$. We will thus set $g(y)$ to 2^y and $h(x)$ to x. Now we are interested in how to compute 2^{a_i}.

$$2^{a_i} = 2^{a+i2^{u'}} = 2^a \cdot 2^{i2^{u'}}.$$

Note that the values $2^{i2^{u'}}$ depend only on public values, and thus these fixed-point numbers can be computed publicly. Thus, to compute $g_x(a_i) = 2^{a_i}$ we need to multiply $[\![^f 2^{2^a}]\!]$ with the public fixed-point number ${}^f 2^{i2^{u'}}$.

Let us now consider the requirement that we must possess the private value $[\![^f 2^a]\!]$. This, in fact, does not really restrict us much, besides the fact that we need perform somewhat more operations.

Consider the first round. We note that for fixed-point numbers, there is a well-defined minimal positive number — namely, 2^{-m}. Thus we can say that the logarithm of the fixed-point number will be in $[-m, n-m)$. Thus we can set $[\![^f a]\!]$ to ${}^f -m$ and $[\![^f 2^{2^a}]\!]$ to ${}^f(2^{-m})$.

However, if it is not the first round, then we can compute it based on the values we obtained from the previous round using the method described in Section 5.2.

We see that after a round of the algorithm, we obtain some $[\![^f a + ju']\!]$ so that $\log x \in [a+ju', a+(j+1)u')$. We are interested in obtaining the value $[\![^f 2^{a+ju'}]\!]$.

It is clear that $2^{a+ju'}$ is equal to $2^a \cdot 2^{ju'}$. While we possess the value $[\![^f 2^a]\!]$, we do not possess the value $[\![^f 2^{ju'}]\!]$, in fact, the value j is a secret. However, we can obtain j in a different manner.

Note that when we performed the first round of Algorithm 15, we computed the values $[\![c_i]\!] = [\![^f g(a_i)]\!] \stackrel{?}{\leq} [\![^f x]\!] = [\![^f 2^{a_i}]\!] \stackrel{?}{\leq} [\![^f x]\!]$.

As described in the section 5.2, we then compute $[\![d_0]\!] := 1 - [\![c_0]\!]$ and $[\![d_i]\!] := [\![c_{i-1}]\!] - [\![c_i]\!]$ for all $i \in \{1, \ldots, 2^s - 1\}$.

We note that only $d_j = 1$ — for all the other $i \neq j$, $d_i = 0$. This gives us a method for computing $[\![^f 2^{ju'}]\!]$ — namely, we compute $\sum_{i=0}^{2^{s-1}} [\![d_i]\!]^f 2^{iu'}$. Note that

these multiplications can be performed in parallel.

We thus arrive to Algorithm 19. It receives as input the shared fixed-point value $[\![^f x]\!]$ the logarithm of which we compute and a number of parametres, which we have discussed above, and additionally the parametre r that describes how many iterations of the protocol we perform. On lines 1 and 2 we set u to $\lceil \log n \rceil$ and u' to $u-s$, respectively. This is because we know the logarithm is in $[-m,-m+n)$ and after performing an iteration we wish to know the value of $\log x$ with precision $2^{u'}$ — i.e we want to improve the secure knowledge by s bits. As it was noted above, we can initialize $[\![^f a]\!]$ as $^f(-m)$ on line 4 and $[\![^f 2^a]\!]$ as $^f(2^{-m})$ on line 3.

Then on lines 6 to 17 we run the body of the algorithm r times. Every time we learn secure s bits of $\log x$.

The iterated argument is fairly standard, however, in addition to updating $[\![^f a]\!]$ we also update the approximate version of x— the $[\![^f 2^a]\!]$ on line 15.

Algorithm 19: Computing logarithm of a fixed-point number

Data: We get the inputs $[\![^f x]\!], s, m, n, r$. This algorithm securely computes the logarithm of x. We presume $x > 0$. Every round we test 2^s points. The radix-point of the fixed-point number is m and the underlying integer has n bits. We run this operation for r rounds.

Result: Returns a value $[\![^f a]\!]$ so that $\log x \in [a, a + 2^{\lceil \log n \rceil - sr})$.

1 $u \leftarrow \lceil \log n \rceil$
2 $u' \leftarrow u - s$
3 $[\![^f 2^a]\!] \leftarrow {}^f(2^{-m})$
4 $[\![^f a]\!] \leftarrow {}^f(-m)$
5 **for** $j = 0, j < r, j{+}{+}$ **do**
6 $\quad \{[\![^f a_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![^f a]\!] + i \cdot {}^f 2^{u'}\}_{i=1}^{2^s-1}$
7 $\quad \{[\![^f b_i]\!]\}_{i=1}^{2^s-1} \leftarrow \{[\![^f 2^a]\!] \cdot {}^f 2^{i2^{u'}}\}_{i=1}^{2^s-1}$
8 $\quad \{[\![c_i]\!]\}_{i=1}^{2^s-1} \leftarrow \mathsf{LTEProtocol}(\{[\![^f b_i]\!]\}_{i=1}^{2^s-1}, \{[\![^f x]\!]\}_{i=1}^{2^s-1})$
9 $\quad [\![^f c]\!] \xleftarrow{f} \sum_{i=1}^{2^s-1} {}^f 2^{u'+m} [\![c_i]\!]$
10 $\quad$ **for** $i = 0, i < 2^s - 1, i{+}{+}$ **do**
11 $\quad\quad [\![d_i]\!] \leftarrow [\![c_i]\!] - [\![c_{i+1}]\!]$
12 $\quad [\![d_{2^s-1}]\!] \leftarrow [\![c_{2^s-1}]\!]$
13 $\quad [\![^f a]\!] \leftarrow [\![^f a]\!] + [\![^f c]\!]$
14 $\quad$ **if** $j < r - 1$ **then**
15 $\quad\quad [\![^f 2^a]\!] \leftarrow [\![^f 2^a]\!] \cdot (\sum_{i=0}^{2^{s-1}} [\![d_i]\!]^f 2^{iu'})$
16 $\quad\quad u \leftarrow u'$
17 $\quad\quad u' \leftarrow u - s$
18 **return** $[\![^f a]\!]$

We can summarize the results of this section with the following claim.

Claim 10. *Algorithm 19, when given inputs* $[\![^f x]\!], n, s, r$ *where* $x > 0$*, outputs a fixed-point number* $[\![^f a]\!]$ *so that* $\log x \in [a, a + 2^{\lceil \log n \rceil - sr})$.

5.5. Conclusion and results

We implemented and benchmarked the iterated point-counting method for the inverse function, the square root function and the logarithm. One benefit of such approach is that we can very easily fine-tune the desired precision. The specific results and analysis can be found in Chapter 7.

6. GOLDEN SECTION NUMBERS

Previously we saw that both fixed-point numbers and floating-point numbers can be used in the secure setting. However, we saw that in this case, the costs of different operations vary from the usual case. Thus it is sensible to try techniques that might not be practical in the usual, non-secure setting. We can also apply this idea to real number types — perhaps there are use cases where neither fixed-point numbers nor floating-point numbers are the optimal real number type. Thus it makes sense to consider other real number types for secure computation.

In this chapter we will describe a real number type that can depict signed real numbers, has local addition, and other operations that are comparable to fixed-point numbers in efficiency.

We denote by φ the golden ratio $\frac{\sqrt{5}+1}{2}$. We define a golden section number to be a triple (a,b,x) where $a,b \in \mathbb{Z}$, $x \in \mathbb{R}$, and $a - \varphi x \approx x$. The idea is storing a and b in computer memory in order to store the value x. We also define a secret golden section number to be a triple $([\![a]\!],[\![b]\!],x)$, where $a,b \in \mathbb{Z}$, $x \in \mathbb{R}$, and $a - \varphi x \approx x$. Here, likewise, the secure integers $[\![a]\!]$ and $[\![b]\!]$ are stored with the aim to store x securely. For the sake of brevity, we will denote a golden number (a,b,x) with ${}^{g}x$ and a secret golden section number $([\![a]\!],[\![b]\!],x)$ with $[\![{}^{g}x]\!]$. We also say that the pair (a,b) represents x. Which one it will refer to will be clear from the context. We also use the term *golden numbers* for golden section numbers for the sake of brevity.

We refer to the first two elements of the triple as the *representatives* of the golden number, more specifically, we refer to the first element as the integer representative and to the second element as the φ-representative of the number. The value x will also be called the *value* of the golden number. Instead of referring to the absolute value of the value of a golden number, we will simply refer to it as the absolute value of that golden number.

Note that it would also be possible to define the golden section numbers in an alternative way where the number represented by the tuple (a,b) would be $a+\varphi b$. This is largely a matter of taste.

There are some properties that a system that represents real numbers should have. First, we want to be able to represent numbers with a sufficiently high granularity, and to perform some elementary arithmetic operations using the system. For a system of secure real numbers, we require that computing those arithmetic operations would not be too expensive or inaccurate in a secure setting. In this chapter we shall show that golden section numbers achieve these properties.

We begin with demonstrating that addition and multiplication, provided the existence of addition and multiplication of integers, is relatively straightforward. Suppose that we wish to add two golden numbers (a,b,x) and (c,d,y). We note the property

$$a - \varphi b + c - \varphi d = (a+c) - \varphi(b+d).$$

Hence if $a-\varphi b \approx x$ and $c-\varphi d \approx y$, then $(a+c)-\varphi(b+d) \approx x+y$ and thus the sum of (a,b,x) and (c,d,y) can be considered to be $(a+c,b+d,x+y)$.

For multiplication, likewise, suppose that we wish to multiply two golden numbers (a,b,x) and (c,d,y). We note that $\varphi^2 = \varphi + 1$ and thus obtain

$$(a-\varphi b)\cdot(c-\varphi d) = (ac+bd)-\varphi(bc+ad-bd). \tag{6.1}$$

Thus if $a-\varphi b \approx x$ and $c-\varphi d \approx y$, then $(ac+bd)-\varphi(bc+ad-bd) \approx x\cdot y$.

We now continue with the property of granularity. However, to discuss that, we first need a language to describe how well some pair (a,b) represents the value x.

Definition 2. Given a real number x, we say that the tuple of integers (a,b) is a (k,ε)-approximation of x if $|a|,|b| \leq k$ and $|a-\varphi b-x| \leq \varepsilon$. If k is implied by the context or not important in the context, we shall refer to (a,b) as just an ε-representation of x. If ε is implied or not important, we shall refer to (a,b) as a $(k,\cdot)$-representation of x.

If neither are important (or are implied) we refer to (a,b) as just as a representation of x.

Now, let us note that it is preferable to use such a (k,ε)-approximation of a number where both k and ε are relatively small. While it is clear that a small ε implies a small error and is therefore better, the reason why a small k is good requires some explanation.

Namely, we observe that when we multiply two golden section numbers x and y with a $(k,\cdot)$-approximation, then their product is with high probability a $(2k+1,\cdot)$-approximation of xy. Thus, we quickly run into the risk of overflow in the underlying data type. This could possibly happen even after one multiplication when k is too large. We will solve this problem by replacing a (k,ε)-representation of a number with a $(k',\varepsilon+\varepsilon')$-representation where $k' << k$ and ε' is suitably small. We will give more details about this method later.

Throughout the section we shall assume that the error ε is small, several orders of magnitude smaller than 1 because otherwise, the data type would be so inaccurate that it would be unusable. Thus, when we discuss how either the number or the representatives need to be bounded by some quite large numbers, we shall ignore ε in those analyses as the rounding down used in finding those large numbers will cover any overflow ε might cause.

Now let us consider how to characterize the distribution of golden section numbers. We want them to be distributed as uniformly as possible. There are two properties concerning distribution that we are interested in.

First, is the property of granularity — given a k, what is the largest distance between two consecutive golden section numbers that have $(k,\cdot)$-representations. More formally, what is

$$\max\{y-x \mid x=a_0-\varphi b_0, y=a_1-\varphi b_1, x<y, |a_0|,|a_1|,|b_0|,|b_1| \leq k, \\ \nexists z : z=a_2-\varphi b_2, x<z<y, |a_2|,|b_2| \leq k\}.$$

Granularity is a property that describes the maximal error we make if we want to give a $(k,\cdot)$-representation of a number. It is very important for describing a real number system.

The second property that we are interested in is equidistributedness. A set of elements where all elements fall to some interval $[a,b]$ is said to be equidistributed if the probability of falling to some subinterval of $[a,b]$ is proportional to the length of that subinterval in relation to $[a,b]$. The Weyl equidistribution theorem [17] states that for $\alpha \in \mathbb{R}$, the sequence $\{i \cdot \alpha \pmod 1\}_{i=0}^{\infty}$ is equidistributed in $[0,1)$ if and only if α is irrational. This gives us a sense of why we can expect golden numbers to be fairly evenly distributed.

This result suggests that the obtained set would be rather well equidistributed on $[0,1)$, as $\{i \cdot \varphi \pmod 1\}_{i=0}^{k}$ would with a high probability be relatively evenly distributed in $[0,1)$, given a large enough k. Thus, given some $\beta \in [0,1)$, we could find some $b_\beta \in \mathbb{Z}$ so that $b_\beta \cdot \varphi \pmod 1 \approx \beta$. Then the golden number representing β would be $b_\beta \cdot \varphi - \lfloor b_\beta \cdot \varphi \rfloor$. Now, to represent a number $\beta + t$ where $\beta \in [0,1)$ and $t \in \mathbb{Z}$, we obtain the representation (a_β, b_β) for β in the manner we just discussed and then use $(a_\beta + t, b_\beta)$ for $\beta + t$.

This argument however, is not very good. We now will give an argument that is better for several reasons. First, we will give a proof. Second, we will describe how good a $(k,\cdot)$-representation we will obtain. Finally, it will be constructive.

Lemma 2 gives us a more specific construction for obtaining approximations for specific numbers. However, to prove that, we first need Lemma 1. Recall that $\mathscr{F}_j$ denotes the j-th Fibonacci number.

Lemma 1. *Let $j > 1$ and let $\{a_i\}_{i=0}^{j}$ be bits such that $a_i = a_{i+1} = 1$ holds for no i. Then $\sum_{i=0}^{j} a_i \mathscr{F}_i < \mathscr{F}_{j+1}$.*

Proof. We prove by induction over j. First, if $j = 2$, then the claim obviously holds.

Now, let us assume that $\sum_{i=0}^{j} a_i \mathscr{F}_i < \mathscr{F}_{j+1}$ for all $j \leq k$ and show that it holds for $\sum_{i=0}^{k+1} a_i \mathscr{F}_i < \mathscr{F}_{k+2}$.

We note that there are two possibilities, either $a_{k+1} = 0$ or $a_k = 0$, otherwise the assumption would not hold.

Let us consider the case when $a_{k+1} = 0$. In that case $\sum_{i=0}^{k+1} a_i \mathscr{F}_i = \sum_{i=0}^{k} a_i \mathscr{F}_i < \mathscr{F}_{k+1} < \mathscr{F}_{k+2}$.

Now let us consider the case when $a_k = 0$. Then $\sum_{i=0}^{k+1} a_i \mathscr{F}_i = \sum_{i=0}^{k-1} a_i \mathscr{F}_i + a_{k+1} \mathscr{F}_{k+1} < \mathscr{F}_k + \mathscr{F}_{k+1} = \mathscr{F}_{k+2}$.

Thus the claim is proven. □

Lemma 2. *For a real number x which satisfies $|x| < \varphi^{s+1}$, and a positive integer k, there exists a $\left(\frac{\varphi^{s+1}+\varphi^{k+2}}{\sqrt{5}} - 1, \varphi^{-k}\right)$-approximation of x.*

Proof. We note that we can write every positive real number as a (possibly infinite) sum of powers of φ. We can write $x = \sum_{i=-\infty}^{s} a_i \varphi^i$ where $a_i \in \{0,1\}$ and where there is no i so that $a_i = 1$ and $a_{i+1} = 1$ would both hold [9].

We also note that given such a requirement, $\sum_{i=-\infty}^{j} a_i \varphi^i < \varphi^{j+1}$ holds for any j.

Thus, taking $j = -k$ if we choose to represent x as $x = \sum_{i=-k}^{s} a_i \varphi^i$, the error we make is no greater than φ^{-k}, that is, $|x - \sum_{i=-k}^{s} a_i \varphi^i| \leq \varphi^{-k}$.

The following three facts about Fibonacci numbers $\mathscr{F}_j$ ($j \in \mathbb{Z}$) are common knowledge and are easily proven:

- $\varphi^j = \mathscr{F}_j \varphi + \mathscr{F}_{j-1}$ for every j [42],
- $|\mathscr{F}_j| = |\mathscr{F}_{-j}|$ for every j [41],
- $\mathscr{F}_j \approx \frac{\varphi^j}{\sqrt{5}}$ for every positive j [41].

Considering this, if $x \approx \sum_{i=-k}^{s} a_i \varphi^i$, then we note that $\sum_{i=-k}^{s} a_i \varphi^i = \sum_{i=-k}^{s} a_i (\mathscr{F}_i \varphi + \mathscr{F}_{i-1})$ and thus we can represent x with $\sum_{i=-k}^{s} a_i \mathscr{F}_{i-1} + \varphi \sum_{i=-k}^{s} a_i \mathscr{F}_i$. We know that this is a φ^{-k}-approximation.

Let us now show that $\sum_{i=-k}^{s} a_i \mathscr{F}_{i-1}$ and $\sum_{i=-k}^{s} a_i \mathscr{F}_i$ can be bounded from above by $\mathscr{F}_{s+1} + \mathscr{F}_{k+2} - 1$.

Namely, we note that

$$\begin{aligned}\left|\sum_{i=-k}^{s} a_i \mathscr{F}_i\right| &\leq \sum_{i=-k}^{-1} a_i \mathscr{F}_i + \sum_{i=0}^{s} a_i \mathscr{F}_i \\ &\leq \sum_{i=1}^{k} a_i \mathscr{F}_i + \sum_{i=0}^{s} a_i \mathscr{F}_i \\ &< \mathscr{F}_{k+1} + \mathscr{F}_{s+1} - 1\,.\end{aligned}$$

The last inequality comes from Lemma 1. Likewise,

$$\sum_{i=-k}^{s} a_i \mathscr{F}_{i-1} < \mathscr{F}_s + \mathscr{F}_{k+2} - 1.$$

Both of these values can be bounded from above by $\mathscr{F}_{s+1} + \mathscr{F}_{k+2} - 1$, which is approximately equal to $\frac{\varphi^{s+1} + \varphi^{k+2}}{\sqrt{5}} - 1$.

Thus the lemma is proven.

□

We have obtained a method for finding a golden number representing a value with a desired precision, provided that we can compute the numbers a_i that describe the φ-decompostion of the number and perform elementary arithmetic operations. This is easy to do in the public setting. However, it might be more resource-costly in the private setting. Thus, one solution is to only use golden-section numbers throughout the secure computation session, so that we could input the secure values in a golden number format.

However, it is possible that we might wish to perform some secure computations in a different data format and only then convert to a secure golden number format. Thus, later, we will present protocols for securely converting a private fixed-point number to a private golden section number and *vice versa*.

We will presume the following protocols for this chapter:

- integer addition, subtraction, and multiplication,
- fixed-point number addition, subtraction and multiplication,
- $\mathsf{ObliviousChoiceProtocol}(\llbracket b \rrbracket, \llbracket x \rrbracket, \llbracket y \rrbracket)$,
- $\mathsf{BitExtract}(\llbracket x \rrbracket)$,
- $\mathsf{MSNZB}(\llbracket x \rrbracket)$.

6.1. Normalization

As it was previously noted, to get good precision in our number system, we use representatives that are rather large. This holds even when the real numbers that we represent are quite small. This, however, means that when multiplying several real numbers, the representatives may grow exponentially and may overflow very fast. Secure computation is especially vulnerable to these kinds of errors as checking for possible overflows in a secure way is expensive. We would like to keep the absolute value of the representatives smaller than some reasonable constant in order to prevent them from overflowing.

The solution for this comes from the fact that there may be several different (k, ε)-approximations of a number. Thus we want a method for replacing a (k_1, ε_1)-approximation with a (k_2, ε_2)-approximation where ε_2 may be slightly greater than ε_1, but where $k_2 << k_1$. We shall use a *normalization* method, which is, in essence, subtraction of a suitable representation of 0 from the golden section number.

Definition 3. We say that a golden section number is ℓ-normalized if the absolute value of its integer representative is not greater than ℓ.

This definition might seem deficient at first — it would seem more natural to demand that both integer representative and φ-representative were smaller than some bound. However, we shall see that if we slightly restrict the set of numbers that we represent, then the boundedness of the φ-representative follows from the boundedness of the integer representative. Thus it suffices to define ℓ-normalization using only the integer representative.

Lemma 3. *Let an ℓ-normalized golden section number $a-\varphi b$ satisfy the inequality $|a-\varphi b|\leq \ell(\varphi-1)$. Then $|b|\leq \ell$.*

Proof. Using the reverse triangle inequality, we obtain

$$||a|-|b\varphi||\leq |a-b\varphi|\leq \ell(\varphi-1).$$

From this we obtain

$$|b\varphi|-\ell(\varphi-1)\leq |a|\leq |b\varphi|+\ell(\varphi-1).$$

From $|b\varphi|-\ell(\varphi-1)\leq |a|$ we obtain $|b\varphi|-\ell\varphi+\ell\leq \ell$, i.e. $|b\varphi|\leq \ell\varphi$. This is equivalent to $|b|\leq \ell$. □

Considering this result, we must require that all the golden section numbers we use have absolute values less than $\ell(\varphi-1)$ in order for the definition of ℓ-normalization to have its intended meaning. Therefore, we need to agree on a certain value for ℓ so that from here on, we would assume that unless otherwise specified, all the golden numbers we use have absolute values less than $\ell(\varphi-1)$ for the ℓ that we will now specify.

One reasonable property that ℓ should have is that we should be able to multiply two ℓ-normalized golden section numbers with no possibility of overflow happening. On the other hand, ℓ should be as large as possible, because it bounds the choices we have for representatives and thus also precision — the larger ℓ is, the greater precision we can achieve. Because of this, we shall generally take $\ell=\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$, where n refers to the bit length of a and b.

The following lemma shows that no overflow will happen when multiplying two ℓ-normalized golden numbers.

Lemma 4. *If two golden section numbers $a-\varphi b$ and $c-\varphi d$ are $\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$-normalized, then, assuming that both they and their product is smaller than*

$$\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor(\varphi-1),$$

both the integer representative and the φ-representative of their product are smaller than 2^{n-1}.

Proof. Let us denote $(a-\varphi b)\cdot(c-\varphi d)$ with $x-\varphi y$, i.e. $x=ac+bd$ and $y=ad+bc-bd$. We give the proof for $x\geq 0$. The proof for $x<0$ is analogous. We assume that $|a|,|b|,|c|,|d|\leq \ell$. Thus $x=ac+bd\leq 2\ell^2<2^{n-1}$. We assumed that the absolute value of the product $x-\varphi y$ is no greater than $\ell(\varphi-1)$. In a similar way as we did in the proof of Lemma 3, we obtain that $|y\varphi|-\ell(\varphi-1)\leq x$. Thus $|y\varphi|\leq 2\ell^2+\ell(\varphi-1)$ which gives us $|y|<2\ell^2<2^{n-1}$. □

From now on, ℓ shall refer to $\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$. Likewise, when we speak of normalized numbers we will mean $\left\lfloor\sqrt{\frac{2^{n-1}-1}{2}}\right\rfloor$-normalized numbers.We will also assume that all the numbers that we will deal with from now on will have absolute values no greater than $\ell(\varphi-1)$. This can be thought of as a specific kind of overflow. In the private setting we do not have checks for overflow as that would leak information and it is assumed that the user will take care that the values do not overflow. The user can do this by for example only allowing inputs from a certain range, or adding oblivious checks that prevent possible overflows. We require that the user takes the similar necessary precautions also in this setting.

We want to significantly reduce the absolute values of the representatives of the number while keeping its value relatively the same.

There are various possibilities for this, but due to the nature of the task, they are equivalent to deducing suitable ε-representations of zero from the number. Namely, suppose that we normalized $a-\varphi b$ and ended up with $a'-\varphi b'$. Let the error made in this process be no greater than ε, i.e. $\left|a-\varphi b-(a'-\varphi b')\right|\leq\varepsilon$. This means that $|(a-a')-\varphi(b-b')|\leq\varepsilon$, i.e. $\left|(a-a')-\varphi(b-b')\right|$ is an ε-representation of 0. The smaller the ε, the smaller the error arising from the normalization process and thus it is desirable to obtain ε-representations of 0 where ε is very small. Also, the process should not be very resource-consuming. We first note that, thanks to Lemma 3, it suffices to normalize only the integer representative of the number. If the normalization error is small, then the result will still be an ε-representation of a golden number, and, thus, the absolute value of the φ-representative of the result will also be smaller than ℓ.

Note that in order to normalize an integer representative down to 2^k, we need either the $n-k$ most significant bits to be zero (if this repesentand is positive) or the $n-k$ most significant bits to be one (if it is negative) in the end-result. In principle, k can be chosen freely, but we generally use $k=\frac{n}{2}-1$. There are several possibilities for normalization which we shall now describe.

6.1.1. First Normalization Method

First, for the sake of simplicity, let us consider the case where the integer representative is non-negative. We will discuss the negative case later, and it will be very similar to the positive case.

As we noted above, we need that after normalization, the $n-k$ most significant bits of the integer representative should be zero. Using the protocol $\mathsf{BitExtract}(\llbracket\cdot\rrbracket)$, we can have access to the individual bits of a. Thus we arrive at the idea of normalizing the integer representative bitwise.

We noted that normalization can be essentially thought of as subtracting suitable representations of 0. We use this idea by building a basis from 0-representations where the basis elements correspond to different powers of 2.

First we will give an example that does not work for reasons specified afterwards, but introduces the general idea. Namely, we will try to use the set of golden numbers $(2^k, \left\lceil \frac{2^k}{\varphi} \right\rfloor, 0)$, $(2^{k+1}, \left\lceil \frac{2^{k+1}}{\varphi} \right\rfloor, 0), \ldots (2^{n-2}, \left\lceil \frac{2^{n-2}}{\varphi} \right\rfloor, 0)$ for the basis vectors.

Thus, if we want to normalize a golden number $a - \varphi b$, we would perform $\mathsf{BitExtract}(\llbracket a \rrbracket)$ to obtain the bits $a_{n-1}, \ldots, a_0$. For every bit a_i in $\{a_{n-2}, a_{n-3}, \ldots, a_k\}$ that is equal to 1 we subtract 2^i from a and $\left\lceil \frac{2^k}{\varphi} \right\rfloor$ from b. Note that this can be easily done by just subtracting $\sum_{i=k}^{n-2} a_i 2^i$ from a and subtracting $\sum_{i=k}^{n-2} a_i \left\lceil \frac{2^k}{\varphi} \right\rfloor$ from b. The integer representative of our result will be $a - \sum_{i=k}^{n-2} a_i 2^i = \sum_{i=0}^{k-1} a_i 2^i$ and that is certainly smaller than $2^k - 1$.

However, there is a problem with this approach. Namely, in the general case, the numbers $(2^i, \left\lceil \frac{2^i}{\varphi} \right\rfloor)$ are very poor approximations for 0. Intuitively speaking, this was to be expected. Generally, as suggested by the Weyl equidistribution theorem, we would expect that if we picked an integer c randomly, then the value $\left| c - \left\lceil \frac{c}{\varphi} \right\rfloor \varphi \right|$ would fall somewhere in the interval $[0, \frac{\varphi}{2})$ with roughly uniform probability. Because the powers of 2 do not have any special property that would cause the values $\left| 2^i - \left\lceil \frac{2^i}{\varphi} \right\rfloor \varphi \right|$ to be especially close to 0, they are similar to most other integers in that they make for poor integer representative in the representations of 0.

Thus, we need to modify our algorithm somewhat. We want to find numbers x_i that are close to 2^i but where $\left| \varphi \left\lceil \frac{x_i}{\varphi} \right\rfloor - x_i \right|$ is very small. On the other hand, we also wish $|2^i - x_i|$ to be small, since if $|2^i - x_i|$ is too large, then the number that we obtain after the normalization process is still not normalized.

We shall now describe the algorithm for normalization. We use a number of pairs of public constants (x_i, y_i) such that $x_i - \varphi y_i \approx 0$ and that x_i is close to 2^i. After describing the algorithm we shall show what properties they must satisfy. The algorithm that we have described is formalized in Algorithm 20.

We are given a golden section number $\llbracket a \rrbracket - \varphi \llbracket b \rrbracket$. We perform bit decompo-

sition on $[\![a]\!]$ and obtain its bits $[\![a_0]\!], \ldots, [\![a_{n-1}]\!]$. Out of these, we are interested in bits with large indices as the less significant bits will not be important in normalizing. As it was noted before, we will at first consider only the positive case. In the algorithm, we will compute both the positive case and the negative case and then use a_{n-1} to obliviously choose between them.

If $a_i = 1$ and $n-1 > i \geq k$, we will subtract x_i from a and y_i from b. This is done by multiplying x_i with $[\![a_i]\!]$ and subtracting $[\![x_i a_i]\!]$ from $[\![a]\!]$, and likewise, multiplying y_i with $[\![a_i]\!]$ and subtracting $[\![y_i a_i]\!]$ from $[\![b]\!]$.

Likewise, in the negative case, if $a_i = 0$ and $n-1 > i \geq k$, we will add x_i to a and y_i to b.

Algorithm 20: GoldenNorm

Data: $[\![a]\!], [\![b]\!], \{x_i\}_{i=k}^{n}, \{y_i\}_{i=k}^{n}$

Result: Given a golden section number and a normalization set, returns the number normalized according to the set.

$\{[\![a_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}([\![a]\!])$
$\{[\![z_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![a_i]\!]\}_{i=k}^{n-2} \cdot \{x_i\}_{i=k}^{n-2}$
$\{[\![w_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![a_i]\!]\}_{i=k}^{n-2} \cdot \{y_i\}_{i=k}^{n-2}$
$\{[\![z'_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![1-a_i)]\!]\}_{i=k}^{n-2} \cdot \{x_i\}_{i=k}^{n-2}$
$\{[\![w'_i]\!]\}_{i=k}^{n-2} \leftarrow \{[\![(1-a_i)]\!]\}_{i=k}^{n-2} \cdot \{y_i\}_{i=k}^{n-2}$
for $i \leftarrow k$ **to** $n-2$ **do**
    $[\![a']\!] \leftarrow [\![a]\!] - [\![z_i]\!]$
    $[\![b']\!] \leftarrow [\![b]\!] - [\![w_i]\!]$
    $[\![a'']\!] \leftarrow [\![a]\!] + [\![z'_i]\!]$
    $[\![b'']\!] \leftarrow [\![b]\!] + [\![w'_i]\!]$
$[\![a]\!] \leftarrow \mathsf{ObliviousChoice}([\![a_{n-1}]\!], [\![a']\!], [\![a'']\!])$
$[\![b]\!] \leftarrow \mathsf{ObliviousChoice}([\![a_{n-1}]\!], [\![b']\!], [\![b'']\!])$
return $[\![a]\!], [\![b]\!]$

Now we will explain what properties the pairs $\{(x_i, y_i)\}_{i=k}^{n-2}$ must satisfy so that the integer representative of the final result would have an absolute value no greater than ℓ and that the final results difference from the original golden number would be no greater than ε.

We want the end result, which in the positive case is $a - \sum_{i=k}^{n-2} a_i x_i$ and in the negative case is $a + \sum_{i=k}^{n-2} (1-a_i) x_i$, to be in the interval $(-\ell, \ell)$. We note that in the positive case the following equality holds:

$$a - \sum_{i=k}^{n-2} a_i x_i = \sum_{i=0}^{k-1} a_i 2^i + \sum_{i=k}^{n-2} a_i (2^i - x_i).$$

Likewise, in the negative case the following holds:

$$a+\sum_{i=k}^{n-2}(1-a_i)x_i = -1+\sum_{i=0}^{k-1}(a_i-1)2^i+\sum_{i=k}^{n-2}(a_i-1)(2^i-x_i).$$

In attempting to estimate these quantities with inequalities, it is important whether 2^i is smaller or greater than x_i. Thus, by distinguishing these cases, we can bound the values of the normalized numbers from both below and above and, thus, arrive at the following inequalities:

$$\begin{aligned}\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}}(2^i-x_i) &\le \sum_{i=0}^{k-1}a_i2^i+\sum_{i=k}^{n-2}a_i(2^i-x_i)\\ &\le 2^k-1+\sum_{\substack{i':2^{i'}>x_i\\ k\le i\le n-2}}(2^{i'}-x_{i'}),\end{aligned}$$

and

$$\begin{aligned}\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}}(x_i-2^i) &\ge -1+\sum_{i=0}^{k-1}(a_i-1)2^i+\sum_{i=k}^{n-2}(1-a_i)(x_i-2^i)\\ &\ge -2^k+\sum_{\substack{i:2^i>x_i\\ k\le i\le n-2}}(x_i-2^i).\end{aligned}$$

We thus have upper and lower bounds to the normalized numbers. Considering that, in order to achieve that $a-\sum_{i=k}^{n-2}a_ix_i$ or $a+\sum_{i=k}^{n-2}(1-a_i)x_i$ belongs the interval $(-\ell,\ell)$, it suffices for both cases that

$$-\ell \le \sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}}(2^i-x_i)$$

and

$$2^k+\sum_{\substack{i':2^{i'}>x_i\\ k\le i\le n-2}}(2^{i'}-x_{i'})\le \ell.$$

Thus we arrive at the following definition.

Definition 4. A (k,ℓ,ε,n)-normalization set is a set of integers $\{x_k,\dots,x_{n-1},y_k,\dots,y_{n-1}\}$ with the following properties:

1. $\sum_{i=k}^{n-2}|x_i-\varphi\cdot y_i|\le\varepsilon$,
2. $\sum_{\substack{i:2^i<x_i\\ k\le i\le n-2}}(2^i-x_i)\ge-\ell$,

3. $2^k + \sum_{\substack{i':2^{i'}>x_i \\ k\le i\le n-2}} (2^{i'} - x_{i'}) \le \ell$.

There is some freedom in choosing k, with lower values giving us more freedom in choosing the normalization set, but reducing the number of values that a normalized number can have. We will now show how to find such sets.

6.1.2. Finding Normalization Sets

In essence, finding normalization sets is about finding good 0-approximations $x_i - \varphi y_i$ with the additional constraint that the x_i should not be too far from the respective powers of two 2^i. It is an optimization problem: we intend to minimize $\sum_{i=k}^{n-2} |x_i - \varphi \cdot y_i|$ with the constraints 2 and 3 from Definition 4. Note that it does not really matter how tightly or loosely the constraints 2 and 3 are satisfied. If they are satisfied, then we can be sure that the result of the normalizing process will be a normalized number, however, we will not gain any extra benefits if they, in fact, satisfy stronger constraints.

On the other hand, the smaller we make the sum $\sum_{i=k}^{n-2} |x_i - \varphi \cdot y_i|$, the smaller the maximal possible error will be. More specifically, what matters is the sum of the positive errors and the sum of the negative errors. It is clear that if some errors are positive and others negative, they will cancel each other out by some amount and thus the total error will be smaller. Hence, the largest possible error can happen if all the errors have the same sign.

Our solution to the optimization problem has two main components. First, finding pairs $(x_{i,j}, y_{i,j})$ so that $x_{i,j} - \varphi y_{i,j}$ are as small as possible so that $x_{i,j} \in [a_i, b_i]$. We will describe later how a_i, b_i will be chosen. The second part of our solution is picking the suitable pair $(x_{i,j}, y_{i,j})$ for every i so that the constraints 2 and 3 in Definition 4 hold and the maximal error is very small.

We shall first consider the first component. This part of the optimization problem is doable in time that is logarithmic to $b_i - a_i$. It does require a considerable amount of space, though, and it is necessary to be careful when tuning the constants to keep the space costs practical. We first note that in order to find such pairs $x_{i,j} - \varphi y_{i,j}$ it suffices to search only for numbers $x_{i,j}$ in $[a_i, b_i]$ where $\varphi \left[\frac{x_{i,j}}{\varphi} \right] - x_{i,j}$ is small, or, alternatively, for $y_{i,j}$ in $\left[\frac{a_i}{\varphi}, \frac{b_i}{\varphi} \right]$ so that $\varphi y_{i,j} - [\varphi y_{i,j}]$ is small. We can search in either of those intervals as good solutions in one of the intervals correspond to good solutions in the other. As the interval $\left[\frac{a_i}{\varphi}, \frac{b_i}{\varphi} \right]$ is shorter than $[a_i, b_i]$, the search problem is easier there. For this reason, we will frame our question as finding $y_{i,j}$ in that interval so that $\varphi y_{i,j} - [\varphi y_{i,j}]$ would be small.

Because we will be dealing with the values $\varphi c - [\varphi c]$ and $\varphi \left[\frac{d}{\varphi}\right] - d$ quite a lot, let us define for integers c, d the functions $\mathrm{err}(c) := \varphi c - [\varphi c]$ and $\mathrm{err}'(d) := \varphi \left[\frac{d}{\varphi}\right] - d$. These values measure how well the pairs $([\varphi c], c)$ $(d, \left[\frac{d}{\varphi}\right])$ would represent 0. These two measurements are intrinsically connected.

Lemma 5. *For all $d \in \mathbb{Z}$, $\mathrm{err}'([\varphi d]) = \mathrm{err}(d)$.*

Proof. Let ε_d be the number so that $\varphi d - \varepsilon_d \in \mathbb{Z}$ and $|\varepsilon_d| < 0.5$, i.e. ε_d measures how far φd is from the closest integer. We note that it is impossible for $|\varepsilon_d|$ to be equal to 0.5, as then φ would be rational. We now observe that since $\varphi d - \varepsilon_d = [\varphi d]$, $\mathrm{err}(d) = \varphi d - [\varphi d] = \varepsilon_d$. On the other hand

$$\mathrm{err}'([\varphi d]) = \varphi \left[\frac{(\varphi d - \varepsilon_d)}{\varphi}\right] - (\varphi d - \varepsilon_d) = \varphi d - \varphi d + \varepsilon_d = \varepsilon_d.$$

Thus $\mathrm{err}'([\varphi d]) = \mathrm{err}(d)$. □

From now on, we will refer to $\mathrm{err}(y)$ as the error of y.We will now present some results that will be necessary for the algorithm. The proof of the following lemma is trivial.

Lemma 6. *For all $x \in \mathbb{Z}$, $\mathrm{err}(-x) = -\mathrm{err}(x)$.*

The following lemma shows us the condition under which the error function is linear.

Lemma 7. *Let a and b be integers with the property that $|\mathrm{err}(a)| + |\mathrm{err}(b)| < \frac{1}{2}$. Then*

$$\mathrm{err}(a+b) = \mathrm{err}(a) + \mathrm{err}(b).$$

Proof. Let us consider the cases for the signs of $\mathrm{err}(a)$ and $\mathrm{err}(b)$. Both of them can be either non-negative or negative, resulting in four total cases. We will give the proof for $\mathrm{err}(a) \geq 0$ and $\mathrm{err}(b) \geq 0$, the proofs for the other cases are analogous.

Let $\mathrm{err}(a) \geq 0$ and $\mathrm{err}(b) \geq 0$. Thus $[\varphi a] \leq \varphi a$ and $[\varphi b] \leq \varphi b$.

We shall partition $\varphi a = [\varphi a] + \mathrm{err}(a)$ and $\varphi b = [\varphi b] + \mathrm{err}(b)$.

Because $\mathrm{err}(a) \geq 0$ and $\mathrm{err}(b) \geq 0$ and $|\mathrm{err}(a)| + |\mathrm{err}(b)| < \frac{1}{2}$, $\mathrm{err}(a) + \mathrm{err}(b) \in [0, \frac{1}{2})$.

Now it holds that

$$\varphi(a+b) = \varphi a + \varphi b = [\varphi a] + \mathrm{err}(a) + [\varphi b] + \mathrm{err}(b).$$

Note now that $\varphi(a+b)$ is the sum of an integer $[\varphi a] + [\varphi b]$ and a real number $\mathrm{err}(a) + \mathrm{err}(b)$ that has absolute value smaller than $\frac{1}{2}$. Thus $[\varphi a] + [\varphi b]$ must be

the closest integer to $(a+b)\varphi$ and thus $[(a+b)\varphi] = [a\varphi] + [b\varphi]$ and

$$\begin{aligned}\operatorname{err}(a+b) &= \varphi(a+b) - [\varphi(a+b)] \\ &= ([\varphi a] - \operatorname{err}(a) + [\varphi b] - \operatorname{err}(b)) - [\varphi a] + [\varphi b] \\ &= \operatorname{err}(a) + \operatorname{err}(b).\end{aligned}$$

Thus the claim holds.

□

We can expand this easily to the following lemma.

Lemma 8. *Let k be a positive integer. If $k \cdot |\operatorname{err}(y)| < \frac{1}{2}$, then $\operatorname{err}(k \cdot y) = k \cdot \operatorname{err}(y)$.*

Proof. We prove this by induction over k. The case when $k = 1$ is obvious. Let now the statement hold for $k = j$ and let us show that it holds for $k = j+1$. We thus assume that $(j+1) \cdot |\operatorname{err}(y)| < \frac{1}{2}$. Thus also $j \cdot |\operatorname{err}(y)| < \frac{1}{2}$ and thus $\operatorname{err}(j \cdot y) = j \cdot \operatorname{err}(y)$. Now, $|\operatorname{err}(j \cdot y)| + |\operatorname{err}(y)| = j \cdot \operatorname{err}(y) + \operatorname{err}(y) = (j+1)\operatorname{err}(y) < \frac{1}{2}$ and thus $\operatorname{err}(j \cdot y + y) = \operatorname{err}(j \cdot y) + \operatorname{err}(y) = (j+1)\operatorname{err}(y)$. Hence, the statement is proven. □

We can obtain the following corollary from the previous three lemmas.

Corollary 1. *Let k and l be positive integers. If $k \cdot |\operatorname{err}(y)| + l \cdot |\operatorname{err}(w)| < \frac{1}{2}$, then $\operatorname{err}(\pm k \cdot y \pm l \cdot w) = \pm k \cdot \operatorname{err}(y) + \pm l \cdot \operatorname{err}(w)$.*

We also need to note that generally, $\operatorname{err}(\mathscr{F}_k)$ tend to be very small. More precisely, as $\mathscr{F}_k = \dfrac{\varphi^k - (-\varphi)^{-k}}{\sqrt{5}}$, and we know that $[\varphi \mathscr{F}_k] = \mathscr{F}_{k+1}$ for all positive k, we get that

$$\begin{aligned}\operatorname{err}(\mathscr{F}_k) &= \varphi\mathscr{F}_k - \mathscr{F}_{k+1} \\ &= \varphi\frac{\varphi^k - (-\varphi)^{-k}}{\sqrt{5}} - \frac{\varphi^{k+1} - (-\varphi)^{-k-1}}{\sqrt{5}} \\ &= \frac{\varphi^{k+1} - \varphi(-\varphi)^{-k} - \varphi^{k+1} + (-\varphi)^{-k-1}}{\sqrt{5}} \\ &= \frac{(-\varphi)^{-k-1} - \varphi(-\varphi)^{-k}}{\sqrt{5}} \\ &= \frac{(-\varphi)^{-k}}{\sqrt{5}}((-\varphi)^{-1} - \varphi) \\ &= \frac{(-\varphi)^{-k}}{\sqrt{5}}(-\sqrt{5}) \\ &= -(-\varphi)^{-k}\end{aligned} \tag{6.2}$$

Now we are ready to explain the first part of the algorithm. Suppose that in some interval $[a,b]$ we have found all such elements g that $\operatorname{err}(g) \leq \alpha$. For

shorthand we denote this set with G. If this interval $[a,b]$ is small enough, those elements can be found by brute force, if it is larger, however, they can be found recursively using the same method. Nevertheless, we presume that α has been chosen so that G is relatively small.

Now suppose that we want to find all elements g' in a larger interval $[a,b']$ with a smaller error. More specifically, we want to find all such g' so that $\text{err}(g') \leq \alpha'$ in an interval $[a,b']$ where $\alpha' << \alpha$ and $b-a << b'-a$. Let us denote this set with G'. We will show that then we can find all such elements g' by only checking at most $\lceil \frac{b'-a}{b-a} \rceil \cdot |G|$ elements.

The idea is that because of the linearity of $\text{err}(\cdot)$ and the smallness of the errors of the Fibonacci numbers, the distance of any element g' with a small error from some element of G must be a multiple of a $\mathscr{F}_k$, where $\mathscr{F}_k$ is a suitably chosen large enough Fibonacci number.

More specifically, let $\mathscr{F}_k$ be such a Fibonacci number so that $b-a > \mathscr{F}_k$ and $\alpha' + \text{err}(\mathscr{F}_k)\frac{b'-a}{\mathscr{F}_k} \leq \alpha$. This is only possible if b' and α' are suitably chosen. Thus, attention must be paid when choosing a new α' and b' so that such a Fibonacci number would exist. We also want to choose $\mathscr{F}_k$ in such a manner that $\frac{b'-a}{\mathscr{F}_k}$ would be relatively small.

Now consider the set $H := \{g + i \cdot \mathscr{F}_k | g \in G, g + i \cdot \mathscr{F}_k \in [a,b'], i \in \mathbb{Z}\}$. We claim that $G' \subseteq H$. Because the size of H is upper bound by $\frac{b'-a}{\mathscr{F}_k}|G|$ and we chose our parametres so that $\frac{b'-a}{\mathscr{F}_k}$ and $|G|$ would not be very large, we can search the set H for the set G'. We shall now state this claim as a theorem and prove it.

Theorem 3. *Let $a < b < b'$ and $G := \{g \in [a,b] || \text{err}(g)| \leq \alpha, g \in \mathbb{Z}\}$. Let $\mathscr{F}_k$ be such that $b-a \geq \mathscr{F}_k$ and also $\alpha' + |\text{err}(\mathscr{F}_k)| \lfloor \frac{b'-a}{\mathscr{F}_k} \rfloor \leq \alpha$ and $\alpha < \frac{1}{2}$. Let G' be defined as $G' := \{g' \in [a,b'] || \text{err}(g')| \leq \alpha', g' \in \mathbb{Z}\}$ and H be defined as $H := \{g + i \cdot \mathscr{F}_k | g \in G, g + i \cdot \mathscr{F}_k \in [a,b'], i \in \mathbb{Z}\}$. Then $G' \subseteq H$.*

Proof. Let us consider for all $g' \in G'$ the set

$$H_{g'} := \{g' - i \cdot \mathscr{F}_k | 0 \leq i \leq \frac{b'-a}{\mathscr{F}_k}, i \in \mathbb{Z}\}.$$

We shall first see that, for all $h \in H_{g'}$ and for all $g' \in G'$ the equation $|\text{err}(h)| \leq \alpha$ is satisfied. Indeed, we see that if $h \in H_{g'}$, then $h = g' - j\mathscr{F}_k$ for some j. Because $|\text{err}(g')| + j \cdot |\text{err}(\mathscr{F}_k)| < \frac{1}{2}$, the error of h is linear and $\text{err}(h) = \text{err}(g') - j \cdot \text{err}(\mathscr{F}_k)$. By the triangle inequality we get $|\text{err}(h)| \leq \alpha' + j \cdot |\text{err}(\mathscr{F}_k)|$. We

know that j is an integer no greater than $\frac{b'-a'}{\mathscr{F}_k}$ and hence $j \leq \lfloor \frac{b'-a'}{\mathscr{F}_k} \rfloor$. Thus

$$|\mathrm{err}(h)| \leq \alpha' + \lfloor \frac{b'-a'}{\mathscr{F}_k} \rfloor |\mathrm{err}(\mathscr{F}_k)| \leq \alpha.$$

Second, we shall see that for all $g' \in G'$ the intersection of $H_{g'}$ and $[a,b]$ is nonempty. Either $g' \in G$, in which case this trivially holds, as $g' \in H_{g'}$, or there exists the smallest i such that $g' - i \cdot \mathscr{F}_k > b$. Now consider $g' - (i+1) \cdot \mathscr{F}_k$. If $g' - (i+1) \cdot \mathscr{F}_k < a$, then

$$b < g' - i \cdot \mathscr{F}_k < a + \mathscr{F}_k \leq a + b - a = b$$

which would be a contradiction. Thus $a \leq g' - (i+1) \cdot \mathscr{F}_k \leq b$ and thus $g' - (i+1) \cdot \mathscr{F}_k \in [a,b]$. Now note that as $g' \leq b'$, we get that $a \leq b' - (i+1) \cdot \mathscr{F}_k$, i.e $i+1 \leq \frac{b'-a}{\mathscr{F}_k}$. Thus $g' - (i+1) \cdot \mathscr{F}_k \in H_{g'}$ and hence $g' - (i+1) \cdot \mathscr{F}_k \in [a,b] \cap H_{g'}$.

These two observations combined give us that for any $g' \in G'$ the intersection of $H_{g'}$ and G is nonempty.

Now the main result easily follows. Consider any $g' \in G'$. We shall show that $g' \in H$. We know that there must exist an element $g \in G \cap H_{g'}$. We know that for some j, $g = g' - j \cdot \mathscr{F}_k$ and thus also $g' = g + j \cdot \mathscr{F}_k$. Based on the definition of H we obtain that $g' \in H$.

□

Hence, we have a method for obtaining the set of values with a smaller error in a larger interval from the set of values with a larger error in a smaller interval. Alternatively, we can describe this as a method for obtaining the set that contains all the elements g' from $[a,b']$ with $|\mathrm{err}(g')| \leq \alpha'$ from the set that contains all the elements g from $[a,b]$ with $|\mathrm{err}(g)| \leq \alpha$ where $b' - a >> b - a$ and $\alpha' << \alpha$.

This is formalized in Algorithm 21.

Algorithm 21: Recursion step

Data: $G, \alpha, \alpha', a, b, b', \mathscr{F}_k$

Result: Takes in the set G of elements $g \in [a,b]$ that satisfy $|\mathrm{err}(g)| \leq \alpha$. Also takes in $\mathscr{F}_k$ that satisfies $\mathscr{F}_k < b-a$ and $\alpha' + \mathrm{err}(\mathscr{F}_k)\lfloor \frac{b'-a}{\mathscr{F}_k} \rfloor \leq \alpha$. The algorithm outputs the set G' that contains all the elements g' in $[a,b']$ for which $|\mathrm{err}(g')| \leq \alpha'$.

1 **foreach** $g \in G$ **do**
2 $\quad H_g \leftarrow \{g + i \cdot \mathscr{F}_k | g \in G, g + i \cdot \mathscr{F}_k \in [a,b']\}$
3 $H \leftarrow \cup_{g \in G} H_g$
4 $G' \leftarrow \{\}$
5 **foreach** $h \in H$ **do**
6 $\quad$ **if** $|\mathrm{err}(h)| \leq \alpha'$ **then**
7 $\quad\quad G' \leftarrow G' \cup h$
8 **return** G'

For all elements g of G we find the set H_g. Then we take the union over all the H_g on line 3 and then find the elements with a sufficiently small error in that set by checking every element on lines 4 to 7. We assume that parametres were chosen in a way that this union is small enough and thus this step should be rather fast.

This is the main tool for recursively finding the elements y with the smallest $|\mathrm{err}(y)|$ from any interval $[a,b]$. However, there are still some technical problems that must be answered. For example, how to choose the different constants such as α and α', the $\mathscr{F}_k$ and b. In the beginning, we just have a possibly very large interval $[a,b]$ and we want to find either the element y with the smallest error or k best elements for some small value of k.

Weyl's equidistribution theorem suggests that if we pick consecutive integers $a, a+1, \ldots, b$, then approximately $2(b-a)\alpha$ of them have errors with absolute values smaller than α, that is

$$|\{y \in [a,b] || \mathrm{err}(y)| < \alpha\}| \approx 2(b-a)\alpha. \tag{6.3}$$

However, Weyl's equidistribution theorem does not say anything about the speed of the convergence and, thus, we need a different result if we want to quantify how well this intuition holds for different $b-a$ and α.

For this we need the following definition.

Definition 5. Let $[d_0, d_1, \ldots]$ be the continued fraction representation of a real number β. Let $\frac{p_n}{q_n}$ be defined as $[d_0, d_1, \ldots, d_n]$. We call q_n the n-th partial quotient denominator of β.

Note that for φ the partial quotient denominators are precisely the Fibonacci numbers. We shall now give a result from [43] which it follows that if $b-a$ is

a Fibonacci number $\mathscr{F}_k$ and $\alpha = \frac{1}{k}$, then the equation in 6.3 holds with equality. Also, for the sake of brevity, we will denote with $m(x)$ the value $x \pmod 1$, i.e, the fractional part of x.

Lemma 9. *[43] If β is an irrational number and q is a partial quotient denominator of β, then every interval $\left[\frac{i}{q}, \frac{i+1}{q}\right)$ contains precisely one of the points $(m(i\cdot\beta))_{k=0}^{q-1}$.*

However, we would like that the claim would also hold if $b-a$ is not a Fibonacci number. Also, it would be useful to consider number systems that use different irrational number instead of φ. We would like the claim also hold for other values of β.

We now give a present a lemma that is helpful for estimating the distribution of elements for the case when $b-a$ is not necessarily a Fibonacci number.

Lemma 10. *Let $k \in \mathbb{Z}_+$. Let $\frac{n}{\mathscr{F}_k} \leq 2\alpha < \frac{n+1}{\mathscr{F}_k}$ for some integer n. Let a,b,m,t be such non-negative integers that $b = a + m\mathscr{F}_k + t$ and let $0 \leq t < \mathscr{F}_k$. Then $mn \leq |\{y \in [a,b] || \mathrm{err}(y)| \leq \alpha\}| \leq mn + 2m + t$.*

Proof. Let $\gamma_j := (a + j\mathscr{F}_k)\varphi$. Let $R_j := \{y \in [a + j\mathscr{F}_k, a + (j+1)\mathscr{F}_k - 1] || \mathrm{err}(y)| \leq \alpha\}$ for $j \in \{0,\ldots,s-1\}$. Note that due to Lemma 9, every interval

$$\left[m\left(\gamma_j + \frac{i}{\mathscr{F}_k}\right), m\left(\gamma_j + \frac{i+1}{\mathscr{F}_k}\right)\right]$$

holds exactly one element of the set $\{m(y\varphi)\,|y \in [a + j\mathscr{F}_k, a + (j+1)\mathscr{F}_k - 1]\}$. Now note that the interval $[-\alpha,\alpha]$ certainly contains n of these intervals and is contained in $n+2$ of them. Thus we obtain that $n \leq |R_j| \leq n+2$. By summing over all the R_j and adding the final t points that do not belong in any of the R_j, we get that $mn \leq |\{y \in [a,b] || \mathrm{err}(y)| \leq \alpha\}| \leq mn + 2m + t$.

□

Note that the upper bound is not very strict here as for equality to be achieved, all the last t points would have to lie in $[-\alpha,\alpha]$, which is unlikely as the points tend to be evenly distributed. However, as obtaining more accurate estimates is not our goal, we will not give them.

Based on Lemma 10, we obtain the following estimate.

Corollary 2. *Let k be such that $\mathscr{F}_{k-1}^2 < b-a \leq \mathscr{F}_k^2$ and $\mathscr{F}_k \geq 3$. Let $\alpha < 0.5$. Then $||\{y \in [a,b] || \mathrm{err}(y)| < \alpha\}| - 2(b-a)\alpha| \leq 3\mathscr{F}_k - 2$.*

Proof. Let $b = a + m\mathscr{F}_k + t$ where $0 \leq t < \mathscr{F}_k$. Let n be such that $\frac{n}{\mathscr{F}_k} \leq 2\alpha < \frac{n+1}{\mathscr{F}_k}$.

Now, $\frac{n}{\mathscr{F}_k}(m\mathscr{F}_k+t) \leq 2(b-a)\alpha \leq \frac{n+1}{\mathscr{F}_k}(m\mathscr{F}_k+t)$. This simplifies to $nm \leq 2(b-a)\alpha \leq nm+m+\frac{(n+1)t}{\mathscr{F}_k} \leq nm+m+n+1$.

Thus, $|\{y \in [a,b] || \mathrm{err}(y)| < \alpha\}| - 2(b-a)\alpha| \leq m+n+1, 2m+t$. Hence, we are interested in choosing a $\mathscr{F}_k$ in such a manner that would minimize both $m+n+1$ and $2m+t$. Note that $m+n+1 \leq m+1+2\mathscr{F}_k\alpha < m+2$.

We can also rewrite the relationship between a and b as $b = a+(m+1)\mathscr{F}_k + (\mathscr{F}_k - t)$ which gives us $b-a > (m+1)\mathscr{F}_k$. On the other hand $\mathscr{F}_k^2 \geq b-a$, thus $\mathscr{F}_k^2 > (m+1)\mathscr{F}_k$ which gives us $m+1 < \mathscr{F}_k$.

Thus we get that $m+n+1 < m+2 < \mathscr{F}_k+1$ on one hand and $2m+t \leq 2\mathscr{F}_k-2+t$ on the other. Assuming that $\mathscr{F}_k \geq 3$, we get that $\mathscr{F}_k+1 \leq 2\mathscr{F}_k-2+t$.

Thus we obtain that $|\{y \in [a,b] || \mathrm{err}(y)| < \alpha\}| - 2(b-a)\alpha| \leq 2\mathscr{F}_k - 2 + t$.

Now note that as $t \leq \mathscr{F}_k$, we get that

$$|\{y \in [a,b] || \mathrm{err}(y)| < \alpha\}| - 2(b-a)\alpha| \leq 3\mathscr{F}_k - 2.$$

□

Recall that we intend to call Algorithm 21 several times recursively in order to find the set of results. Suppose that we want to find the k elements with the smallest absolute errors in $[a,b]$. We cannot start the recursion with $[a,b]$ as it is too big. We need to start from some smaller interval that we can search with brute-force.

Let the number of recursion steps be s. We will later estimate what the value of s should be. We, thus, start with some small interval $[a,b_s]$ and a relatively large maximal allowed error α_s. At the first step we perform a brute-force search and obtain a solution set $G_s\{y \in [a,b_s] || \mathrm{err}(y)| \leq \alpha_s\}$ for that interval and maximal allowed error. After that, we start recursively applying Algorithm 21. We use G_{i+1} as an input and obtain better sets of results G_i where the interval is larger and the maximal allowed result is smaller. Note that if some intermediate set of results G_{i+1} is empty, then the algorithm stops and we have to restart it from some previous point. This is undesirable. On the other hand, if some intermediate set of results is too large, then it might be difficult to store it in the memory and also Algorithm 21 may be too costly. Thus we must be careful when choosing the b_i and α_i for the intermediate steps i. We also should estimate how large the original b_s and α_s should be and how many recursion steps are needed.

We will now try to obtain estimates for these parametres. We start with the interval $[a,b_0]$ and α_0. We supposed that we want to find k elements Here $b_0 = b$. We now take $\alpha_0 := \frac{k}{2(b-a)}$. By using Corollary 2 as an estimate, we can expect that there are approximately k elements y with $|\mathrm{err}(y)| \leq \alpha_0$ in $[a,b]$.

We may also let α_0 be slightly larger if we want to be more sure about obtaining at least k elements with the corresponding error. We now need to find suitable

candidates for $b_1, b_2, \ldots b_s$ and $\alpha_1, \alpha_2, \ldots \alpha_s$ which will respectively be the endpoints of the interval and the maximal allowed errors for the intermediate steps in the recursive algorithm. Let us denote $G_j := \{y \in [a, b_j] || \mathrm{err}(y)| \leq \alpha_j\}$.

We will start by finding G_s by brute force, as the interval $[a, b_s]$ should be small enough to be searchable by brute force. Then, at every recursive step, we will find G_j by using G_{j+1} where we will be using the Fibonacci number $\mathscr{F}_{k_j}$. At the end we will obtain G_0 which is the desired result.

The main constraint that we will face here will probably be the space constraint — we will need to keep $\frac{b_j - a}{\mathscr{F}_{k_j}} |G_{j+1}|$ elements in memory. Let K be the number of elements that we can comfortably store without loss of performance.

Given a b_j, we shall always pick the largest possible Fibonacci number allowed, thus making $\frac{b_j - a}{\varphi} \leq \mathscr{F}_{k_j} \leq b_j - a$. Seeing as $|G_{j+1}| \approx 2(b_{j+1} - a)\alpha_{j+1}$, if $2\varphi(b_j - a)\alpha_j \leq K$ holds, then $\frac{b_j - a}{\mathscr{F}_{k_j}} |G_{j+1}| \leq K$ also holds.

Thus,

$$\frac{b_j - a}{\mathscr{F}_{k_j}} |G_{j+1}| \approx 2(b_{j+1} - a)\alpha_{j+1}.$$

Suppose that we have found b_j and α_j. Now we need that

$$\alpha_j + |\mathrm{err}(\mathscr{F}_{k_{j+1}})| \cdot \frac{c_j}{\mathscr{F}_{k_{j+1}}} \leq \alpha_{j+1}.$$

Knowing estimates on $\mathrm{err}(\mathscr{F}_k)$ and $\mathscr{F}_k$, we get that this approximately simplifies to

$$\alpha_j + \frac{\sqrt{5} c_j}{\mathscr{F}^2_{k_{j+1}}} \leq \alpha_{j+1}. \tag{6.4}$$

As $\frac{1}{c_{j+1}} \geq \frac{1}{\mathscr{F}_{k_{j+1}}} \geq \frac{\varphi}{c_{j+1}}$, we get that if

$$\alpha_j + \frac{\sqrt{5} c_j}{c^2_{j+1}} \leq \alpha_{j+1} \tag{6.5}$$

holds, then Equation 6.4 also holds. Every step we will choose $\alpha_{j+1} := \frac{K}{2\varphi(b_{j+1} - a)}$ which gives $(b_{j+1} - a) = \frac{K}{2\varphi\alpha_{j+1}}$. Let us take $d_j := \frac{b_j - a}{b_{j+1}} = \frac{\alpha_{j+1}}{\alpha_j}$.

Then Equation 6.5 simplifies to $d_j^2 \left(\frac{\alpha_j \sqrt{5}}{K} \right) - d\alpha_j + \alpha_j \leq 0$. This inequality holds for values of d_j that are in $\left[\frac{K\left(1 - \sqrt{1 - \frac{8\varphi\sqrt{5}}{K}}\right)}{2\varphi}, \frac{K\left(1 + \sqrt{1 - \frac{8\varphi\sqrt{5}}{K}}\right)}{2\varphi} \right]$.

Thus we take $b_{j+1}-a$ to be the largest Fibonacci number that is smaller than $(b_j-a)\frac{K\left(1+\sqrt{1-\frac{8\varphi\sqrt{5}}{K}}\right)}{2\varphi}$ and $\alpha_{j+1} := \frac{K}{2\varphi c_{j+1}}$. Note that for Theorem 3 to hold, it is necessary for α_{j+1} to be less than 0.5. If we compute α_{j+1} to be greater than 0.5, we simply terminate the computation and use α_j and b_s as the starting point.

Algorithm 22: Finding the base set for the recursive normalization-set-finding algorithm

Data: K, a, b_0, α_0, L

Result: Takes in the number of elements K that can be easily stored without loss of performance and the number of L for which brute-force search takes an acceptably small time. Returns the set $G_0 := \{y \in [a,b_0] || \text{err}(y)| \leq \alpha_0\}$.

```
1  j ← 0
2  while b_j − a ≥ L do
3      b_{j+1} ← PrevFibo((b_j − a) K(1+√(1−8φ√5/K))/(2φ)) + a
4      α_{j+1} ← K/(2φ(b_{j+1} − a))
5      if α_{j+1} ≥ 1/2 then
6          break
7      F_{k_{j+1}} ← b_{j+1} − a
8      j ← j + 1
9  G_j ← {}
10 for i = a; i ≤ b_j; i++ do
11     if |err(i)| ≤ α_j then
12         G_j ← G_j ∪ i
13 while j > 0 do
14     j ← j − 1
15     G_j ← NormStep(G_{j+1}, α_{j+1}, α_j, a, b_{j+1}, b_j, F_{k_j})
16 return G_0
```

This can be thus summed up with Algorithm 22. We denote with $\mathsf{PrevFibo}(x)$ the function that returns the largest Fibonacci number that is smaller than or equal to x. We denote with $\mathsf{Normset}()$ the algorithm 21, with the corresponding input. We first recursively compute the b_j, α_j and $\mathbb{F}_{k_j}$ in the way as described above and increase the counter j by one each step. We do this until $b_j - a$ is suitably small to brute-force search it or when α_{j+1} would be greater than $\frac{1}{2}$. We then brute-force

search $[a, b_j]$ for elements y with $|\mathrm{err}(y)| \leq \alpha_j$ which we shall denote with G_j. We then recursively compute all the smaller G_j, using G_{j+1} and the coefficients we have computed before. In the end we obtain

$$G_0 := \{y \in [a, b_0] || \mathrm{err}(y)| \leq \alpha_0\}$$

which is exactly what we were looking for.

We now have the method for obtaining the set $G_0 := \{y \in [a, b_0] || \mathrm{err}(y)| \leq \alpha_0\}$. However, this isn't still enough provide us with a normalization set. We need to first set a suitable $[a, b_0]$ and α_0 for each 2^i, then apply the algorithm 22, and then to pick the right element from each G_0.

One option is to let for each 2^i the interval $[a, b_0]$ to be the interval $[2^i - \ell, 2^i + \ell]$. We can be sure that the elements y with the smallest $|\mathrm{err}(y)|$ will be in these sets.

However, we do not have efficient methods for finding an optimal or near-optimal solution if the pool of possible solutions is large for every 2^i. Note that in essence, this is an instance of integer programming. While it also has some additional structure, we do not know how to use that in order to effectively obtain a near-optimal solution.

Thus, we tried to keep the set of possible approximants small for every 2^i relatively small, choosing only 2 or 3 approximants for every 2^i. After that, we applied some regular integer programming techniques to obtain normalisation sets.

6.1.3. Second Normalization Method

Note that we assumed that we have to normalize after every multiplication. Because normalization is rather costly and introduces inaccuracy, we would prefer to do it more rarely. We could do it more rarely if we cast our representatives to some larger ring, perform several multiplications there and then do only one normalization before converting back. However, because finding good normalization sets is difficult, we can only do it for a smaller number of bigger rings, while it would be preferrable to be able to specify the bigger ring size depending on the number of multiplications we need to perform depending on the application. While it would technically be possible to pre-compute normalization sets for a large number of rings $\mathbb{Z}_{2^n}$, the normalization value ℓ could also depend on the specification, and thus one would have to compute a new normalization set for every new application, which would be too resource-costly. Thus a normalization method where less precomputation is needed would be good.

We now describe such a method for normalization. This is a method that only works for secret-sharing based frameworks, due to it using some properties of those frameworks. While it supposes that the value is secret-shared between two parties, it can be also applied for other cases than 2-party secret-sharing as in other cases, it is possible to use the ReshareToTwo primitive beforehand. The method is based on how secret-sharing computation works. In essence, we note

that when the secret integer representative $[\![a]\!]$ is shared between two parties P_0 and P_1 with P_i holding a_i so that $a_0 + a_1 = a \pmod{2^n}$ then both of them can *normalize their shares locally* and only minor modifications are needed for the method to result in an (approximately) correct result. However, here we need to pay attention on how we interpret a member of $\mathbb{Z}_{2^n}$ as a member of $\mathbb{Z}$. There are, of course, an infinite number of such interpretations, but usually, we use one fixed interpretation and this question can be ignored. However, in this section, we need to use two different interpretations, and thus we will recall the necessary notation for those two interpretations.

Recall the function $z : \mathbb{Z}_{2^n} \to \mathbb{Z}$ that maps an element $c \in \mathbb{Z}_{2^n}$ to the member of the residue class of c that is in the interval $[0, 2^n - 1]$. We also will use the following helpful lemma the proof of which is trivial.

Lemma 11. *Let $c, d \in \mathbb{Z}_{2^n}$. If $z(c) + z(d) \in 2^n$, then $z(c) + z(d) = z(c + d)$. Otherwise, $z(d) + z(d) = z(c + d) + 2^n$.*

Note that if some value e is additively 2-shared as (e_0, e_1), then this lemma can be applied to it, with $c = e_0$ and $d = e_1$. Also recall the function $t : \mathbb{Z}_{2^n} \to \mathbb{Z}$ that maps an element $c \in \mathbb{Z}_{2^n}$ to the member of the residue class of c that is in the interval $[-2^{n-1}, 2^{n-1} - 1]$. Before, we simply used c instead of $t(c)$ but as in this subsection we need to deal with different conventions, we use this notation to avoid confusion. A lemma similar to Lemma 11 holds, the proof of which is similarly trivial.

Lemma 12. *Let $c, d \in \mathbb{Z}_{2^n}$. $t(c) \pm t(d) = t(c \pm d)$ if and only if $t(c) \pm t(d) \in [-2^{n-1}, 2^{n-1}]$.*

The following note follows directly from the definitions of $t(c)$ and $z(c)$

Note 3. Note that if $t(c) \geq 0$, then $t(c) = z(c)$, otherwise $t(c) = z(c) - 2^n$. Likewise, if $z(c) < 2^{n-1}$, then $t(c) = z(c)$, otherwise $t(c) = z(c) - 2^n$.

Now we shall describe the method more precisely. Let there be a golden number $a - \varphi b$ that shall be normalized. We shall apply the $\mathsf{ReshareToTwo}(a)$ primitive to obtain a sharing of a where only the first two parties hold nonzero shares. The value b does not have to be shared between two parties.

Now let both parties hold their shares of a, a_0 and a_1, respectively. The method now specifies threshold values ℓ_i for both parties. The role of these values is similar to ℓ in the previous normalization method — it describes, how much a value should be normalized. More precisely, suppose that we normalize (a, b) and obtain the value (a', b') with the shares of a' being a'_0 and a'_1. We require that $0 \leq t(a'_i) \leq \ell_i$ for both P_0 and P_1 for the number to be correctly normalized.

The values of the ℓ_i depend on the desired specifics of the application. While often it is reasonable for ℓ_0 and ℓ_1 to be equal, sometimes different values can result in a better efficiency. Generally, we here assume that $\ell = \ell_0 + \ell_1$.

Let us only consider the case where $t(a)$ is non-negative. We note that we can apply the same protocol to $-a$ for the negative case and obliviously choose the correct option based on the most significant bit of a. While obtaining the most

significant bit is rather expensive, we have not found a cheaper option. Thus we, from here on, presume that $t(a)$ is non-negative.

Both parties P_i locally find pairs of numbers (x_i, y_i) so that $z(x_i) - \varphi z(y_i) \approx 0$ and that $0 \leq z(a_i) - z(x_i) \leq \ell_i$. How this is done will be explained in more detail later, but essentially, it can be done by deducing Fibonacci numbers until the number obtained is smaller than ℓ_i.

Then they compute $a'_i := a_i - x_i$ and $b'_i := b_i - y_i$ and let the normalized value be $a' = (a'_0, a'_1, 0, \ldots, 0), b' = (b'_0, b'_1, b_2, \ldots, b_M)$. Now we know that the value of $z(a')$ is now no greater than $\ell_0 + \ell_1 = \ell$. However, whether the value of $t(a) - \varphi t(b)$ is approximately equal to the value of $t(a') - \varphi t(b')$ depends on a value that we call the *overflow bit*.

Namely, note that there are two possibilities for the pair (a_0, a_1) — either $z(a_0) + z(a_1) = z(a)$ or $z(a_0) + z(a_1) = z(a) + 2^n$. It is easy to see that other cases are not possible. We shall see that if $z(a_0) + z(a_1) = z(a)$, then $t(a) - \varphi t(b) \approx t(a') - \varphi t(b')$, however, if $z(a_0) + z(a_1) = z(a) + 2^n$, then either $t(a) - \varphi t(b)$ and $t(a') - \varphi t(b')$ differ by about $\frac{2^n}{\varphi}$ or $t(a) - \varphi t(b)$ is already sufficiently normalized. We have access to a primitive that returns us the overflow bit and thus we can use that to obtain the correct result.

We shall see that the question whether $t(a) - \varphi t(b)$ and $t(a') - \varphi t(b')$ are approximately equal is equivalent to the question whether $t(x) - \varphi t(y)$ is approximately zero. This equivalence will be shown by the following three lemmas.

We will use the variables introduced before in this subsection in the context as they have been assumed.

Namely, we will assume that $a, a', b, b', a_0, a_1, a'_0, a'_1, x_0, x_1, y_0, y_1 \in \mathbb{Z}_{2^n}$ and $\ell_0, \ell_1, \ell, \varepsilon_0, \varepsilon_1, \varepsilon \in \mathbb{R}$ and that they satisfy the following properties: $a_0 + a_1 = a$, $\ell_0 + \ell_1 = \ell$, $\varepsilon_i = z(x_i) - \varphi z(y_i)$, $\varepsilon = \varepsilon_0 + \varepsilon_1$, $0 \leq z(a_i) - z(x_i) \leq \ell_i$, $a'_i := a_i - x_i$ and $b'_i := b_i - y_i$ for $i \in \{0, 1\}$ and $a' = a'_0 + a'_1$. We also assume $|\varepsilon_i| << 1$.

We now give three lemmas that state that under certain fairly natural constraints, $t(a') = t(a) - t(x)$ and $t(b') = t(b) - t(y)$.

Lemma 13. *Let the variables have the properties as as described above. Let $t(a) \geq 0$. Then $t(a') = t(a) - t(x)$.*

Proof. We know that $t(a) \in [0, 2^{n-1} - 1]$.

Thus $t(x) \in [0, 2^{n-1} - 1]$. Thus $t(a) - t(x)$ must lay in $[-2^{n-1}, 2^{n-1} - 1]$. The claim follows from Lemma 12. □

Lemma 14. *Let the variables have the properties as as described above. Let $\varphi\ell < 2^{n-1}$.*

Let $t(a) \geq 0$. Let $z(a_0) + z(a_1) < 2^n$.

Then $t(b') = t(b) - t(y)$.

Proof. As $t(a) \in [0, 2^{n-1})$, and $|t(a) - \varphi t(b)| \leq \ell(\varphi - 1)$, we get that

$$t(b) \in \left[\frac{-\ell(\varphi-1)}{\varphi}, \frac{2^{n-1} + \ell(\varphi-1)}{\varphi} \right).$$

From $t(a) \geq 0$, it follows that $z(a) < 2^{n-1}$. Applying Lemma 11, we get that $z(a_0) + z(a_1) < 2^{n-1}$. Thus $0 \leq z(x_0) + z(x_1) < 2^{n-1}$. Thus $z(y_0) + z(y_1) < \frac{2^{n-1}}{\varphi}$. Because $z(y_0) + z(y_1) \in [0, 2^{n-1}]$, $z(y_0) + z(y_1) = z(y) = t(y)$. Thus $t(y) \in \left[0, \frac{2^{n-1}}{\varphi}\right)$

Combining these two estimates, we get that

$$t(b) - t(y) \in \left(\frac{-\ell(\varphi-1) - 2^{n-1}}{\varphi}, \frac{2^{n-1} + \ell(\varphi-1)}{\varphi} \right) = (-2^{n-1}, 2^{n-1}).$$

Now, due to Lemma 12, $t(b-y) = t(b) - t(y)$. As $b' = b - y$, $t(b') = t(b) - t(y)$ and hence the claim is proven. □

Lemma 15. *Let the variables have the properties as as described above. Let* $\ell < 2^{n-1}\varphi + \varepsilon\varphi$. *Let* $t(a) \geq \ell$. *Let* $z(a_0) + z(a_1) \geq 2^n$. *Then* $z(y_0) + z(y_1) \in [2^{n-1}, 2^n)$ *and* $t(b') = t(b) - t(y)$.

Proof. We begin by noting that because $z(a_0) + z(a_1) \in [2^n + \ell, 2^n + 2^{n-1})$ and $0 \leq z(a_i) - z(x_i) \leq \ell_i$, then $z(x_0) + z(x_1) \in [2^n, 2^n + 2^{n-1})$. From this we get that $z(y_0) + z(y_1) \in \left[\frac{2^n}{\varphi}, \frac{2^n + 2^{n-1}}{\varphi} \right] \subset [2^{n-1}, 2^n)$ which is the first part of what we needed to prove.

This also means that $z(y_0) + z(y_1) = z(y) = t(y) + 2^n$, due to Note 3. We also get that $z(x) = z(x_0) + z(x_1) - 2^n$. Note 3 also gives us that $t(a) = z(a)$. Now there exists such a positive v such that $z(y_0) + z(y_1) = \frac{2^n}{\varphi} + v$. Thus $z(x_0) + z(x_1) = \varphi \cdot (z(y_0) + z(y_1)) + \varepsilon_0 + \varepsilon_1 = 2^n + v\varphi + \varepsilon$.

As $0 \leq z(a_i) - z(x_i) \leq \ell_i$, we get that

$$2^n + v\varphi + \varepsilon \leq z(a_0) + z(a_1) \leq z(x_0) + \ell_0 + z(x_1) + \ell_1 \leq \ell + 2^n + v\varphi + \varepsilon$$

Because of this and because of $z(a) = z(a_0) + z(a_1) - 2^n$, we obtain that

$$z(a) \in [v\varphi + \varepsilon, \ell + v\varphi + \varepsilon] \tag{6.6}$$

Because $z(a_0) + z(a_1) \geq 2^n$, $z(a) \in [\ell, 2^{n-1}$, thus $t(a) = z(a)$ and is thus also in $[v\varphi + \varepsilon, \ell + v\varphi + \varepsilon]$.

We know that $|t(a) - \varphi t(b)| \leq \ell(\varphi - 1)$ and hence

$$t(b) \in \left[\frac{t(a) - \ell(\varphi-1)}{\varphi}, \frac{t(a) + \ell(\varphi-1)}{\varphi} \right].$$

Plugging in that $t(a) \in [v\varphi + \varepsilon, \ell + v\varphi + \varepsilon]$, we obtain that

$$t(b) \in \left[v + \frac{\varepsilon}{\varphi} - \frac{\ell}{\varphi^2}, v + \ell + \frac{\varepsilon}{\varphi} \right].$$

Thus

$$\begin{aligned} t(b) - t(y) \ &\in \left[v + \frac{\varepsilon}{\varphi} - \frac{\ell}{\varphi^2} - (\frac{2^n}{\varphi} + v - 2^n), v + \ell + \frac{\varepsilon}{\varphi} - (\frac{2^n}{\varphi} + v - 2^n) \right] \\ &= \left[-\frac{\ell}{\varphi^2} - \frac{2^n}{\varphi} + 2^n + \frac{\varepsilon}{\varphi}, \ell - \frac{2^n}{\varphi} + 2^n + \frac{\varepsilon}{\varphi} \right] \\ &= \left[\frac{2^n - \ell}{\varphi^2} + \frac{\varepsilon}{\varphi}, \frac{2^n}{\varphi^2} + \ell + \frac{\varepsilon}{\varphi} \right] \\ &\subset (0, 2^{n-1}) \end{aligned} \tag{6.7}$$

Per Lemma 12, $t(b) - t(y) = t(b - y) = t(b')$. Hence the claim is proven.

□

Now, we combine these lemmas and obtain the following proposition.

Proposition 1. *Let the variables have the properties as as described above. Let $\varphi\ell < 2^{n-1}$. Let $t(a) \geq \ell$. If $z(a_0) + z(a_1) < 2^n$, then $t(a) - \varphi t(b) = t(a') - \varphi t(b') + \varepsilon$. If $z(a_0) + z(a_1) \geq 2^n$, then $t(a) - \varphi t(b) = t(a') - \varphi t(b') + \frac{2^n}{\varphi} + \varepsilon$.*

This proposition gives us a strategy for normalization. Namely, if $0 \leq a \leq \ell$, then we do not want to normalize. If $z(x_0) + z(x_1) < 2^n$, then parties P_0 and P_1 behave as described before — they simply replace their respective shares a_i and b_i with a_i' and b_i' respectively. If $z(x_0) + z(x_1) \geq 2^n$, then they should deduce a golden number with a value equal to $\frac{2^n}{\varphi}$. Naturally, they have to perform all three cases and then use corresponding bits to obliviously choose between these options. This will be formalized later. Now we will give the proof for the proposition.

Proof. Per the lemmas that were just proven, $t(a') = t(a) - t(x)$ and $t(b') = t(b) - t(y)$. (Note that if $\varphi\ell < 2^{n-1}$, then also $\ell < 2^{n-1}\varphi + \varepsilon\varphi$ and thus the prerequisite is satisfied also for Lemma 15.) Thus $t(a) - \varphi t(b) - t(a') - \varphi t(b') = t(x) - \varphi t(y)$.

Now, consider the case when $z(a_0) + z(a_1) < 2^n$. We note that because $0 \leq z(x_i) \leq z(a_i)$ then $0 \leq z(x_0) + z(x_1) \leq z(a_0) + z(a_1) < 2^{n-1}$ and thus $z(x_0) + z(x_1) = z(x) = t(x)$. Clearly $\varepsilon < \varphi$, thus also $z(y_i) = \frac{z(x_i)}{\varphi} - \frac{\varepsilon_i}{\varphi} > -1$ and because the $z(y_i)$ have to be integers, we get that $z(y_0), z(y_1) \in [0, 2^{n-1})$ and thus $z(y_0) + z(y_1) = z(y) = t(y)$. Now

$$t(x) - \varphi t(y) = z(x_0) + z(x_1) - \varphi(z(y_0) + z(y_1)) = \varepsilon_0 + \varepsilon_1 = \varepsilon.$$

Thus, if $z(a_0) + z(a_1) < 2^n$, $t(a) - \varphi t(b) = t(a') - \varphi t(b') + \varepsilon$.

Now, consider the case when $z(a_0)+z(a_1) \geq 2^n$. Note that in this case also $z(a_0)+z(a_1) \geq 2^n + \ell$. As $0 \leq z(a_i) - z(x_i) \leq \ell_i$,

$$2^n \leq z(a_0)+z(a_1)-\ell_0-\ell_1 \leq z(x_0)+z(x_1) \leq z(a_0)+z(a_1) < 2^n + 2^{n-1}.$$

Thus $z(x_0)+z(x_1) \in [2^n, 2^n+2^{n-1}]$ and hence $z(x) = z(x_0)+z(x_1)-2^n$. Thus $z(x) \in [0, 2^{n-1})$ and hence $t(a) = z(a)$. Thus, $t(x) = z(x_0)+z(x_1)-2^n$.

On the other hand, Lemma 15 also gave us that $z(y_0)+z(y_1) \in [2^{n-1}, 2^n)$. Thus $z(y) = z(y_0)+z(y_1)$ and $t(y) = z(y) - 2^n = z(y_0)+z(y_1)-2^n$. Combining those we obtain that, if $z(x_0)+z(x_1) \geq 2^n$, then

$$t(x) - \varphi t(y) = z(x_0)+z(x_1)-2^n - \varphi \cdot (z(y_0)+z(y_1)-2^n) = \frac{2^n}{\varphi} + \varepsilon.$$

Thus the claim is proven.

□

Now let us consider the two more questions that must be solved for the argument to work — how exactly do the parties find the pairs (x_i, y_i) and how do we safely deduce $\dfrac{2^n}{\varphi}$ from the result.

We shall explain now how to find the numbers (x_i, y_i) with the desired properties. Note that while this can be done locally and thus we can more easily perform operations that would be expensive otherwise, we still should have a reasonably efficient algorithm for that. Here we use the fact that the number $F_n - \varphi F_{n-1}$ is very small, more precisely, it is equal to $(-\varphi)^{-k}$, as shown in 6.2. Thus, the party P_i starts deducing as large Fibonacci numbers from a_i as possible. This is formalized in Algorithm 23.

We choose the first $\mathscr{F}_k$ to be such that $\mathscr{F}_k \leq a < \mathscr{F}_{k+1}$. A simple calculation shows that this is equivalent to $k = \lfloor \log_\varphi a + \log_\varphi \sqrt{5} \rfloor$. We then start trying to deduce smaller Fibonacci numbers until a is no greater than ℓ and remember the sum of the Fibonacci numbers that we deduce. Note that if parties locally perform this operation as has been described until now, it opens up a timing attack. Namely, for some values of a logarithmically many comparisons and subtractions need to be made whereas for others less are needed (e.g in the case when $a < \ell$). Thus this problem should be corrected for in that every call of the algorithm would take the same amount of time. Since this requires only a small amount of very simple arithmetic, the time can be upper-bounded by some small amount of time p. Thus

the algorithm returns the result only when the time p is over.

Algorithm 23: LocalFiboApproximator

Data: Gets in a positive integer a and a positive parameter ℓ and a timing parameter p.

Result: Finds (x,y) so that $0 \leq a-x \leq \ell$ and $x-\varphi y \approx 0$

```
k ← ⌊log_φ a + log_φ √5⌋
(x,y) ← (0,0)
while a > ℓ do
    if a ≥ 𝓕_k then
        a ← a − 𝓕_k
        x ← x + 𝓕_k
        y ← y + F_{k−1}
    k ← k − 1
delayuntil(p)
return x, y
```

Now let us consider how to deduce $\frac{2^n}{\varphi}$. We note that this is a value much larger than we usually allow for (i.e larger than $\ell(\varphi-1)$) and thus we should be cautious about which of the claims we have proven we can use.

We thus need a pair of integers (c,d) with the following properties: if the variables are defined as above and $z(a_0)+z(a_1) \geq 2^n$, then $t(a'+c)-\varphi t(b'+d) \approx t(a)-\varphi t(b)$ and $t(a'+c) \in [-\ell,\ell]$.

Thus we need that $t(c) \in [-\ell,0]$. We will show that if $t(c) \in [-\ell,0]$ and $t(c)-\varphi t(d) \approx -\frac{2^n}{\varphi}$, then the pair (c,d) has the desired properties. We measure how far $t(c)-\varphi t(d)$ is from $-\frac{2^n}{\varphi}$ with $\bar{\varepsilon}$. While the proposition requires only that $\ell \pm \frac{\varepsilon-\bar{\varepsilon}}{\varphi} < 2^{n-1}$ holds, (which is true for any practical values of ℓ, ε and $\bar{\varepsilon}$, as $\ell << 2^{n-1}$ and $\varepsilon, \bar{\varepsilon} << 1$) the $\bar{\varepsilon}$ introduces an additional error term and thus it is desirable to keep it small.

Proposition 2. *Let the variables be defined as in above. Let $c,d \in \mathbb{Z}_{2^n}$ be such that $t(c) \in [-\ell,0]$ and $t(c)-\varphi t(d) = \frac{2^n}{\varphi} + \bar{\varepsilon}$ where is such that $\ell \pm \frac{\varepsilon-\bar{\varepsilon}}{\varphi} < 2^{n-1}$ holds.Let $\varphi\ell < 2^{n-1}$. Let $t(a) \geq \ell$.*

Let $z(a_0)+z(a_1) \geq 2^n$.

Then $t(a'+c) \in [-\ell,\ell]$ and $t(a'+c)-\varphi t(b'+d) = t(a)-\varphi t(b)+\bar{\varepsilon}-\varepsilon$.

Proof. Note that as $t(c) \in [-\ell,0]$ and $t(a') \in [0,\ell]$, thus $t(a')+t(c) \in [-\ell,\ell] \subset [-2^{n-1},2^{n-1})$ and thus $t(a'+c) = t(a')+t(c) \in [-\ell,\ell]$, thus obtaining the first thing we needed to prove.

For the second, we note that if $t(a'+c) = t(a')+t(c)$ and $t(b'+d) = t(b')+$

$t(d)$ then

$$\begin{aligned} t(a'+c) - \varphi t(b'+d) &= t(a') + t(c) - \varphi(t(b') + t(d)) \\ &= t(a) - \varphi t(b) - \frac{2^n}{\varphi} - \varepsilon + \frac{2^n}{\varphi} + \bar{\varepsilon} \\ &= t(a) - \varphi t(b) + \bar{\varepsilon} - \varepsilon \end{aligned} \tag{6.8}$$

Thus we need to show that $t(a')+t(c), t(b')+t(d) \in [-2^{n-1}, 2^{n-1})$ for the second part.

We already showed that $t(a')+t(c) \in [-2^{n-1}, 2^{n-1})$ above. Now, we know that

$$t(d) = \frac{t(c)}{\varphi} - \frac{2^n}{\varphi^2} - \frac{\bar{\varepsilon}}{\varphi} \in \left[\frac{-2^n}{\varphi^2} - \frac{\ell}{\varphi} - \frac{\bar{\varepsilon}}{\varphi}, \frac{-2^n}{\varphi^2} - \frac{\bar{\varepsilon}}{\varphi} \right].$$

For $t(b')$ we know that $t(b') = t(b) - t(y)$ and Equation 6.7 gives us that

$$t(b) - t(y) \in \left[\frac{2^n - \ell}{\varphi^2} + \frac{\varepsilon}{\varphi}, \frac{2^n}{\varphi^2} + \ell + \frac{\varepsilon}{\varphi} \right].$$

Thus we get that $t(b')+t(d) \in [-\ell + \frac{\varepsilon - \bar{\varepsilon}}{\varphi}, \ell + \frac{\varepsilon - \bar{\varepsilon}}{\varphi}] \subset [-2^{n-1}, 2^{n-1}]$ and hence $t(b'+d) = t(b')+t(d)$. Hence the claim is proven.

□

Algorithm 24: GoldenNorm2Pos

Data: Gets in a golden section number $([\![a]\!],[\![b]\!])$, integers c,d such as described in Proposition 2 and bounding coefficients , ℓ_0 and ℓ_1. Also gets in the timing parameter p. We assume $a \geq \ell$ where $\ell = \ell_0 + \ell_1$.

Result: Returns the golden number $[\![a']\!],[\![b']\!]$ so that $a - \varphi b \approx a' - \varphi b'$.

$[\![a]\!] \leftarrow \mathsf{ReshareToTwo}([\![a]\!])$
for $i = 0,1$ **do**
  $(x_i, y_i) \leftarrow \mathsf{Fibo}(a_i, \ell_i, p)$
  $a_i^{'} \leftarrow a_i - x_i$
  $b_i^{'} \leftarrow b_i - x_i$
$[\![o]\!] \leftarrow \mathsf{Overflow}([\![a']\!])$
$[\![a']\!] \leftarrow [\![a']\!] + [\![o]\!] \cdot c$
$[\![b']\!] \leftarrow [\![b']\!] + [\![o]\!] \cdot d$
return $[\![a']\!],[\![b']\!]$

Algorithm 25: GoldenNorm2

Data: $[\![a]\!],[\![b]\!],c,d,\ell_0,\ell_1,k$

Result: Given a golden section number and a normalization pair, returns the number normalized according to the pair.

begin in parallel
  $([\![a^{'}]\!],[\![b^{'}]\!]) \leftarrow \mathsf{Posnorm}([\![a]\!],[\![b]\!],c,d,\ell_0,\ell_1)$
  $(-[\![a^{''}]\!],-[\![b^{''}]\!]) \leftarrow \mathsf{Posnorm}([\![-a]\!],[\![-b]\!],c,d,\ell_0,\ell_1)$
$\ell \leftarrow \ell_0 + \ell_1$
begin in parallel
  $[\![f_0]\!] \leftarrow \mathsf{LTEProtocol}([\![a]\!],[\![\ell]\!])$
  $[\![f_1]\!] \leftarrow \mathsf{LTEProtocol}([\![-\ell]\!],[\![a]\!])$
$([\![a]\!],[\![b]\!]) \leftarrow \mathsf{ObliviousChoiceProtocol}([\![f_0]\!],([\![a]\!],[\![b]\!]),([\![a^{'}]\!],[\![b^{'}]\!]))$
$([\![a]\!],[\![b]\!]) \leftarrow \mathsf{ObliviousChoiceProtocol}([\![f_0]\!],([\![a^{''}]\!],[\![b^{''}]\!]),([\![a]\!],[\![b]\!]))$
return $[\![a]\!],[\![b]\!]$

The algorithm is formalized in Algorithms 24 and 25. We shall refer to Algorithm 24 as $\mathsf{Posnorm}([\![a]\!],[\![b]\!],c,d,\ell_0,\ell_1)$. It intakes a golden section number $a - \varphi b$ and a pair of integers c and d that satisfies the conditions described in the Proposition 2 and also ℓ_0 and ℓ_1. We assume $a \geq \ell$.

We first reshare $[\![a]\!]$ to two parties in the sense that only the first two parties hold nonzero shares. Then parties P_0 and P_1 call out Algorithm 23 and obtain pairs (x_i, y_i) and (a_i', b_i') with the properties described above. We then use $\mathsf{Overflow}([\![a']\!])$ to obtain the overflow bit of $[\![a']\!]$ — i.e the bit that describes whether $z(a_0^{'}) + z(a_1^{'}) < 2^n$ or not.

After that we use the overflow bit to deduce (c,d) if necessary. If $a \geq \ell$, then this gives us the correct result. If $-\ell < a < \ell$, then we do not have to normalize.

Thus, we need to only deal with the case where $a \leq -\ell$. In that case, however, it is easy to see that $-a \geq \ell$ and thus we can instead normalize $(-a,-b)$. This is summed up in Algorithm 25.

We now apply the Posnorm twice and compute $\mathsf{Posnorm}([\![a]\!],[\![b]\!],c,d,\ell_0,\ell_1)$ and $-\mathsf{Posnorm}([\![-a]\!],[\![-b]\!],c,d,\ell_0,\ell_1)$ in parallel. After that we compare $[\![a]\!]$ to ℓ and $-\ell$. Based on that result, we obliviously pick either one of those or leave the pair $([\![a]\!],[\![b]\!])$ unchanged.

6.1.4. A Variation on The Second Method

Note that while the second method greatly reduced the need to find specific constants, we still have to find the pair (c,d) that depends on both 2^n and the bound ℓ. We propose a slight modification of the second method where no constants would need to be precomputed. Namely, instead of the ring $\mathbb{Z}_{2^n}$ we will use the ring $\mathbb{Z}_{\mathscr{F}_k}$. Most of the things would be the same, if we only replace 2^n with $\mathscr{F}_k$ everywhere.

Thus, instead of the equation 6.9 we will have

$$\begin{aligned} &x_0 + x_1 - \mathscr{F}_k - \varphi(y_0 + y_1 - \mathscr{F}_k) \\ &\quad = x_0 - \varphi y_0 + x_1 - \varphi y_1 - (\mathscr{F}_k - \varphi\mathscr{F}_k) \\ &\quad \approx -(\mathscr{F}_k - \varphi\mathscr{F}_k) \\ &\quad \approx F_{k-1}. \end{aligned} \tag{6.9}$$

Thus we can take $(-F_{k-1},0)$ for (c,d), computing of which takes practically no time.

6.2. Why φ?

We note that the golden numbers are essentially built on the equality $\varphi^2 = \varphi + 1$ which allows us to reduce the product of

$$(a_0 - \varphi b_0)\cdot(a_1 - \varphi b_1) = a_0a_1 - \varphi(a_0b_1 + b_0a_1) + \varphi^2 b_0b_1$$

to

$$(a_0a_1 + b_0b_1) - \varphi(a_0b_1 + b_0a_1 - b_0b_1).$$

However, any irrational number γ that satisfied an equality $\gamma^2 = u\gamma + v$ where u and v are small integers would allow us to perform an analogous reduction. Thus we could build a similar number scheme out of such real numbers γ. The aim of this section is to discuss this possibility and why we chose φ specifically.

First note that technically, we are not even bound to require that γ must satisfy $\gamma^2 = u\gamma + v$ for some $u,v \in \mathbb{Z}$ for the multiplication scheme to work. γ could also, for example, satisfy some $\gamma^3 = u\gamma^2 + v\gamma + w$ or even another polynomial with a higher degree.

However, in that case we would have to deal with not only the integer representative and the γ-representative, but also the γ^2-representative, and, depending on

the degree of the polynomial, possibly with even more representatives. The problem with this is that with three or more representatives, it would be more difficult to obtain a result similar to Lemma 3 — suppose that we are dealing with such a situation with three representatives a, b and c. Now, even if the absolute value of one of these three is bounded, and the value of the real number it represents is also bounded, it is possible that the other two can have values with arbitrarily large absolute values. For example, if we are talking about numbers in the format $a+b\gamma+c\gamma^2$, then we could have $a=0$ and $a+b\gamma+c\gamma^2 \approx 0$, but also $b \approx -c\gamma$, with b being as large as the number format permits.

Thus, when we would want to normalize such numbers, we would have to normalize at least two of the representatives simultaneously. While this is possible, it is more resource-consuming.

Let us for now consider only the solutions to equations $x^2 = ux+v$ with u and v being small integers. There are several things we would want out of such solutions γ. Let us for now denote with $\gamma_{u,v,0}$ and $\gamma_{u,v,1}$ the solutions to the equation $x^2 = ux+v$. Let $\gamma_{u,v,\cdot}$ refer to both of them.

- It is well known that a solution to a quadratic equation with integer coefficients such an equation must be either an integer, an irrational real number, or a complex number with a nonzero imaginary part. While it is perhaps possible to depict complex numbers using a number system similar to this, it is not our current goal. On the other hand, if the $\gamma_{u,v,\cdot}$ are integers, then our number system is not able to represent non-integers at all, making it useless. Thus, the $\gamma_{u,v,\cdot}$ must be irrational real numbers.
- There should be many good rational approximations to the $\gamma_{u,v,\cdot}$. The error of normalization directly depends on how good the approximations that make can be. In this aspect, φ does not perform well. It is known [42] to be the worst irrational number to rationally approximate.
- In addition to good rational approximations existing, it should not be too difficult to compute them. In the case of φ, Fibonacci numbers have proved useful in several aspects, such as finding normalization sets and proving statements about them.

  Thus it would be useful if the number γ had some similar integers with similar properties to the Fibonacci numbers.

  Pisot numbers are defined to be positive numbers greater than 1 whose all conjugate elements are smaller than 1. They have the property that their powers are almost integers [57]. This property makes it easy to compute some rational approximations and proves useful in other places. φ is a Pisot number. Other Pisot numbers, such as the plastic constant $1.32472\ldots$ might prove useful in this aspect.
- The larger u and v are, the larger multiplication will make the representatives. Note that the multiplication becomes

$$(a_0+\gamma b_0)(a_1+\gamma b_1) = a_0a_1 + \gamma_{u,v}(a_0b_1+b_0a_1) + \gamma_{u,v}^2 b_0b_1 \\ = a_0a_1 + ub_0b_1 + \gamma_{u,v}(a_0b_1 + b_0a_1 \\ + vb_0b_1). \tag{6.10}$$

If u and v are too large, then the multiplication amplifies the results too much and the respective value of ℓ must be significantly smaller, thus reducing other good properties that a number system might otherwise have. In the case of φ, the constants u and v are small.

- We want the set $\{a+b\gamma_{u,v} | a,b \in \mathbb{Z}, |a|, |b| \leq M\}$ to be as equidistributed as possible to obtain good granularity. Measuring this, however, turns out to be nontrivial. We decided to measure it by measuring the largest difference between two consecutive γ-section numbers in $[0,1)$. To measure this we introduce the function $b(\gamma,n)$.

  Definition 6. Let γ be irrational and n be a positive integer. Let the numbers $a_i := i \cdot \gamma \pmod 1$ with $0 \leq i \leq n$. Let $\{a'_i\}_{i=0}^n$ be the permutation of the set $\{a_i\}_{i=0}^n$ that is in the ascending order. (that is, $a'_i < a'_j$ if and only if $i < j$) We define the function $b(\gamma,n)$ as

$$b(\gamma,n) = \max_{0<i'\leq n}(\{a'_i - a'_{i-1}\}) \tag{6.11}$$

  This value seems like a natural measurement for granularity. We took the idea for using such a measurement from [37] however, we were not able to reproduce the results given there.

  The problem with this measurement and also other types of measurements we tried is that it is very dependent on the value n in a quite discontinuous way. The maximum distance stays constant until another point is inserted into the largest interval at which point another interval becomes the largest one. It is essentially a decreasing step function. Thus one might want to characterize γ by observing the average length of the steps and by how much on average $b(\gamma,n)$ decreases at every step. Concerning how the number of the intervals goes up, it is rather natural that it takes longer to hit the largest interval. It is desirable that the length of the steps would increase slowly and that $b(\gamma,n)$ would decrease quickly.

  We ran tests for several $\gamma_{u,v,\cdot}$ with small u,v and observed the following interesting pattern. The two parametres described seem strongly linked — their product appears to be very close to 1 for every number we tested. In this way, no number we tested appears to be better than the others. Notably, for φ the step lengths increased most, and the $b(\gamma,n)$ decreased quickest. We also proved the following lemma to note that behaviour of partial quotient denominators bounds granularity.

  Lemma 16. *Let q_1 and q_2 be two partial quotient denominator of α so that $q_1 < q_2$.Then for any $q_1 \leq p \leq q_2$, it holds that* $\dfrac{1}{q_2} \leq b(\alpha,p) \leq \dfrac{2}{q_1}$.

Proof. Let $q_1 \leq p \leq q_2$. It is clear that if we place q_1 points on $[0,1)$, then there must be two consecutive points with difference $\frac{1}{q_1}$.
Namely, note that if $r_0, r_1 \ldots, r_{q_1}$ are the ordered points in $[0,1)$, then

$$1 = (r_1 - r_0) + (r_2 - 2_1) + \cdots + r_{q_1} - r_{q_1-1} + r_0 - r_{q_1} \pmod 1 \leq q_1 b().$$

By Lemma 9, every interval $\left[\frac{i}{q_1}, \frac{i+1}{q_1}\right)$ contains precisely one of the points $(i \cdot \alpha)_{k=0}^{q_2-1}$. Thus the difference between two consecutive points can be at most $\frac{2}{q_1}$ □

6.3. Protocols for Golden Section Numbers

We shall now describe a few protocols for golden section numbers. These give ideas how the golden section numbers can be used. We have already described addition, multiplication, and normalization protocols, and thus we will not discuss them any further here.

For normalization, we do not specify everywhere which normalization method exactly we use. Generally, the first method gives us better precision while the second one is faster and more flexible as less precomputation is necessary to find constants needed for normalization. Thus, for example, if we need to perform several consecutive multiplications, we can find such a ring $\mathbb{Z}_K$ so that we can be sure that overflow would not happen, convert the multiplicands to $\mathbb{Z}_K$ and perform the multiplications there. Then, we could find normalization constants for $\mathbb{Z}_K$, normalize the value there and convert back to the original ring. We assume four-field signed fixed-point numbers in this part.

It is somewhat complicated to compare the efficiency of multiplication to, for example, fixed-point or floating-point multiplication. The multiplication itself is a fast operation, requiring only four parallel integer multiplications and some summation, however, after a number of multiplications, we need to normalize the value.

To obtain some standard approach, we shall assume that all the inputs that we get to a multiplication protocol are normalized and that we have to normalize the output.

6.3.1. Multiplication by φ

We will start by describing a sub-protocol that we will often use. We will describe now a protocol for multiplying an integer by the golden ratio. This protocol, presented in Algorithm 26, will be useful for performing golden-to-fix conversion

described in Section 10.

Algorithm 26: MultWithPhi

Data: Gets in a secure signed integer $[\![x]\!]$, with n bits and the fixed-point parameter m (where $m > n$), bits of φ denoted with $\{p_i\}_{i=0}^{\infty}$

Result: Returns the fixed-point number $[\![x\varphi]\!]$ where the underlying integer type has $n+m$ bits and the radix point is m.

1 $\{[\![x_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}([\![x]\!])$

2 $[\![s_0]\!] \leftarrow \sum_{i=0}^{m} p_i \cdot (\sum_{j=0}^{n-2} [\![x_j]\!] \cdot 2^{m+j-i})$

3 $[\![s_1]\!] \leftarrow \sum_{i=m+1}^{m+n} p_i \cdot (\sum_{j=i-m}^{n-2} [\![x_j]\!] \cdot 2^{m+j-i})$

4 $[\![s]\!] \leftarrow [\![s_0]\!] + [\![s_1]\!]$

5 $\{[\![x'_i]\!]\}_{i=0}^{n-1} \leftarrow \mathsf{BitExtract}([\![-x]\!])$

6 $[\![s'_0]\!] \leftarrow \sum_{i=0}^{m} p_i \cdot (\sum_{j=0}^{n-2} [\![x'_j]\!] \cdot 2^{m+j-i})$

7 $[\![s'_1]\!] \leftarrow \sum_{i=m+1}^{m+n} p_i \cdot (\sum_{j=i-m}^{n-2} [\![x'_j]\!] \cdot 2^{m+j-i})$

8 $[\![s']\!] \leftarrow [\![s'_0]\!] + [\![s'_1]\!]$

9 $[\![r]\!] \leftarrow \mathsf{ObliviousChoice}([\![x_{n-1}]\!], [\![s']\!], [\![s]\!])$

10 **return** $([\![x_{n-1}]\!], [\![r]\!])$

The protocol takes in a secret n-bit signed integer $[\![x]\!]$ and returns a signed fixed-point number $[\![^f x\varphi]\!]$ where the underlying integer type has $m+n$ bits and the radix-point is m. This protocol needs one bit-extraction protocol and one oblivious choice. We start with a secret signed integer $[\![x]\!]$. We also have the bits of φ which are denoted here by $\{p_i\}_{i=0}^{\infty}$ with $\varphi = \sum_{i=0}^{\infty} p_i \cdot 2^{-i}$. We begin by extracting the bits $[\![x_i]\!]$ from the input $[\![x]\!]$ on line 1. For now, suppose that x is non-negative and thus the most significant bit is zero. We want to compute the representative of the product of φ and $[\![x]\!]$, thus we would like ideally to compute $2^m(\sum_{i=0}^{-\infty} p_i 2^{-i})(\sum_{j=0}^{n-2} [\![x_j]\!] 2^j) = \sum_{i=0}^{\infty} p_i (\sum_{j=0}^{n-2} [\![x_j]\!] 2^{m+j-i})$. We operate on integers and thus compute instead $\sum_{i=0}^{\infty} p_i (\sum_{j=0}^{n-2} [\![x_j]\!] \lfloor 2^{m+j-i} \rfloor)$. Note that if $i > m+n$, then

$$\sum_{j=0}^{n-2} ([\![x_j]\!] \lfloor 2^{m+j-i} \rfloor) = 0,$$

if $m+1 \leq i \leq m+n$, then

$$\sum_{j=0}^{n-2} ([\![x_j]\!] \lfloor 2^{m+j-i} \rfloor) = \sum_{j=i-m}^{n-2} [\![x_j]\!] 2^{m+j-i},$$

and if $0 \leq i \leq m$, then

$$\sum_{j=0}^{n-2}([\![x_j]\!]\lfloor 2^{m+j-i}\rfloor) = \sum_{j=0}^{n-2}[\![x_j]\!]2^{m+j-i}.$$

All of these can be computed relatively easily.

We thus, on lines 2 to 4 compute $\sum_{i=0}^{m} p_i \cdot \left(\sum_{j=0}^{n-2}[\![x_j]\!]\cdot 2^{m+j-i}\right) + \sum_{i=m+1}^{m+n} p_i \cdot \left(\sum_{j=i-m}^{n-2}[\![x_j]\!]\cdot 2^{m+j-i}\right)$ that represents $x\varphi$ if x is positive. We then do the same for $-x$ on lines 5 to 8 and obliviously choose between the two cases based on the most significant bit of x on line 9. The most significant bit of x is also the sign of the resulting fixed-point number, as multiplication with φ does not change the sign.

Conversion to a Fixed-Point Number. Algorithm 27 presents the protocol for converting a golden section number to a fixed-point number.

Algorithm 27: GoldToFix

Data: Gets in a secret golden number $([\![a]\!],[\![b]\!])$ where the underlying integers are n-bit integers and the fixed-point parameter m (where $n > m$)

Result: Returns a fixed-point number that represents the same value as the golden number input $[\![a]\!] - \varphi[\![b]\!]$. The underlying integer of the fixed-point number has $n+m$ bits and its radix point is m.

1 $[\![A]\!] \leftarrow \mathsf{ConvertUp}([\![a]\!], n, n+m)$
`// we will also obtain` a_{n-1} `as a side product from the ConvertUp function.`
2 $[\![^f A]\!] \leftarrow ([\![a_{n-1}]\!], [\![a]\!]\cdot 2^m)$
3 $[\![^f B]\!] \leftarrow \mathsf{MultWithPhi}([\![b]\!], n, m)$
4 $[\![^f C]\!] \leftarrow \mathsf{FixSubtract}([\![^f A]\!], [\![^f B]\!])$
5 **return** $[\![^f C]\!]$

While conversion functions are important on their own, here we will also use them as subprotocols in more complicated algorithms. Since we have access to MultWithPhi function, converting a golden number to a fixed-point number is trivial. We need to convert both the integer representative and the φ-representative to a respective fixed-point number and deduce the second from the first.

Return a Constant Based on the Floor of The Logarithm. We will see that in both the inverse protocol and the square root protocol, we get a secret golden number $[\![^g x]\!]$ and, based on the interval $[2^i, 2^{i+1})$ its absolute value is in, return a golden number $[\![^g z_i]\!]$.

The protocol for performing this operation is presented in Algorithm 28.

Algorithm 28: TwoPowerConst

Data: $[\![^g x]\!], \{^g z_i\} = \{(x_i, y_i)\}, n, m < n$

Result: Will return the sign of the input and (x_j, y_j) if $|x| \in [2^j, 2^{j+1})$.

1 $[\![sign]\!], [\![f]\!] \leftarrow \mathsf{GoldToFix}([\![^g x]\!])$

2 $\{[\![b_i]\!]\}_{i=0}^{n+m-1} \leftarrow \mathsf{MSNZB}([\![f]\!])$

3 $([\![s]\!], [\![t]\!]) \leftarrow \sum_{i=-m}^{n-1} ([\![b_i]\!] \cdot x_i, [\![b_i]\!] \cdot y_i)$

4 **return** $[\![sign]\!], ([\![s]\!], [\![t]\!])$

The computation is performed the following way. We convert the input to a fixed-point number. Note that the interval $[2^j, 2^{j+1})$ where the absolute value is described by the most significant nonzero bit of the representative of the fixed-point number. We thus then perform MSNZB on the integer representative of the fixed-point number and compute the scalar product with the set of public coefficients $\{^g z_i\}$, thus obtaining (x_j, y_j).

Inverse. We shall now describe the protocol for computing the inverse of a secret golden number $[\![^g x]\!]$. A protocol for computing the inverse of numbers in $[0.5, 1]$ is presented in Algorithm 29. It uses the approximation

$$\frac{1}{x} = \prod \left((1-x)^{2^i} + 1 \right)$$

that works well in the neighbourhood of 1 (being equivalent to the respective Taylor series). We refer to it as HalfToOneInv .

Algorithm 29: HalfToOneInv

Data: Gets in a golden section number $[\![^g x]\!]$ where $(x \in [0.5, 1])$ and integers n, m, and k where $(n > m)$-

Result: Returns the golden section number $[\![^g \frac{1}{x}]\!]$ the value of which is approximately equal to $\frac{1}{x}$.

1 $[\![^g y]\!] \leftarrow 1 - [\![^g x]\!]$

2 $[\![^g y_0]\!] \leftarrow [\![^g y]\!]$

3 **for** $i \leftarrow 0$ **to** $k-1$ **do**

4 $\quad [\![^g y_{i+1}]\!] \leftarrow \mathsf{GoldenMult}([\![^g y_i]\!], [\![^g y_i]\!])$

5 $[\![^g z]\!] \leftarrow \mathsf{GoldenProd}([\![^g y_0]\!] + 1, [\![^g y_1]\!] + 1, \ldots, [\![^g y_k]\!] + 1)$

6 **return** $[\![^g z]\!]$

The protocol for computing the inverse of a golden number is presented in

Algorithm 30.

Algorithm 30: GoldInv

Data: Gets in the secure golden section number $[\![^g x]\!]$ integers n and m (with $n > m$), and constants $\{(x_i, y_i)\}$.

Result: Returns the golden section number $[\![^g \frac{1}{x}]\!]$ the value of which is approximately equal to $\frac{1}{x}$.

1 $([\![sign]\!], [\![^g y]\!]) \leftarrow \mathsf{TwoPowerConst}([\![^g x]\!], \{^g z_i\})$
2 $[\![^g x']\!] \leftarrow \mathsf{GoldenMult}([\![^g x]\!], [\![^g y]\!])$
3 $[\![^g z]\!] \leftarrow \mathsf{HalfToOneInv}([\![^g x']\!])$
4 $[\![^g w]\!] \leftarrow \mathsf{GoldenMult}([\![^g y]\!], [\![^g z]\!])$
5 $[\![^g u]\!] \leftarrow \mathsf{ObliviousChoice}([\![sign]\!], -[\![^g w]\!], [\![^g w]\!])$
6 **return** $[\![^g u]\!]$

We use Algorithm 29 as a subprotocol. Given ${}^g x$ as an input, we need to find ${}^g x'$ and ${}^g y$ so that $x' \in [0.5, 1]$ and that $x \cdot y = x'$. We can then use the HalfToOneInv function to compute $\frac{1}{x'} = \frac{1}{x} \cdot \frac{1}{y}$ which we shall then multiply with y to obtain $\frac{1}{x}$. We note that if $|x| \in [2^j, 2^{j+1})$, then $|x| \cdot 2^{-j-1}$ is in $[0.5, 1]$ which would thus be a suitable value for $|x'|$ — hence y should be 2^{j-1}.

We compute y using the function $\mathsf{TwoPowerConst}([\![^g x]\!], \{^g z_i\})$. Here the ${}^g z_i$ are approximations of different powers of 2 – when $x \in [2^j, 2^{j+1})$, then TwoPowerConst should return approximately 2^{-j-1}.

Thus we obtain the following algorithm. We first compute $[\![^g x']\!]$ by multiplying $[\![^g x]\!]$ and $[\![^g y]\!]$. We then use the HalfToOneInv protocol on $[\![^g x']\!]$, obtaining $[\![^g \frac{1}{x'}]\!]$. To get this back to the correct range, we multiply it by $[\![^g y]\!]$.

Finally, since our current result is approximately $[\![^g \left|\frac{1}{x'}\right|]\!]$, we have to make an oblivious choice between the result and its additive inverse so that it would have the correct sign.

Square Root Protocol. Algorithm 31 presents the protocol for finding the square root of a golden section number.

Algorithm 31: GoldSqrt

Data: Gets in a secret golden section number $[\![^g x]\!]$. The underlying integer type has n bits. Uses fixed-point numbers with $n+m$ bits and radix point m as an ancillary data type. Does k rounds and uses the constants $\{^g w_i\}$.

Result: Returns the secret golden section number $[\![^g \sqrt{x}]\!]$ the value of which is approximately equal to $\sqrt{x}$.

1 $[\![^g y_0]\!] \leftarrow \mathsf{TwoPowerConst}([\![^g x]\!], \{^g w_i\})$
2 **for** $i \leftarrow 0$ **to** $k-1$ **do**
3 $\quad [\![^g z_0]\!] \leftarrow \mathsf{GoldenMult}([\![^g y_i]\!], [\![^g x]\!])$
4 $\quad [\![^g z_1]\!] \leftarrow \mathsf{GoldenMult}([\![^g y_i]\!], [\![^g z_0]\!])$
5 $\quad [\![^g z_1]\!] \leftarrow 3 - [\![^g z_1]\!]$
6 $\quad [\![^g z_2]\!] \leftarrow \mathsf{GoldenMult}([\![^g y_i]\!], {}^g 0.5)$
7 $\quad [\![^g y_{i+1}]\!] \leftarrow \mathsf{GoldenMult}([\![^g z_1]\!], [\![^g z_2]\!])$
8 $[\![^g w]\!] \leftarrow \mathsf{GoldenMult}([\![^g x]\!], [\![^g y_k]\!])$
9 **return** $[\![^g w]\!]$

The protocol is following. We first compute the inverse square root of the input x and then multiply it with x. There exists an iterative algorithm for $\frac{1}{\sqrt{x}}$ where the formula for the nth approximation is $y_{n+1} = 0.5 y_n (3 - x y_n^2)$. The reason why we use inverse square root to compute square root is that general iterative methods for square root need division, which is too costly in our setting.

To obtain the starting approximations, we shall use the function $\mathsf{TwoPowerConst}$ where the constants are $2^{\frac{i}{2}}$ – if $x \in [2^j, 2^{j+1})$, the function will return $2^{\frac{j}{2}}$. We then iteratively compute $y_{n+1} = 0.5 y_n (3 - x y_n^2)$ k times, thus obtaining an approximation of $\frac{1}{\sqrt{x}}$. In the end, we will multiply $[\![\frac{1}{\sqrt{x}}]\!]$ with $[\![x]\!]$, obtaining $[\![\sqrt{x}]\!]$.

7. RESULTS AND CONCLUSIONS

7.1. Benchmarking

We have implemented and benchmarked most of the protocols given in this thesis. In this chapter we give benchmarks for our implementations. One must note that the benchmarks have been run on fairly different settings. Thus while we can put some data on the same graph, it would not give a fair overview on the relative effectiveness of the different methods, as they were not run on the same conditions. Also, the methods have different precisions. Most importantly, we should note that the golden number algorithms were run using the PDSL optimizer in a domain-specific language that gave large gains for all algorithms it was run on. For example, the optimizer made floating-point multiplication up to 14 times more efficient [59, p.119]. For fairness, we compare golden number multiplication with multiplications of other data types that have been optimized with the same engine.

It would not be possible to apply the PDSL algorithms to all of the methods, as the point-counting methods are built in such a way that the algorithms obtained can not be optimized with the PDSL algorithm. The reason behind this is that the optimizer works on the dependency graph of the algorithm and the dependency graphs of the algorithms that use point-counting methods are relatively large and thus applying those methods to point-counting algorithms in infeasible.

7.1.1. Hybrid Method Benchmarking

For the hybrid method the four described functions were implemented on Sharemind 3 computing platform that uses 3-additive secret sharing and the implementations were benchmarked. To measure the performance of the floating point operations the developed software was deployed on three servers connected with fast network connections.

More specifically, each of the servers used contains two Intel X5670 2.93 GHz CPUs with 6 cores and 48 GB of memory. They were connected with 1 Gbps Ethernet connection. Since on Sharemind parallel composition of operations is more efficient than sequential composition, all the operations were implemented as vector operations. To see how much the vector size affects the average performance, we ran tests for different input sizes for all our inputs. The results given are the amortized costs — for example, if it took 3500 microseconds to evaluate a 1000 instances of an operation, then the amortized cost for that operation is 3.5 microsecond. We did 5 tests for each operation and input size and computed the average.

Note that for the square root protocol we applied the Correction algorithm to only one side due to not taking into account the case where the result would be ill-formed when the input was zero. However, this requires only one more secure multiplication which is performed in parallel with another multiplication, which is a very small cost when compared with the overall cost of the protocol.

7.1.2. Point-Counting Benchmarking

We implemented three functions – inverse, square root and logarithm using the point-counting technique. All three computing nodes for the Sharemind platform that we used contained two Intel X5670 2.93 GHz CPUs with 6 cores and had 48 GB of RAM. They were connected with 1 Gbps Ethernet connection. Although we optimized the methods concerning round-efficiency, total communication cost became the deciding factor for efficiency. Since these methods were designed to fully use the communication capacity of channels, communication cost is proportional to time. We performed tests for 32-bit numbers and 64-bit numbers as the basic integer data type, and with different precision levels. To see how vector size affects the performance, we ran tests for different vector sizes. We performed 20 tests for every value given here for the inverse, square root and the Gaussian error function and 10 tests for every value for the logarithm and averaged the result.

We chose the precision parameters for square root and inverse for the following reasons. We wished to have a near-maximal precision for both 32-bit and 64-bit numbers but for practical reasons, we implemented the method for 30 or 62 bits of precision. We also ran tests for 16 bits of precision for 32-bit numbers and 32 bits of precision for 64-bit numbers. These precisions are approximately half of the near-maximal precision but also close to the precisions of the corresponding functions computed with the hybrid method.

Comparing these benchmarks with the near-maximal precision we can see how doubling the number of bits for precision also approximately doubles the execution time. Based on the nature of the protocol, we can also assume that this pattern also holds more generally — when a protocol with n bits precision would take time t, then the same protocol would take time $2t$ for $2n$ bits of precision. Consequently, this protocol can be used with reasonable efficiency for applications needing very high precision. Verifying this conclusion assumes implementation of our algorithms on a platform providing 128-bit primitive types, so this remains the subject for future research.

We can also see that for 64-bit numbers, the performance for vector size 10000 is poorer than for vector size 1000. This happens because for 64-bit numbers, the number of maximal possible comparison operations that can be efficiently performed in parallel is smaller than 10000 and thus we have to perform more operations in a round than can be efficiently computed in parallel.

As for the logarithm, we implemented a logarithm function for 32-bit and 64-bit fixed-point numbers, although only for positive logarithms. We have reason to believe though, that as computing both positive and non-positive logarithms only means extending the starting interval somewhat, this would not lead to a notable decrease in performance.

Our implemented methods had precisions of 2^{-15} and 2^{-31}, respectively, due to the respective radix-point used — for the radix-point used, higher precision was not possible. We can see that while the method proposed by Aliasgari *et al*

can achieve very high precision due to the nature of the floating-point data type, the point-counting method is faster, and for larger vector sizes, faster by several orders of magnitude.

7.1.3. Golden Numbers Benchmarking

We have implemented golden section numbers on the SHAREMIND SMC platform. We chose SHAREMIND because of its maturity, tooling, and availability of fixed-point and floating-point numbers. As existing number systems were already implemented using SHAREMIND's domain-specific language [49], we decided to also use it for the golden section representation. The protocol language provides us with directly comparable performance and allows to avoid many complexities that a direct C++ implementation would have.

Performance measurements were made on a cluster of three computers connected with 10Gbps Ethernet. Each cluster node was equipped with 128GB DDR4 memory and two 8-core Intel Xeon (E5-2640 v3) processors, and was running Debian 8.2 Jessie with memory overcommit and swap disabled.

To provide a clear overview of accuracy and speed trade-offs, we measured the performance of golden section numbers on multiple bit-widths. Generally, higher bit-widths offer us better accuracy for the cost of performance.

We implemented three versions of golden numbers — where the underlying integers were 32-bit, 64-bit, and 128-bit integers respectively. A golden section number where the underlying integer type has n bits provides comparable accuracy to n-bit fixed-point numbers with radix point at $\lfloor n/2 \rfloor$.

Accuracy was measured experimentally, by identifying the range of inputs in which the largest errors should be found, and then uniformly sampling this range to find maximum error.

Performance measurements were made on a cluster of three computers connected with 10Gbps Ethernet. Each cluster node was equipped with 128GB DDR4 memory and two 8-core Intel Xeon (E5-2640 v3) processors, and was running Debian 8.2 Jessie with memory overcommit and swap disabled.

We measured each operation on various input sizes, executing the operation in parallel on the inputs. Each measurement was repeated a minimum of ten times and the mean of the measurements was recorded. Measurements were performed in randomized order. Note that due to the networked nature of the protocols, parallel execution improves performance drastically up to the so called *saturation point*.

7.2. Benchmarking Results

7.2.1. Inverse

Table 2 compares various results of computing the inverse function.

For 32-bit hybrid version fixed-point numbers we used the polynomial

Protocol	1	10	100	1000	10000
Catrina, Dragulin, 128 bits, AppDiv2m, LAN(ms) [22]	3.39				
Catrina, Dragulin, 128 bits, Div2m, LAN(ms) [22]	1.26				
Kamm and Willemson, Chebyshev, 32 bits [47]	0.17	1.7	15.3	55.2	66.4
Kamm and Willemson, Chebyshev, 64 bits [47]	0.16	1.5	11.1	29.5	47.2
Hybrid,32 bits, accuracy $1.3 \cdot 2^{-13}$	0.99	8.22	89.73	400.51	437.94
Hybrid 64 bit, accuracy $1.3 \cdot 2^{-26}$	0.82	8.08	62.17	130.35	171.1
Point-counting, 32 bits, accuracy 2^{-30}	0.34	3.60	21.97	98.13	106.15
Point-counting, 64 bits, accuracy 2^{-62}	0.24	2.21	10.8	48.1	37.1
Point-counting, 32 bits, accuracy 2^{-16}	0.61	8.25	40.8	190.9	212.01
Point-counting, 64 bits, accuracy 2^{-32}	0.45	4.11	23.31	100.0	67.13
Fixed-point 32 bits [31]	221.11	1490.79	3674.99	4264.14	5888.87
Fixed-point 64 bits [31]	160.92	733.24	998.84	1276.98	1814.06
Golden 32-bit	168.65	1459.13	7869.32	15468.27	16027.36
Golden 64-bit	125.40	1002.57	3736.34	5389.09	5336.79
Golden 128-bit	96.61	482.03	929.80	1123.85	1275.11

Table 2: Operations per second for different implementation of the inverse function for different input sizes.

$$
\begin{aligned}
&8.528174592103877 - 29.937500008085948 \cdot x \\
&\quad + 55.37549588994695 \cdot x^2 - 56.93285001066663 \cdot x^3 \\
&\quad + 30.856441181457452 \cdot x^4 - 6.889823228694366 \cdot x^5.
\end{aligned} \tag{7.1}
$$

For 64-bit hybrid version fixed-point numbers we used the polynomial

$$
\begin{aligned}
&15.599242404917524 - 109.93750000000036 \cdot x \\
&\quad + 462.0659715136437 \cdot x^2 - 1286.8971364795452 \cdot x^3 \\
&\quad + 2493.839270222642 \cdot x^4 - 3431.3944591425357 \cdot x^5 \\
&\quad + 3352.5408224920825 \cdot x^6 - 2279.4653178732265 \cdot x^7 \\
&\quad + 1027.2836075390694 \cdot x^8 - 276.20062206966935 \cdot x^9 \\
&\quad + 33.566121401778425 \cdot x^{10}
\end{aligned} \tag{7.2}
$$

These coefficients are based on the table on page 175 in [58].

For point-counting, we chose the precisions 2^{-30} and 2^{-62} in order to obtain a maximal possible precision, and 2^{-16} and 2^{-32} in order to compare with the hybrid method.

For golden section numbers, our precision is in the order of magnitude of 2^{-9} for 32-bit numbers, 2^{-23} for 64-bit numbers and 2^{-51} for 128-bit numbers. This was experimentally verified.

We see that our results perform the previous results and that the golden section protocols outperform our other methods. However, one must keep in mind that these were run on different system and that the golden section numbers benefited from the PDSL optimizations. For this reason, we included the fixed-point inverse function that was implemented in [31] in the same system. The fixed-point numbers have comparable strengths and weaknesses to golden section numbers and are thus a good comparison point. Due to precision reasons, it is fair to compare 32-bit fixed point numbers to 64-bit golden section numbers and 64-bit fixed-point numbers to 128-bit golden section numbers. We see that they are about in the same effectiveness range, with fixed-point numbers being more effective in some cases and golden section numbers in other cases. Seeing that golden section numbers are a new number type and thus can probably be improved on, this looks promising for the golden section numbers.

7.2.2. Square Root

Table 3 compares previous results for computing the square root with our results. We used the polynomial

$$
\begin{aligned}
&0.19536315261152867 \\
&\quad + 1.562905220892229 \cdot x - 1.736561356546921 \cdot x^2 \\
&\quad + 1.852332113650049 \cdot x^3 - 1.3230943668928923 \cdot x^4 \\
&\quad + 0.5488391447851997 \cdot x^5 - 0.09978893541549086 \cdot x^6
\end{aligned} \tag{7.3}
$$

Protocol	1	10	100	1000	10000
Liedel [50]	0.204				
Kamm and Willemson, 32 bits [47]	0.09	0.85	7	24	32
Kamm and Willemson, 64 bits [47]	0.08	0.76	4.6	9.7	10.4
Hybrid, 32 bits	0.77	7.55	70.7	439.17	580.81
Hybrid, 64 bits	0.65	6.32	41.75	78.25	119.99
Point-counting, 32 bits, accuracy 2^{-30}	0.30	2.98	19.97	94.13	101.8
Point-counting64 bits, accuracy 2^{-62}	0.21	1.93	9.23	44.79	37.13
Point-counting, 32 bits, accuracy 2^{-16}	0.49	5.98	35.2	152.0	202.3
Point-counting, 64 bits, accuracy 2^{-32}	0.38	3.59	21.2	86.0	79.5
Fixed-point, 32 bits [31]	254.82	2221.34	10322.99	19082.67	20698.29
Fixed-point, 64 bits, [31]	194.28	1389.53	4123.38	5425.28	5892.21
Golden 32-bit	93.11	814.32	4944.88	6741.24	6698.23
Golden 64-bit	57.25	449.16	1438.58	1941.49	1798.58
Golden 128-bit	66.37	383.76	839.77	1052.07	1189.45

Table 3: Operations per second for different implementation of the square root function for different input sizes.

to compute $\sqrt{x}$ for 32-bit fixed-point numbers and the polynomial

$$\begin{aligned} & 0.11762379327093657 + 2.6615226192244417 \cdot x \\ & - 9.371849704313805 \cdot x^2 + 36.81121979293309 \cdot x^3 \\ & - 119.39255388354168 \cdot x^4 + 310.12390025990817 \cdot x^5 \\ & - 644.7233686085026 \cdot x^6 + 1075.8084777766278 \cdot x^7 \\ & - 1442.1892383934844 \cdot x^8 + 1549.0933690616039 \cdot x^9 \\ & - 1323.521774124341 \cdot x^{10} + 887.547679235167 \cdot x^{11} \\ & - 457.04755897707525 \cdot x^{12} + 174.4585858163298 \cdot x^{13} \\ & - 46.49878087370222 \cdot x^{14} + 7.724960904027444 \cdot x^{15} \\ & - 0.6022146941311717 \cdot x^{16} \end{aligned} \tag{7.4}$$

to compute $\sqrt{x}$ for 64-bit fixed-point numbers.

These polynomials are Chebyshev polynomials. Note that while we should have performed corrections on both sides, we only performed it on one side. This, however, adds only one round so the increase is probably not very noticeable.

We estimate the error to be no larger than 2^{-17} for 32 bit case and 2^{-34} for the 64 bit case. We had $m = 31$ in the 32 bit case and $m = 52$ in the 64 bit case.

For reasons similar to the inverse case, we chose the precisions 2^{-30} and 2^{-62} in order to obtain a maximal possible precision, and 2^{-16} and 2^{-32} in order to compare with the hybrid method.

For golden section numbers, our precision is in the order of magnitude of 2^{-7} for 32-bit numbers, 2^{-16} for 64-bit numbers and 2^{-34} for 128-bit numbers. These estimates were obtained experimentally.

We see again that the hybrid model greatly improves on the previous approaches and that the point-counting gives a comparable result with the ability to precisely tune the result.

The results for golden numbers are again comparable to the fixed-point numbers, although the fixed-point numbers are better in this respect.

7.2.3. Exponent

Table 4 compares previous results for computing the exponent with the hybrid method. Our results are up to 2 times faster than the best previously existing implementations. We estimate the error to be no larger than 2^{-17} for 32 bit case and 2^{-39} for the 64 bit case. We had $m = 30$ in the 32 bit case and $m = 62$ in the 64 bit case.

We used the polynomial

$$\begin{aligned} & 1.00000359714456 \\ & + 0.692969550931914 \cdot x + 0.241621322662927 \cdot x^2 \\ & + 0.0517177354601992 \cdot x^3 + 0.0136839828938349 \cdot x^4 \end{aligned} \tag{7.5}$$

Protocol	1	10	100	1000	10000
Aliasgari *et al*. [5]		6.3	9.7	10.3	10.3
Kamm and Willemson, (Chebyshev) 32 bits [47]	0.11	1.2	11	71	114
Kamm and Willemson, (Chebyshev) 64 bits [47]	0.11	1.1	9.7	42	50
Hybrid, 32 bits	0.24	2.41	24.03	104.55	126.42
Hybrid, 64 bits	0.23	2.27	16.66	47.56	44.84

Table 4: Operations per second for different implementation of the exponential function for different input sizes.

to compute e^x on 32-bit fixed-point numbers and the polynomial

$$\begin{aligned} & 1.0000000000010827 \\ & + 0.693147180385251 \cdot x + 0.24022651159438796 \cdot x^2 \\ & + 0.055504061379894304 \cdot x^3 + 0.009618370224295783 \cdot x^4 \\ & + 0.0013326674872182274 \cdot x^5 + 0.00015518279382265856 \cdot x^6 \\ & + 0.000014150935770726401 \cdot x^7 \\ & + 0.0000018751971557376 \cdot x^8 \end{aligned} \tag{7.6}$$

to compute e^x on 32-bit fixed-point numbers. These coefficients are based on the table on page 201 in [58].

7.2.4. Gaussian Error Function

For error function, we used the following polynomials for $n = 32$:

If $n = 32$, we will use the following polynomials.

$$\begin{aligned} p_0(x) = {} & 1.1283793940340756 \cdot x - 0.0000026713076584281906 \cdot x^2 \\ & - 0.3761072585979022 \cdot x^3 - 0.00009283890849651041 \cdot x^4 \\ & + 0.1131594785717268 \cdot x^5 - 0.000814296779279163 \cdot x^6 \\ & - 0.025351408042362075 \cdot x^7 - 0.0020389298750037445 \cdot x^8 \\ & + 0.007118807679212721 \cdot x^9 - 0.0010650286415166768 \cdot x^{10} \\ & - 0.0006807918740908649 \cdot x^{11} + 0.00019635679847600037 \cdot x^{12} \end{aligned}$$

$$\begin{aligned} p_1(x) = {} & 0.02817728429 + 0.9359512202 \cdot x \\ & + 0.5434989619 \cdot x^2 - 1.19447516 \cdot x^3 + 0.6900358762 \cdot x^4 \\ & - 0.1783646656 \cdot x^5 + 0.01787727625 \cdot x^6 \end{aligned}$$

$$\begin{aligned} p_2(x) = {} & -0.942158979 + 4.022796153 \cdot x - 3.575876227 \cdot x^2 + 1.760980817 \cdot x^3 \\ & - 0.5145918337 \cdot x^4 + 0.08734960153 \cdot x^5 - 0.007365154875 \cdot x^6 \\ & + 0.00009644273755 \cdot x^7 + 0.00001896345963 \cdot x^8 \end{aligned}$$

$$
\begin{aligned}
p_3(x) = {} & 0.8576789792 \\
& + 0.2245825763 \cdot x - 0.1480373801 \cdot x^2 + 0.05217363374 \cdot x^3 \\
& - 0.01036910781 \cdot x^4 + 0.001101880093 \cdot x^5 - 0.00004891580119 \cdot x^6
\end{aligned}
$$

For $n = 64$, we used the following polynomials for $\operatorname{erf}(x)$:

$$
\begin{aligned}
p_0(x) = {} & 1.1283793940340756 \cdot x - 0.0000026713076584281906 \cdot x^2 \\
& - 0.3761072585979022 \cdot x^3 - 0.00009283890849651041 \cdot x^4 \\
& + 0.1131594785717268 \cdot x^5 - 0.000814296779279163 \cdot x^6 \\
& - 0.025351408042362075 \cdot x^7 - 0.0020389298750037445 \cdot x^8 \\
& + 0.007118807679212721 \cdot x^9 - 0.0010650286415166768 \cdot x^{10} \\
& - 0.0006807918740908649 \cdot x^{11} + 0.00019635679847600037 \cdot x^{12}
\end{aligned}
$$

$$
\begin{aligned}
p_1(x) = {} & 0.006765005 + 1.068755853503136 \cdot x + 0.2421008129968042 \cdot x^2 \\
& - 0.9749339270141031 \cdot x^3 + 1.0041963324534586 \cdot x^4 \\
& - 1.088243712366528 \cdot x^5 + 1.0471332876840715 \cdot x^6 \\
& - 0.6926003063553184 \cdot x^7 + 0.30152947241780975 \cdot x^8 \\
& - 0.08606929528345982 \cdot x^9 + 0.01564245229830543 \cdot x^{10} \\
& - 0.0016528687686237157 \cdot x^{11} + 0.00007769002084531931 \cdot x^{12}
\end{aligned}
$$

$$
\begin{aligned}
p_2(x) = {} & 1.363422003 - 4.975745564 \cdot x + 12.27381879 \cdot x^2 \\
& - 14.9219185 \cdot x^3 + 11.19823154 \cdot x^4 - 5.711718595 \cdot x^5 \\
& + 2.080646945 \cdot x^6 - 0.5558494039 \cdot x^7 + 0.1102226843 \cdot x^8 \\
& - 0.01621608793 \cdot x^9 + 0.00174112969 \cdot x^{10} - 0.0001305244651 \cdot x^{11} \\
& + 0.000006161109129 \cdot x^{12} - 0.0000001385406897 \cdot x^{13}
\end{aligned}
$$

$$
\begin{aligned}
p_3(x) = {} & -0.7639003533 + 4.00501476 \cdot x - 4.064372065 \cdot x^2 \\
& + 2.419369363 \cdot x^3 - 0.9308524286 \cdot x^4 + 0.2400308095 \cdot x^5 \\
& - 0.04147567521 \cdot x^6 + 0.004630079428 \cdot x^7 \\
& - 0.0003029475561 \cdot x^8 + 0.000008849881454 \cdot x^9
\end{aligned}
$$

We see that the hybrid method gives better results than the paper by Kamm and Willemson, but that the progress is not as great as in the case of the inverse or the square root. This is natural, as we saw that the error function does not break so naturally into a component based on the exponent of the floating-point number and a component on which we can do fixed-point polynomial evaluation as the inverse and square root do.

7.2.5. Logarithm

Table 6 provides data for computing the logarithm. We can see that while the solution of Aliasgari has a much higher precision, the point-counting technique provides a much efficient solution.

	1	10	100	1000	10000
Kamm and Willemson, 32 bits [47]	0.1	0.97	8.4	30	39
Kamm and Willemson, 64 bits [47]	0.09	0.89	5.8	16	21
Hybrid, 32-bit	0.5	4.41	30.65	45.42	49.88
Hybrid, 64-bit	0.46	4.13	21.97	19.54	26.11

Table 5: Operations per second for different implementation of the Gaussian error function for different input sizes.

	1	10	100	1000	10000	100000
Aliasgari [5] , accuracy 2^{-256}		12.36	12.5	13.3	13.3	13.5
Point-counting, 32-bit, accuracy 2^{-15}	2.39	15.43	119.2	549.9	1023.6	1288.9
Point-counting, 64-bit, accuracy 2^{-31}	0.90	6.8	37.9	152.5	244.3	275.6

Table 6: Operations per second for different implementation of the logarithm function for different input sizes.

Protocol	1	10	100	1000	10000
Golden 32-bit	1329.58	12118.27	101972.14	451789.54	598745.03
Golden 64-bit	1129.64	10899.90	77892.54	207509.35	201411.57
Golden 128-bit	168.82	1267.73	13510.12	48624.94	62325.29

Table 7: Operations per second for different sizes of golden number normalization for different input sizes.

Protocol	1	10	100	1000	10000
Fixed-point, 32-bit	955.33	8694.44	74014.86	329001.48	504339.34
Fixed-point 64-bit	794.18	8081.07	56438.50	208979.43	249032.01
Floating-point, 32-bit	618.35	5788.31	40466.17	147214.26	241552.66
Floating-point, 64-bit	514.11	4661.18	33755.96	106290.48	133894.80
Log-float, 16-bit	1755.19	17729.87	153751.54	726174.22	1453678.53
Log-float, 32-bit	1526.81	14872.98	97025.21	433373.20	708271.48
Log-float 64-bit	1755.19	12118.27	101972.14	451789.54	598745.03
Log-float 64-bit	1213.95	11775.79	68705.86	266269.04	339654.53
Golden 32-bit	983.81	9555.30	73186.08	234154.75	258999.05
Golden 64-bit	900.35	7133.79	77892.54	207509.35	201411.57
Golden 128-bit	168.82	1267.73	25530.00	30122.24	28804.89

Table 8: Operations per second for multiplication of different real-number data types for different sizes.

7.2.6. Golden Number Normalization

Table 7 provides data for golden number normalization. While this is a very specific operation to golden section numbers, it is still interesting to see how vector size and bitlength influence efficiency. We see that for small vector sizes, the different bitlengths are rather similar in efficiency, but that as the vector length grows, the differences start to grow also. This might be due to the fact that while the round complexity of the normalization is not very high, being $\log n + 4$ while the bits sent tends to grow superlinearly as n grows.

7.2.7. Multiplication

The results comparing golden section multiplication to fixed-point and floating-point multiplication in the same setting are given in table 8.Measuring golden section multiplication fairly is nontrivial — namely, for practical sizes, one needs to multiply normalized values and those values often are normalized as well. However, if we normalize both before and after multiplication, we are doing double the work, thus we should only normalize once. Whether we normalize before or after the multiplication operation also affects precision. These measurements were made so that we normalize the inputs and leave the output un-normalized. While there are scenarios where this scenario is not what happens, it is one of the most common ones and thus we chose it. The error of the 32-bit of golden multiplication is 2^{-11}, 64-bit of golden multiplication is 2^{-24}, 128-bit of golden multiplication is 2^{-52}.

Comparatively, the error of fixed-point multiplication is 2^{-m} for a fixed-point number with the radix-point m, and the error of n-bit floating-point multiplication is $2^{-(n-1)}$.

We see that the golden number multiplication is comparable to the fixed-point

multiplication of the corresponding precision. Its properties make the golden section numbers more suitable for high latency and high throughput situations, and also better for applications that perform many consecutive operations on small inputs.

The worse performance of golden section numbers after the saturation point can be wholly attributed to increased communication cost. Namely, every multiplication of 64-bit golden section numbers requires 6852 bits of network communication, but a single 32-bit multiplication requires only 2970. We also note that golden section numbers are a new and relatively unstudied data type and thus there may be possibilities of improving it.

7.3. Conclusions

7.3.1. Hybrid Method

We can see that that the hybrid method is much more efficient for larger vector sizes. Considering the amortized cost, it greatly outperforms the approaches that were built for computing only a single operation. For inverse and square root functions, it also outperforms the results of Kamm and Willemson. As this technique was based on improving the results of that paper by replacing a floating-point number with a fixed-point number, this shows that for the inverse and square root the technique gives the expected benefit.

Considering the exponent and the Gaussian error function, our approach still often outperforms the results of Kamm and Willemson, however, the improvement is not as remarkable as in the case of the inverse and square root functions.

Indeed, for the 64-bit exponential function with batch size 10000 the amortized cost of the current result is even slightly outperformed by both the corresponding result by Kamm and Willemson and the amortized cost of the 64-bit exponential function with batch size 1000.

The reason behind why the inverse and square root functions gain more from the optimization presented in this chapter is that it is possible to utilize the fixed-point polynomial evaluation there in a more contained, modular sense — for both the exponential and the Gaussian error function more corrections have to be made to ensure that the inputs and outputs belong to the correct intervals. However, even in those cases, there is generally a notable improvement in efficiency.

7.3.2. Point-Counting

The point-counting method provides an oblivious evaluation of special-format single-variable functions, including, but not limited to functions that can be represented as finding roots of polynomials with secret coefficients. Several important functions belong to this class (e.g. various power functions and binary logarithm). The method is easy to implement and rather flexible as it can be used for various vector sizes and precisions, is designed to fully use the communication capacities

of channels, and it offers good performance/precision ratio and can effectively be used for both small and large datasets and give maximal precision for fixed-point data types. It also allows one to exactly choose the level of precision needed — thus it can be used as a method to refine already obtained answers.

7.3.3. Golden Section Numbers

Golden number representation allows for very fast addition, and with a multiplication speed that is comparable with that of fixed-point numbers. For elementary functions, it performs about in the same efficiency with fixed-point numbers. Seeing that the golden section numbers have arithmetic properties that were not used in this implementation (such that multiplying or dividing a golden section number by φ can be done locally), it is possible that these properties allow for tricks that can make some operations faster. In our approach, the golden section multiplication also took considerably less rounds than the fixed-point multiplication, although the communication overhead was higher. For example, 32-bit fixed-point multiplication took 16 rounds and 2970 bits of network communication , whereas the comparable 64-bit fixed-point number took 11 rounds and 6852 bits of network communication. This suggests that the golden section numbers and fixed-point numbers have their own strengths and weaknesses which suggests that although golden section numbers are a new data-type, there are some settings in which it is more effective than the fixed-point numbers. Thus it is worthy of further research.

BIBLIOGRAPHY

[1] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[2] Abbas Acar, Hidayet Aksu, Arif Selcuk Uluagac, and Mauro Conti. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *CoRR*, 2017.

[3] Mehrdad Aliasgari and Marina Blanton. Secure Computation of Hidden Markov Models. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–12, July 2013.

[4] Mehrdad Aliasgari, Marina Blanton, and Fattaneh Bayatbabolghani. Secure Computation of Hidden Markov Models and Secure Floating-Point Arithmetic in the Malicious Model. *International Journal of Information Security*, 16(6):577–601, Nov 2017.

[5] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure Computation on Floating Point Numbers. In *NDSS*, 2013.

[6] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure Computation on Floating Point Numbers. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[7] Shuangjie Bai, Geng Yang, Jingqi Shi, Guoxiu Liu, and Zhaoe Min. Privacy-Preserving Oriented Floating-Point Number Fully Homomorphic Encryption Scheme. *Security and Communication Networks*, 2018, 2018.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.

[9] George Bergman. A Number System with an Irrational Base. *Mathematics Magazine*, 31(2):98–110, 1957.

[10] George Robert Blakley. Safeguarding Cryptographic Keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, 1979.

[11] Dan Bogdanov. *Sharemind: Programmable Secure Computations with Practical Applications*. PhD thesis, University of Tartu, 2013.

[12] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS'08*, volume 5283 of *LNCS*, pages 192–206, 2008.

[13] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. Improved protocols for the Sharemind Virtual Machine. Technical Report T-4-10, Cybernetica, `http://research.cyber.ee/`., 2010.

[14] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-

performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[15] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security*, pages 325–343, 2009.

[16] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In *Financial Cryptography and Data Security*, pages 142–147, 2006.

[17] Piers Bohl. Über ein in der Theorie der säkularen Störungen vorkommendes Problem. *Journal für die reine und angewandte Mathematik*, 135:189–283, 1909.

[18] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 15–15, 2010.

[19] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22, 2015.

[20] Octavian Catrina. Round-Efficient Protocols for Secure Multiparty Fixed-Point Arithmetic. 07 2018.

[21] Octavian Catrina and Sebastiaan de Hoogh. Improved Primitives for Secure Multiparty Integer Computation. In *Security and Cryptography for Networks*, pages 182–199. 2010.

[22] Octavian Catrina and Claudiu Dragulin. Multiparty Computation of Fixed-Point Multiplication and Reciprocal. In *Database and Expert Systems Application, 2009. DEXA '09. 20th International Workshop on*, pages 107–111, 2009.

[23] Octavian Catrina and Sebastiaan de Hoogh. Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *Computer Security–ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 134–150. 2010.

[24] Octavian Catrina and Amitabh Saxena. Secure Computation with Fixed-Point Numbers. In *Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. 2010.

[25] Heewon Chung. *Secure Computation via Homomorphic Encryption*. PhD thesis, Seoul National University, 2017.

[26] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. 1st edition, 2015.

[27] Eric Crockett and Chris Peikert. $\Lambda o \lambda$: functional lattice cryptography. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 993–1005, 2016.

[28] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography – PKC 2009*, pages 160–179, 2009.

[29] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 643–662, 2012.

[30] Sebastiaan de Hoogh. *Design of Large Scale Applications of Secure Multiparty Computation : Secure Linear Programming*. PhD thesis, Technische Universiteit Eindhoven, 2012.

[31] Vassil Dimitrov, Liisi Kerik, Toomas Krips, Jaak Randmets, and Jan Willemson. Alternative Implementations of Secure Real Numbers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 553–564, 2016.

[32] Martin Franz. *Secure Computations on Non-Integer Values*. PhD thesis, Technische Universität Darmstadt, 2011.

[33] Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser, and Heike Schröder. Secure Computations on Non-Integer Values. In *2010 IEEE International Workshop on Information Forensics and Security*, pages 1–6, Dec 2010.

[34] Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh K Jha, Stefan Katzenbeisser, and Heike Schröder. Secure Computations on Non-Integer Values with Applications to Privacy-Preserving Sequence Analysis. *Information Security Technical Report*, 17(3):117 – 128, 2013. Security and Privacy for Digital Ecosystems.

[35] Martin Franz and Stefan Katzenbeisser. Processing Encrypted Floating Point Signals. In *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security*, pages 103–108, 2011.

[36] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[37] Subrandom numbers. `https://web.archive.org/web/20160304125746/http://mollwollfumble.blogspot.com/`. Online; accessed 22 January 2019.

[38] Wilko Henecka, Nigel Bean, and Matthew Roughan. Conversion of Real-

Numbered Privacy-Preserving Problems into the Integer Domain. In *Information and Communications Security*, pages 131–141, 2012.

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

[40] Erfc. `http://mathworld.wolfram.com/Erfc.html`. Online; accessed 22-January-2019.

[41] Fibonacci Number. `http://mathworld.wolfram.com/FibonacciNumber.html`. Online; accessed 22-January-2019.

[42] Golden Ratio. `http://mathworld.wolfram.com/GoldenRatio.html`. Online; accessed 22-January-2018.

[43] CF1: uniform distribution and Denjoy-Koksma's inequality. `https://matheuscmss.wordpress.com/2012/02/16/cf1-uniform-distribution-and-denjoy-koksmas-inequality/`. Online; accessed 22-January-2019.

[44] Sharemind. `https://sharemind.cyber.ee/`. Online; accessed 22-January-2019.

[45] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, 2011.

[46] Liina Kamm. *Privacy-Preserving Statistical Analysis Using Secure Multi-Party Computation*. PhD thesis, University of Tartu, 2015.

[47] Liina Kamm and Jan Willemson. Secure Floating Point Arithmetic and Private Satellite Collision Analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[48] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for Robust and Scalable Integer and Floating-Point Arithmetic. LNCS. 2016. Accepted to Workshop on Applied Homomorphic Cryptography 2016.

[49] Peeter Laud and Jaak Randmets. A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1492–1503, 2015.

[50] Manuel Liedel. Secure Distributed Computation of the Square Root and Applications. In *Information Security Practice and Experience*, volume 7232 of *Lecture Notes in Computer Science*, pages 277–288. 2012.

[51] Manuel Liedel. *Sichere Mehrparteienberechnungen und datenschutzfreundliche Klassifikation auf Basis horizontal partitionierter Datenbanken*. PhD thesis, Universität Regensburg, 2012.

[52] Yun-Ching Liu, Yi-Ting Chiang, Tsan-Sheng Hsu, Churn-Jung Liau, and Da-Wei Wang. Floating Point Arithmetic Protocols for Constructing Se-

cure Data Analysis Application. *Procedia Computer Science*, 22:152 – 161, 2013. 17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems - KES2013.

[53] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a Secure Two-party Computation System. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, 2004.

[54] Manoranjan Mohanty. *Secret Sharing Approach for Securing Cloud-Based Image Processing*. PhD thesis, National University of Singapore, 2013.

[55] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In *Theory of Cryptography*, pages 368–386, 2009.

[56] Martin Pettai and Peeter Laud. Automatic Proofs of Privacy of Secure Multi-party Computation Protocols against Active Adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 75–89, 2015.

[57] Pisot Number. `http://mathworld.wolfram.com/PisotNumber.html`. Online; accessed 31 August 2018.

[58] Boris A. Popov and Genadiy S. Tesler. *Vyčislenie funkcij na ÈVM - spravočnik (in Russian)*. 1984.

[59] Jaak Randmets. *Programming Languages for Secure Multi-Party Computation Application Development*. PhD thesis, University of Tartu, 2017.

[60] Ronald Linn Rivest, Len Adleman, and Michael Leonidas Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[61] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[62] Chibuike Ugwuoke, Zekeriya Erkin, and Reginald L. Lagendijk. Secure Fixed-point Division for Homomorphically Encrypted Operands. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, pages 33:1–33:10, 2018.

[63] Andrew Chi-Chih Yao. Protocols for Secure Computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[64] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, 1986.

ACKNOWLEDGMENTS

This research was supported by the European Regional Development Fund through Centre of Excellence in Computer Science (EXCS) and the Estonian Research Council under Institutional Research Grant IUT27-1 and Estonian Doctoral School in Information and Communication Technologies

I would like to thank my colleagues in STACC and Cybernetica for collaboration. I would like to thank Liina Kamm, Jaak Randmets, Liisi Kerik, Vassil Dimitrov, and Dominique Unruh. I would like to thank my family and friends for supporting me. I would especially like to thank Jan Willemson for supervision.

INDEX

SUMMARY IN ESTONIAN

Turvaliste reaalarvuoperatsioonide efektiivsemaks muutmine

Turvaline ühisarvutus on distsipliin, mis uurib seda, kuidas arvutada nõnda, et arvutavad osapooled ei saaks midagi uut teada andmete kohta, mis on arvutuse sisendiks. Sel distsipliinil on rakendusi mitmetes erinevates valdkondades, näiteks meditsiiniuuringutes ja arvutuste delegeerimises vähemusaldatud, kuid suure arvutusvõimsusega osapooltele. Käesolev doktoritöö uurib kolme võimalust, kuidas reaalarvuliste sisenditega ühisarvutust efektiivsemaks teha.

Esiteks me uurime meetodit, mis kombineerib ujukoma- ja püsikomatehteid. Kasutatav arvutüüp valitakse selle järgi, kumb antud situatsioonis efektiivsem on. Meetodi efektiivsus sõltub konkreetsest hinnatavast funktsioonist, sest arvesse tuleb võtta ka kulu, mis tekib arvutüüpide vahelisest teisendamisest. Mõnede funktsioonide arvutamise puhul kaalub meetodist saadav tulu teisenduskulud üles, aga see pole alati nii.

Teiseks uurime me meetodit, mis kasutab ära fakti, et mõnedes turvalise ühisarvutuse tehnikates on operatsioonide paralleelkompositsioon nende sekventsiaalkompositsioonist suurusjärkude võrra efektiivsem. Seda meetodit saab rakendada selliste monotoonsete funktsioonide f juures, mille puhul leidub lihtsasti arvutatab 'pöördfunktsioon' g. Me näitame, kuidas funktsiooni f väärtust mingis punktis saab välja arvutada hoopis paljude funktsiooni g paralleelis rakendamiste abil. See võib traditsioonilisest lähenemisest efektiivsem olla.

Kolmandaks tutvustame me uut arvutüüpi, mida nimetame kuldlõikearvudeks. Täisarvude paar (a,b) tähistab reaalarvu $a-\varphi b$, kus $\varphi = 1.618\ldots$ on kuldlõike konstant. Me näitame, et selle arvutüübiga saab saavutada turvalises paradigmas tulemusi, mis on võrreldavad turvaliste püsikomaarvudega. Kuna tegemist on uue ja seega väheuuritud arvutüübiga, on võimalik, et kuldlõikearvudel arvutamist saab muuta veel efektiivsemaks.

CURRICULUM VITAE

Personal data

Name:	Toomas Krips
Birth:	September 23rd, 1988, Tartu, Estonia
Citizenship:	Estonian
Languages:	English, Estonian, French, German
E-mail:	toomaskrips@gmail.com

Education

2012–...	University of Tartu, Ph.D. candidate in Comp. Science
2010–2012	University of Tartu, M.Sc. in Mathematics
2007–2010	University of Tartu, B.Sc. in Mathematics
2004–2007	Hugo Treffner Gymnasium
1995–2004	Tartu Secondary School of Business

Employment

01.09.2012–31.08.2016	STACC, Junior Researcher
01.09.2016–	University of Tartu, Teaching Assistant
01.09.2017–	University of Tartu, Junior Researcher

Scientific work

Main fields of interest:

- Secure Multiparty Computation
- Zero-Knowledge Proofs
- Combinatorics

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Toomas Krips
Sünniaeg:	September 23rd, 1988, Tartu, Eesti
Kodakondsus:	Eesti
Keeled:	eest, inglise, prantsuse, saksa
E-mail:	toomaskrips@gmail.com

Haridus

2012–...	Tartu Ülikool, arvutiteaduse doktorantuur
2010–2012	Tartu Ülikool, matemaatika magistrantuur
2007–2010	Tartu Ülikool, matemaatika bakalaureus
2004–2007	Hugo Treffneri Gümnaasium
1995–2004	Tartu Kommertsgümnaasium

Teenistuskäik

01.09.2012–31.08.2016	STACC, nooremteadur
01.09.2016–	University of Tartu, informaatika assistent
01.09.2017–	University of Tartu, krüptograafia nooremteadur

Teadustegevus

Peamised uurimisvaldkonnad:

- Turvaline ühisarvutus
- Nullteadmus
- Kombinatoorika

LIST OF ORIGINAL PUBLICATIONS

1. *Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations* Krips, Toomas; Willemson, Jan (2014). Proceedings of ISC 2014,, 12-14 October 2014, Hong Kong. Springer-Verlag, 179–197. (Lecture Notes in Computer Science; 8783).
2. *Point-Counting Method for Embarrassingly Parallel Evaluation in Secure Computation* Krips, Toomas; Willemson, Jan (2016). 8th International Symposium on Foundations & Practice of Security, October 26-28 2015, Clermont-Ferrand, France . Ed. Joaquin Garcia-Alfaro, Evangelos Kranakis, Guillaume Bonfante. Springer, 66–82. (Lecture Notes in Computer Science; 9482).
3. *Alternative Implementations of Secure Real Numbers* Dimitrov, Vassil; Kerik, Liisi; Krips, Toomas; Randmets, Jaak; Willemson, Jan (2016). Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (ACM CCS), Vienna, Austria, October 24-28, 2016. Ed. Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, Shai Halevi. ACM, 553–564.

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.