BUILDING BLOCKS FOR SECURE MULTI-PARTY COMPUTATION BASED ON SECRET SHARING

by

Chen Yuan

May 2022

A dissertation submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

Copyright by

Chen Yuan

2022

All Rights Reserved

The thesis of Chen Yuan was reviewed by the following:

Marina Blanton
Associate Professor of Computer Science and Engineering
Thesis Advisor, Chair of Committee

Hongxin Hu
Associate Professor of Computer Science and Engineering
Committee Member

Ziming Zhao
Assistant Professor of Computer Science and Engineering
Committee Member

Acknowledgments

First and foremost, I owe a debt of gratitude to my advisor Dr. Marina Blanton for guiding me through the journey of the PhD. Her immensely valuable guidance, continuous support, and endurance patience has greatly improved all the research I worked on. She shaped my development as a researcher from many different perspectives. I am particularly appreciative for the countless hours we spent meeting and teaching me how I should do the research.

I am grateful to my other committee members, Dr. Hongxin Hu and Dr. Ziming Zhao, for their invaluable inputs. Their knowledge and insight in security has prompted me to think more deeply about my research. In addition, I would like to extend my gratitude to Dr. Shambhu Upadhyaya and Dr. Weihang Wang for valuable suggestions during my qualify exam and to Dr. Hu Xiong for giving me my first exposure to research.

I would like to thank my collaborators, Dr. Michael Goodrich, Dr. Ahreum Kang, Dennis Murphy, and Alessandro Baccarini. I enjoyed working in this collaborative research environment and learned a lot from the experience.

Finally, this dissertation is the product of a long journey that would not have been possible without the support of my family. Thank you to my parents, grandparents, and Xiangnan for always being there for me and cheering me on. This is the best company I could ever hope for.

Table of Contents

List of Tables

List of Figures

Abstract

Secure multi-party computation (SMC) enables a group of participants to perform the computation on their private data jointly, where nothing about the original data will be released but the final results. SMC has shown potential for academic and practical applications in the last decade, ranging from applications in the academic field of secure computational geometry to more practical applications like privacy-preserving machine learning, financial analysis, and biomedical computations. Because the actual adoption of secure computation techniques depends on their performance in practice, it is essential to continue improving their performance. This dissertation focuses on common non-trivial building blocks used by many types of programs, where any advances in their performance would impact the runtime of programs that rely on them.

Firstly, we propose new constructions to access an array at a private location that could significantly outperform conventional implementations of these operations in the setting with an honest majority based on secret sharing. In particular, we propose a general construction that works for any number of computation participants that uses conventional Shamir secret sharing. In addition, we also propose a custom construction for the typical case of three parties, which outperforms the general construction.

Secondly, we explore how to implement binary search in secure computation. Realizing binary search in the context of secure multi-party computation, which demands data-oblivious execution, is extremely non-trivial. We propose a suite of protocols with different properties and different structures to search a private dataset of m elements by a private numeric key. Our proposed solutions result in $O(m)$ and $O(\sqrt{m})$ communication using only standard and readily available operations based on secret sharing.

Finally, we address the problem of floating-point summation in secure computation. Calculating the exact summation of m floating-point numbers is a non-trivial problem, which becomes much more challenging in the context of secure multi-party computation. We are not aware of any prior work that could achieve this goal in the context of secure computation. Therefore, we propose a superaccumulator based solution that could efficiently perform the summation without loss of accuracy.

Chapter 1

Introduction

Secure multi-party computation (SMC) allows a number of parties to compute jointly on private data without releasing unintended information about the private data to any party. Since its introduction by Yao [1, 2] in his seminal works, SMC has been the subject of research for many years but not commonly used in practice because of its complexity and overhead [3]. The last years have witnessed significant progress towards practical solutions for secure computation, which are now suitable for data and computations of significant sizes [4, 5, 6, 7, 8]. Because of its strong security guarantees, secure computation can be increasingly applied to secure statistical analysis of private data sets distributed among a number of participants [9, 10], as well as data analytics and decision making using private distributed data [11, 12, 13].

Of particular interest to the research community in recent years has been privacy-preserving machine learning, which uses non-trivial algorithms to analyze large volumes of data. Computation used in such analyses often requires access to data at private locations, be it due to the nature of data representation, e.g., in the form of sparse data sets or due to the nature of the algorithm itself. When such operations are executed as part of secure computation on private data, we must employ data-oblivious (i.e., data-independent) constructions for realizing the operations to eliminate leakage of private information.

Despite the progresses in the basic SMC applications, designing and building an efficient data-oblivious application in secure computation is still challenging. One of the major challenges is how to optimize performance as much as possible while ensuring obliviousness, which requires that the behavior or access pattern of a program leaks no information about the input data. For efficiency, both the local computation and the communication need to be highly optimized to reduce the computation latency. However, most modern algorithms and optimization strategies are unavailable or ineffective in secure computing because these methods do not satisfy the requirement of obliviousness. Although some existing technologies, such as Oblivious RAM (ORAM) [14, 15], could help with hiding the data access pattern, their performance is far from acceptable, especially for small-to-medium size datasets.

In this work, we study data-oblivious building blocks and algorithms which permit higher-level functionalities to be executed within secure multi-party computation frameworks, ideally with asymptotic complexities of data-oblivious constructions being as close as possible to those of their non-oblivious counterparts. The goals of this work are (1) to design efficient building blocks which could enhance the performance of secure computation applications that rely on them, and (2) to develop new building blocks which would improve the availability of secure computation in practice.

In Chapter 4, we start with the problem of accessing a private array at a private index. As the location of array access or the access pattern may leak sensitive information about the private inputs, obliviousness in array access is a significant property in secure computation. We develop new constructions for access to an array at a private location that significantly outperform conventional implementations of this operations in the setting with honest majority based on secret sharing.

In Chapter 5, we study the problem of performing a binary search in a private array with a private key. Although binary search is one of the most widely used data searching algorithms, realizing it in the context of secure multiparty computation which demands data-oblivious execution, however, is extremely non-trivial. Because of the nature of binary search, its access pattern will inevitably disclose the comparison results, which would leak information about the private target value and even the private array. We develop a suite of protocols with different properties and structure and further combine them into hybrid solutions to improve performance and asymptotic complexity.

In Chapter 6, we further explore the applications of data-oblivious technologies and address the problem of accurate floating-point summation. Calculating the exact summation of multiple floating-point numbers is a challenging problem, even without considering security and obliviousness. We propose the first private floating-point summation protocol in secure computation, which not only guarantees the accuracy but also has a constant number of communication rounds.

We finally conclude the dissertation in Chapter 7.

Chapter 2

Background

Secure multi-party computation refers to the ability of several participants to evaluate a function of their choice on private data without disclosing unintended information about the private data to the computation participants. Such techniques aim to provide the functionality of a trusted third party that conducts computation on private inputs received from one or more parties and returns results to the parties entitled to learn the output.

Throughout this work, we denote the parties that perform the computation as computational parties and denote the parties that own the input or output as input/output parties. It is noted that the input or output parties may be disjoint from the computational parties carrying out the computation (as in the case with outsourcing). We assume that the computational parties $P_1, \ldots, P_n$ and the input/output parties are connected with secure authenticated channels. The input will be shared among computational parties using secret sharing techniques which will be introduced in detail in Section 2.1. Prior to the computation, the input parties are responsible for generating the input shares. The computational parties hold their respective shares of the private inputs before the computation starts. Upon computation completion, the output parties reconstruct the results by using the shares received from all computational parties.

In the rest of this section, we introduce the fundamental cryptographic primitives as well as notions and definitions we use throughout this work.

2.1 Secret Sharing

A secret sharing (SS) scheme allows one to produce shares of secret x such that the secret can only be reconstructed by collecting more than t shares, where t is the threshold. In the context of secure multi-party computation, we create n shares from the secret x and each of the n participants receives x_i. In the case of (n, t) threshold secret sharing schemes, possession of shares stored at any t or fewer parties reveals no information about x, while access to shares stored at $t + 1$ or more parties allows for reconstruction of x. Of particular importance to secure multi-party computation are linear secret sharing schemes, which have the property that a linear combination of secret shared values can be performed locally on the shares.

In what follows, we perform the computation over a field or a ring. Because each secret-shared integer is a field or ring element, the size of the field $\mathbb{F}$ or the ring $\mathbb{R}$ needs to be large enough to be able to represent values in the desired range. For example, to be able to support computation on k-bit integers, we must have that $|\mathbb{F}| \geq 2^k$ or $|\mathbb{R}| \geq 2^k$. We use notation $[x]$ to denote that the integer x is secret shared among the participants using secret sharing. We will discuss about the representation of a secret floating-point number in Chapter 6.

Shamir Secret Sharing (SSS). Shamir secret sharing [16] (SSS) is an (n, t)-linear SS scheme with $t < n$, where computation takes places over a finite field $\mathbb{F}$.

A secret value $s \in \mathbb{F}$ is represented by a random polynomial of degree t with the free coefficient set to s. Each share of s corresponds to the evaluation of the polynomial on a unique non-zero point. Consequently, given $t+1$ or more shares, the parties can reconstruct the polynomial and learn s using Lagrange interpolation. Possession of t or fewer shares, on the other hand, information-theoretically reveals no information about s.

Replicated Secret Sharing (RSS). Replicated secret sharing (RSS) [17] is another type of linear secret sharing that can be used to realize (n, t)-threshold secret sharing (and can be defined for more general access structures Γ, but we limit our use to threshold structures only). RSS can be defined for any $n \geq 2$ and any $t < n$ and works over any finite ring. The RSS access structure uses the notion of qualified sets, which are all subsets of the participants who are permitted to reconstruct the secret (i.e., all subsets of $t+1$ or more parties in our case), while all other subsets are called unqualified. To secret-share private x using RSS, we additively split it into shares x_T such that $x = \sum_{T \in \mathcal{T}} x_T$, where $\mathcal{T}$ consists of all maximal unqualified sets (i.e., all sets of t parties in our case). Then each party P_i stores shares x_T for all $T \in \mathcal{T}$ subject to $i \notin T$. In the general case of (n, t)-threshold RSS, the total number of shares is $\binom{n}{t}$ with $\binom{n-1}{t}$ shares stored by each party, which can become large as n and t grow. However, for small n, the number of shares is small (e.g., with both (3, 1) and (3, 2) RSS, there are the total of 3 shares).

It is note that all protocols and building blocks proposed in this work could be implemented by using these two secret sharing techniques unless indicated otherwise. In the rest of this work, we implement our SSS protocols over a field and implement RSS protocols over a ring.

2.2 Threat Model

We consider a conventional secure multi-party setting with n computational parties, out of which at most t can be corrupt. We work in the setting with honest majority unless indicated otherwise, i.e., $t < n/2$ and focus on security against semi-honest participants, in which the participants are trusted to follow the prescribed computation, but might attempt to learn unauthorized information based on the information they possess. We use a standard simulation-based security definition that requires that the participants do not learn any information beyond their intended output.

As customary with techniques based on secret sharing, the set of computational parties does not have to coincide with (and can be formed independently from) the set of parties supplying inputs in the computation (input providers) and the set of parties receiving output (output recipients). Then if a computational party learns no output, the computation should not reveal any information to that party. Consequently, if we wish to design a functionality that takes input in the secret-shared form and produces shares of the output, any computational party should learn nothing from protocol execution.

We formally define the security in secure multi-party computation in the presence of semi-honest adversaries.

Definition 1. *Let parties $P_1, \ldots, P_n$ engage in a protocol Π that computes function $f(\mathsf{in}_1, \ldots, \mathsf{in}_n) = (\mathsf{out}_1, \ldots, \mathsf{out}_n)$, where in_i and out_i denote the input and output of party P_i, respectively. Let $\mathrm{VIEW}_\Pi(P_i)$ denote the view of participant P_i during the execution of protocol Π. More precisely, P_i's view is formed by its input and internal random coin tosses r_i, as well as messages $m_1, \ldots, m_k$ passed between the parties during protocol execution: $\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$*

denote a subset of the participants for $t < n$, $\mathrm{VIEW}_{\Pi}(I)$ denote the combined view of participants in I during the execution of protocol Π (i.e., the union of the views of the participants in I), and $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the projection of $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ on the coordinates in I (i.e., $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ consists of the i_1th, ..., i_tth element that $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ outputs). We say that protocol Π is t-private in the presence of semi-honest adversaries if for each coalition of size at most t there exists a probabilistic polynomial time (PPT) simulator S_I such that $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_{\Pi}(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$, where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$ and $\equiv$ denotes computational or statistical indistinguishability.

In addition to semi-honest model, malicious model allows the corrupt parties to actively deviate from the prescribed computation. We will prove the security of our protocols in the semi-honest model, while some of our proposed protocols also could work in the latter.

2.3 Building Blocks

In this section, we introduce the common building blocks we use throughout this dissertation.

Arithmetic operations. In a linear secret sharing scheme, such as SSS and RSS, a linear combination of secret shared values can be performed locally on the shares without communication. For example, to add or subtract secret shared $[x]$ and $[y]$, each participant can locally compute the addition or subtraction on shares it possesses. A multiplication protocol requires communication, while the costs vary among different protocols. We use the multiplication protocols from [18] for the SSS and from [19] for RSS, where both only communicate one element within one round in three-party setting.

Pseudorandomness generation. The operation $\mathsf{RandBit}$ is used to generate a secret random bit shared among parties in one round. More recently, edaBit is proposed in [20] for generating many random bits $[r_i]$ in the ring $\mathbb{Z}_2$ and all bits are assembled in an integer $[r]$ such that $[r] = \sum_{i=1}^{k} 2^i [r_i]$, where k represents the number of random bits. edaBit uses noticeably lower communication per bit, but the round complexity is logarithmic in k and t.

Comparisons. An equality protocol $[c] \leftarrow \mathsf{EQ}([a],[b])$ takes two private inputs $[a]$ and $[b]$ and returns a secret result. The result will be $[1]$ iff $a = b$ holds and $[0]$ otherwise. EQ is implemented by invoking an equal-to-zero protocol $\mathsf{EQZ}([a]-[b])$. EQZ returns $[1]$ if the input secret value is zero and returns $[0]$ otherwise.

In addition, less-than-zero comparison $\mathsf{LTZ}([a])$ is also widely used. It checks wether its input secret value is less than zero. In SSS, we use truncation to implement LTZ. An truncation protocol $[a'] \leftarrow \mathsf{Trunc}([a], \ell, k)$ [21] takes an ℓ-bit secret value $[a]$ and truncates the k least significant bits, where $[a'] = [a]/2^k$. Regarding LTZ, we truncate $\ell - 1$ bits from the ℓ-bit input so that only the most significant bit (the sign bit) remains. In RSS, LTZ can be implemented using a most-significant-bit protocol MSB [19] that extracts the most significant bit (the sign bit) of a secret value.

Share conversions. Share conversions is also one of the most widely used building blocks in secure computation. $\mathsf{BitDec}([x], \ell, k)$ performs bit decomposition of a ℓ-bit integer $[x]$ and output its k least significant bits in the form of secret shares. In addition, $\mathsf{B2A}([r]_1, k)$ converts a binary share $[r]_1$ into its corresponding arithmetic share $[r]_k$, where we use $[x]_j$ to represent that the share is a j-bit ring or field element.

2.4 Oblivious RAM

An Oblivious RAM (ORAM) is a (probabilistic) RAM machine that its memory access pattern reveals nothing about the program as well as the data on which it is computed. More precisely, the access patterns from executions on two inputs with an equal number of memory accesses should be the same or indistinguishable [22]. With the ability to conceal access patterns, Oblivious RAM is often used as a general-purpose solution for constructing oblivious algorithms. Although we will not directly work on designing ORAM protocols, we introduce this technique here because it could be an alternative of our solutions. ORAM was introduced by Goldreich and Ostrovsky [23, 14, 15] to address the problem of software protection. In the original formulation, a client outsources its private memory to a remote server without revealing any information including access patterns to the server. Great progress has been made in recent years for optimizing the performance of ORAM. A popular type of ORAM is tree-based [24, 25, 26, 27], with constructions capable of achieving communicating $O(\log m)$ blocks of sufficient size per access with constant client storage [25]. As the ORAM designed for the traditional client-server setting cannot be applied to the secure computation environment, we will focus on the ORAM works for secure computation.

In the context of ORAM for secure computation, SCORAM [28] was among the early constructions in the two-party setting. More recently, Floram [29] built on function secret sharing in the two-party setting and improved performance of prior schemes. In the multi-party setting, Keller and Scholl [30] designed and implemented ORAM constructions on top of SPDZ [31], security of which holds in the malicious model with no honest majority with $n \geq 2$ parties. [32]

is a scheme that works only for three parties with custom techniques based on a variant of [24]. Most recently, Jarecki and Wei [33] re-designed Circuit ORAM (3PC ORAM) for the three-party setting by using customized asymptotically bandwidth-optimal and constant-round protocols.

Chapter 3

Related Work

3.1 Private Array Access

Conventional implementations of performing an array access at a private location via a linear scan can be comparison-based or multiplexer-based as we further discuss in Chapter 4. Optimizations to the simple solutions are available in both two-party setting based on garbled circuits (which does not directly apply to the content of this work) and in the multi-party setting. The closest to our work is the construction due to Laud [34] for array read, which is applicable to both Shamir SS and Sharemind framework. The goal of that work was to minimize the online work (which depends on the private inputs), while our goal is to minimize the overall work. As a result, the proposed solution from [34] has large round complexity. It also offers optimizations, the most effective of which is applicable only to the Sharemind framework. We draw a more detailed comparison to the construction from [34] and our solutions in Chapter 4.

Laud [35] proposed efficient protocols for reading and writing elements of an array at private locations in parallel. The solution is based on sorting and for

ℓ parallel read requests to an array of size m has complexity $O((m+\ell)\log(m+\ell))$. Because the solution is non-trivial and was implemented in the Sharemind framework, we are unable to empirically compare its runtime to our constructions designed for Shamir secret sharing. We, however, note that based on the best known oblivious sorting algorithms, this construction will require $O((m+\ell)\log(m+\ell))$ comparisons each of cost $O(k)$ for k-bit integers. Based on our detailed analysis of its possible implementation in our setting, we expect that it might outperform our proposed constructions only when both m and ℓ are large. That is, when the number of parallel invocations ℓ is small or when the array size m is not large (even with a very large number of parallel invocations), we expect our constructions to outperform the solution from [35]. We provide additional comments in Chapter 4.

ORAM can also be used to realize array read or write at a private location, where a client outsources its private memory to a remote server without revealing any information including access patterns to the server. Floram [29] is one of the best performing ORAM constructions among two- and three-party implementations. We will compare performance of our constructions Floram in Chapter 4.

3.2 Private Binary Search

All prior work that securely performs binary search in the context of secure multi-party computation we are aware of [25, 29, 36, 26, 37, 28] focuses on leveraging ORAM operations to hide search patterns, typically invoking a logarithmic number of ORAM accesses per binary search. With the ability to conceal access patterns, ORAM is often used as a general-purpose solution for constructing oblivious algorithms.

Floram [29] reported the best binary search performance among [36, 26, 37, 38], and thus we empirically compare performance of our constructions to that of Floram. Bunn et al.'s ORAM [39] has similar properties to Floram; i.e., it is based on a distributed point function [40], has the same round, communication, and local work complexities as Floram, but uses three parties instead of two. While its performance can be competitive to other ORAM schemes, it comes without an implementation and it is not possible for us to do meaningful comparisons of our solutions to that ORAM, especially because the underlying techniques differ (e.g., it uses oblivious transfer). We can only say that our constant round binary search construction is expected to outperform binary search based on Bunn et al.'s ORAM for small datasets. Its (linear per access) local work will also be the bottleneck when m is large and our optimization from Chapter 5 reduces it from $O(m \log m)$ to $O(m)$ per search, which would match $O(\sqrt{m})$ communication and $O(m)$ local work of our best construction. In addition, 3PC ORAM was reported to have a lower bandwidth than Floram when the dataset size is $< 2^{16}$ and thus is also competitive. Earlier, Gentry et al. [27] proposed an ORAM optimization for binary search for their tree-based construction, which enables binary search to have the asymptotic cost of a single (recursive) ORAM access. This optimization is also applicable to 3PC ORAM and therefore we apply the optimization to 3PC ORAM and include the optimized scheme in the discussion as well. The above constructions were developed for the semi-honest adversarial model, and the only ORAM secure against malicious adversaries we are aware of is by Keller and Yanai [41] which can be based on Circuit ORAM [42] or Path ORAM [25]. Although other recent ORAM constructions exist [43, 44, 45, 46, 47], they are designed for client-server environments and are not applicable to secure computation.

In addition to the above generic solutions, there have been efforts to improve efficiency of oblivious computation by designing oblivious data structures. Toft [48] proposed an oblivious secure priority queue with an amortized cost of $O(\log^2 n)$ per insertion and removal operation. Wang et al. [38] presented oblivious data structures that include priority-queues and stacks building upon techniques of [25] and [27]. Keller and Scholl [30] proposed oblivious data structures using two ORAM schemes [24, 25] and basic secure multi-party operations from [49]. The work realized oblivious arrays, dictionaries, and priority-queues in the multi-party setting using secret sharing. Shi [50] and Jafargholi et al. [51] proposed oblivious priority queues that achieve optimal $O(\log m)$ complexity. However, most of these data structures cannot be directly used for binary search operations in the context of secure multi-party computation. Among these oblivious data structures, the closest to our work is the oblivious dictionary construction for secure computation from [30]. It is capable of performing record search in a binary-search-like manner with communication complexity of $O(m)$, while the best of our protocols has sublinear communication complexity.

Lastly, a recent article by Rao et al. [52] claims to achieve private binary search based on secret sharing. However, the protocol in [52] is not data-oblivious, simply invokes $O(\log m)$ comparisons the way a regular binary search would, and has to disclose the locations used in the comparisons. Furthermore, there are other significant discrepancies in that work. For example, the protocol's complexity is not analyzed in the text, but is said to be $O(m \log m)$ communication in $O(1)$ rounds in the abstract, which disagrees with the protocol itself.

3.3 Private and Accurate Floating-Point Summation

Neal [53] describes algorithms that use a number representation, named by *superaccumulator*, to exactly sum n floating point numbers, which will be convert to a faithfully-rounded floating-point number. Unfortunately, while Neal's superaccumulator representation reduces carry-bit propagation, it does not eliminate it, as is needed for secure computations with few rounds. A similar idea has been used in ExBLAS [54], an open source library for exact floating point computations. Shewchuck [55] describes an alternative representation for exactly representing intermediate results of floating-point arithmetic, but the method also does not eliminate carry-bit propagation in summations; hence, it also does not lead to efficient secure protocols. In addition to these methods, there are a number of other adaptive methods for exactly summing n floating point numbers using various other data structures for representing intermediate results, which do not consider the security or privacy of the data. Further, these methods, which include ExBLAS [54] and algorithms in [56, 57, 58, 59, 60, 61, 62, 63, 64, 65] are not amenable to conversion to secure protocols with constant rounds.

Whereas the integer arithmetic in secure multi-party computation has been extensively investigated, the study of floating-point arithmetics has gradually attracted the attention in the last decade. [66] extended secure computation from integer arithmetic to fixed-point arithmetic for the first time and applied it to linear programming [67, 68]. Franz and Katzenbeisser [69] proposed a solution, based on homomorphic encryption and garbled circuits, for floating-point arithmetics in the two-party setting with no implementation or perfomance results. Aliasgari et al. [70] designed a set of protocols for basic floating-point operations based on Shamir secret sharing and developed several advanced operations such as logarithm, square root and exponentiation of floating-point numbers. Their solution was improved and extended for other settings and applications [71, 72] later. Dimitrov et al. [73] proposed two sets of protocols based on new representations, but failed to follow IEEE 754 standard for efficiency reasons. In addition to the above works on improving efficiency of unary/binary floating-point operations, Catrina [74, 75] proposed and improved several multi-operand operations such as sum and dot-product. Because their solutions are still based on traditional floating-point addition, the round-off errors will be introduced inevitably in each addition operation.

Chapter 4

Private Array Access

In this chapter, we study performance improvements to private array access in secure multi-party computation based on secret sharing. We present two optimized protocols for accessing an element of an array at a private index. The former of our constructions are based on Shamir secret sharing and could work with any number of parties. The latter uses 2-out-of-2 additive secret sharing in the three-party setting with honest majority, but offers superior performance compared to general constructions. To be compatible with computation based on Shamir secret sharing, we also provide conversion procedures to convert between the two representations. We implement the presented constructions in the setting with three computational parties and show that they offer attractive performance in both LAN and WAN settings.

We first define the problem before decomposing our constructions. Assume that we are given an array of m (private or public) elements $a_0, \ldots, a_{m-1}$ and would like to retrieve the element a_j at a private index j. We will abstract the array access functionality as a standalone building blocks. The input private array and private index as well as the output will be in form of secret sharing.

Protocol 1 $[b] \leftarrow \mathsf{OriginalArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$

```
1: for $i = 0, \ldots, m-1$ in parallel do
2:     $[c_i] \leftarrow \mathsf{EQ}([j], i)$;
3: end for
4: $[b] \leftarrow \sum_{i=0}^{m-1} [c_i] \cdot [a_i]$;;
5: return $[b]$;
```

4.1 General Construction

Conventional implementations of this functionality via linear scan include (i) privately comparing j to every integer in the range $[0, m-1]$ to compute m bits and computing the dot product of the resulting bits and the array elements and (ii) bit-decomposing the index and using a multiplexer to retrieve the desired element. The latter approach was implemented in the PICCO compiler [76] using conventional Shamir secret sharing arithmetic, while the former was later shown to be slightly faster for this setting [77]. Our starting point for improving the general solution was the first traditional approach above where we privately compare j to each position of the array and retrieve the element for which the result of the comparison was true. If we let EQ denote the operation of privately comparing two integers for equality with at least one of them being private, this operation can be represented as shown in Protocol 1. This computation is written to take an array of private elements as its input, but when the elements are public, the computation proceeds similarly.

To optimize performance of this operation, our first observation stems from the fact that j is compared to all index values between 0 and $m-1$ and, as a result, part of the computation might be redundant. To determine whether this might be the case, let us look at the details of the secure equality operation EQ. The most efficient constant-round equality protocol in our setting is due to Catrina and de Hoogh [21], which we specify in Protocol 2. It proceeds by

Protocol 2 $[b] \leftarrow \mathsf{EQZ}([a], k)$

```
1: $([r'], [r], [r_{k-1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(k, k)$;
2: $c \leftarrow \mathsf{Open}([a] + 2^k [r'] + [r])$;
3: $(c_{k-1}, \ldots, c_0) \leftarrow Bits(c, k)$;
4: for $i = 0, \ldots, k-1$ in parallel do
5:     $[d_i] \leftarrow c_i + [r_i] - 2c_i[r_i]$;
6: end for
7: $[b] \leftarrow 1 - \mathsf{KOr}([d_{k-1}], \ldots, [d_0])$;
8: return $[b]$;
```

comparing a single private integer a to 0 and is denoted by EQZ. To compare a to b, one would enter their difference $a - b$ as the input to the protocol. The algorithm also takes a second argument, which is the bitlength k of the first operand a.

Here, the operation $\mathsf{PRandM}(k, \alpha)$ assumes that we work with k-bit integers and generates a $(k + \rho)$-bit random integer for a statistical security parameter ρ, the α least significant bits of which are available in the bit-decomposed form. The returned result is the shares of α random bits $r_0, \ldots, r_{\alpha-1}$, α-bit $r = \sum_{i=0}^{\alpha-1} 2^i r_i$, and $(k + \rho - \alpha)$-bit integer r'. The Open function reveals the value of its private argument. $Bits(c, \alpha)$ simply returns the α least significant bits of its public argument c. Lastly, KOr computes the k-ary OR of its k private input bits.

This operation hides the value of a by adding large random $2^k \cdot r' + r$ to it and opening the sum.[1] Because the bits of r are available (as r_0 through r_{k-1}), the remaining computation can efficiently compute the bits of a (in step 4) and consequently test whether at least one of them is 1 (in step 5) using k-ary OR of k bits. The cost of this operation is dominated by PRandM which contributes k (parallel) interactive operations, while KOr costs $4\log(k)$ and Open costs 1 inter-

[1]Note that the original EQZ in [21] was designed for signed k-bit integers. Because of that, it also specified to add 2^{k-1} to the value being opened, to move the input into the positive range. In our application, we use only positive values and let the entire k-bit space be occupied by them. For that reason, one should omit adding that constant.

active operation, respectively. The overall number of rounds is 4.

When we compare private j to all possible indices i in the set, we invoke EQZ on inputs $j - i$, the adjacent values of which differ by 1. This introduces significant inefficiencies because expensive generation of random bits is invoked for each i to protect related values with a known difference. This means that, instead of generating independent random bits for each $j - i$ via a new call to PRandM, we could execute this function once, protect j using the random values as in step 2 above, and open this protected value as c. Given the protected value c of j, we can then form protected values of $j - i$ by computing $c - 0, c - 1, \ldots, c - (m - 1)$ if we assume that i ranges from 0 to $m - 1$.

This optimization reduces the cost of array read from $m(\log m + 4 \log\log m + 1) + 1$ interactive operations in 5 rounds to $4m \log\log m + \log m + 2$ in 5 rounds. Alternatively, we could use a simple tree-like implementation of KOr with $\log m - 1$ interactive operations in $\log\log m$ rounds, which makes the complexity of $\mathsf{ArrayRead}$ be $m(\log m - 1) + \log m + 1$ in $\log\log m + 2$ rounds.

This, however, still appears redundant because the bits of v, and consequently bits d provided as input into the k-ary OR in step 3(d), are often reused from one loop iteration i to another. For example, we know that c and $c - 1$ are going to differ in their least significant bits, but a number of most significant bits might be the same. Also, because the bitlength of j is $\log m$, we know that most of (or all) possible combinations of $\log m$ bits will be used in KOr across all i. In other words, for any given v, its ith bit will be either the ith bit of c or its complement, and most of all possible $2^{\log m}$ combinations of bits will be used across all is to form vs. To combat this inefficiency, we design a new efficient mechanism for computing OR of all possible combinations of bits and then incorporate it in the private lookup protocol.

Protocol 3 $\langle [b_0], \ldots, [b_{2^k-1}] \rangle \leftarrow \mathsf{AllOr}([d_{k-1}], \ldots, [d_0])$

```
1: if $k = 1$ then
2:     return $\langle [d_0], 1 - [d_0] \rangle$;
3: end if
4: $\ell \leftarrow \lfloor k/2 \rfloor$;
5: $[u_0], \ldots, [u_{2^\ell - 1}] \leftarrow \mathsf{AllOr}([d_{\ell-1}], \ldots, [d_0])$;
6: $[v_0], \ldots, [v_{2^{k-m}-1}] \leftarrow \mathsf{AllOr}([d_{k-1}], \ldots, [d_\ell])$;
7: for $i = 0, \ldots, 2^{k-\ell} - 1$ and $j = 0$ to $2^\ell - 1$ in parallel do
8:     $[b_{2^\ell i + j}] \leftarrow [v_i] + [u_j] - [v_i] \cdot [u_j]$;
9: end for
10: return $\langle [b_0], \ldots, [b_{2^k} - 1] \rangle$;
```

Our algorithm for computing ORs of bits uses a divide-and-conquer approach, where we split the original size into two halves, recurse on each half, and then assemble the result. It is denoted as AllOr and given in Protocol 3. On input k bits d_i, it computes 2^k k-ary ORs of the form $\bigvee_{i=0}^{k-1} c_i$, where c_i is either d_i or its complement $\neg d_i$.

To integrate this solution into our array read protocol, we apply AllOr to the bits r_is computed in step 1 of the last variant of ArrayRead and, as before, reveal the value of j protected by r; let the $\log m$ least significant bits of the protected value be denoted by c'. The intuition is now that the computed k-ary ORs correspond to all possible k-ary ORs over all k-bit integers "shuffled" based on the value of r and the only OR that evaluates to 0 will be at position r. This means that if we would like to know whether, e.g., $j = 0$, we need to test whether $c' = r$ or, equivalently, whether the c'th position in the array of k-ary ORs corresponds to 0. Similarly, for testing whether $j = i$, we test whether $c' = r + i$ (or, equivalently, whether $r = c' - i$) and retrieve the $(c' - i)$th value in the returned array. Lastly, because we need a single OR evaluate to 1 with the remaining values being 0, we complement the result of the AllOr operation. (Note that the original implementation of EQZ from [21] computes $c \oplus r$ instead

Protocol 4 $[b] \leftarrow \mathsf{ArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$

```
1: $([r'], [r], [r_{\log m-1}], \ldots, [r_0]) \leftarrow \mathsf{PRandM}(\log m, \log m)$;
2: $\langle [b_0], \ldots, [b_{2^{\log m}-1}] \rangle \leftarrow \mathsf{AllOr}([r_{\log m-1}], \ldots, [r_0])$;
3: for $i = 0, \ldots, 2^{\log m} - 1$ in parallel do
4:     $[b_i] = 1 - [b_i]$;
5: end for
6: $c \leftarrow \mathsf{Open}([j] + 2^{\log m}[r'] + [r])$;
7: $c' \leftarrow c \bmod 2^{\log m}$;
8: $[b] \leftarrow \sum_{i=0}^{m-1} [b_{c'-i \bmod 2^{\log m}}] \cdot [a_i]$;
9: return $[b]$;
```

of $c - r$ prior to calling KOr using a more complex logic to show correctness of the algorithm, but the same approach does not work in our case.) We obtain the Protocol 4.

To demonstrate security, we note that all instructions are input-independent and follow a similar structure to that of EQZ from [21]. All steps operate on shares except step 4, in which the value of c is revealed. The value of c corresponds to private j protected by a random value at least ρ bits longer than j. This means that the probability that any information is revealed about j is negligible in the security parameter ρ and is therefore acceptable. This implies that we are able to simulate the adversarial view without access to the inputs, as is formally shown in the Section 4.3.

4.2 Custom Three-Party Construction

In this section, we provide a second construction which is designed to work only with $n = 3$ parties using custom computation, but offers superior performance compared to the general construction. Our second construction uses 2-out-of-2 additive secret sharing, which means that if we would like to use it together with a standard SS framework such as Shamir SS, we need to provide procedures for converting between the two representations. This is what we do at the end of this section as well.

In what follows, we use notation $[\![x]\!]$ to denote that the value of $x \in \mathbb{F}$ is secret shared using 2-out-of-2 additive secret sharing. We note that this solution works over any finite ring, which has performance benefits such as using native hardware implementations of arithmetic in $\mathbb{Z}_{2^k}$ for some k. For the purposes of this work, we let computation to be over a finite field to be compatible with other constructions we propose.

Because in this representation the shares are held by two parties out of three, for concreteness of the presentation, we let the notation include the parties holding the shares. Thus, we use $[\![x]\!]_{p_1p_2}$ to indicate that the value is split between parties $p_1, p_2 \in [1, 3]$ with $p_1 \neq p_2$. For example, we might use $[\![x]\!]_{12}$. Then notation $[\![x]\!]_{p_1}$ and $[\![x]\!]_{p_2}$ denotes the individual shares when x is secret shared as $[\![x]\!]_{p_1p_2}$.

In our construction, the data set is originally additively shared between parties 1 and 2 (i.e., we have $[\![a_0]\!]_{12}, \ldots, [\![a_{m-1}]\!]_{12}$). The private index j can be secret-shared using any linear SS scheme and for simplicity we assume it is shared using Shamir SS as $[j]$. The intuition behind our solution is that the data set is rotated by a private number of positions and the value of j gets adjusted by that value. Then the parties who do not have information about the entire amount of rotation learn the modified value of j and read the element at that position. To implement this idea, we need to be careful to ensure that reading the element is performed on the shares to prevent any single party from having access to the read element. And at the same time we must enforce that the parties with clear text access to the modified j do not know by which value j was modified from its original value.

To realize this intuition, we instruct parties 1 and 2 to rotate their shares of the data set by random amount $r_1 \in \mathbb{Z}_m$ known only to the two of them. Next, party 1 re-shares its shares of the data set between parties 2 and 3, which makes the rotated data set to be shared between these two parties. Now parties 2 and 3 again rotate the shared data set by random amount r_2 known only to the two of them, after which party 2 re-shares its data set shares among parties 1 and 3. At this point, the data set has been rotated by $r_1 + r_2$ and is shared between parties 1 and 3, neither of whom knows the value of $r_1 + r_2$. Thus, we open $h = (j + r_1 + r_2) \bmod m$ to parties 1 and 3 who consequently retrieve the element at position h in their data sets and return their share as the output.

In our solution, we propose that the parties generate r_1 and r_2 non-interactively using a shared seed to a pseudo-random generator. That is, parties 1 and 2 share key k_{12}, while parties 2 and 3 share key k_{23}. Because generation of r_1 and r_2 is a one-time cost independent of the data set size, any other suitable mechanism for agreeing on these values will work as well (e.g., if one wants to maintain information-theoretic security of the protocol). The computation then proceeds as described in Protocol 5.

This computation is dominated by communicating $4m$ elements in two rounds, i.e., similar to that of executing m multiplications in parallel. There might also be communication for computing h or $h \bmod m$ depending on the underlying SS scheme. In particular, if h is secret-shared using additive SS in $\mathbb{Z}_m$, no additional communication is needed. That is, with additive SS, we would need to modify only one of the shares to perform addition of r_1 or r_2, and the opened value will be in $\mathbb{Z}_m$, as desired, because the arithmetic is in $\mathbb{Z}_m$. With a different type of SS such as Shamir SS, the parties need to update h and re-share its value across all parties with fresh randomness. Similarly, when computation is not

Protocol 5 $[b] \leftarrow \mathsf{3PartyArrayRead}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [j])$

```
1: Parties 1 and 2 agree on random $r_1 \in \mathbb{Z}_m$ and locally rotate their shares as $\langle [\![a_{r_1}]\!]_p, \ldots, [\![a_{m-1}]\!]_p, [\![a_0]\!]_p, \ldots, [\![a_{r_1-1}]\!]_p \rangle \leftarrow \langle [\![a_0]\!]_p, \ldots, [\![a_{m-1}]\!]_p \rangle$, where $p \in [1, 2]$, and also let $[h] \leftarrow [j] + r_1$.
2: Party 1 randomly generates $s_i \in \mathbb{F}$ for $i \in [0, m-1]$ and sends $\langle s_0, \ldots, s_{m-1} \rangle$ to party 2, who consequently sets $[\![a'_i]\!]_2 = [\![a_i]\!]_2 + s_i$ for $i \in [0, m-1]$.
3: Party 1 sets $[\![a'_i]\!]_3 = [\![a_i]\!]_1 - s_i$ for $i \in [0, m-1]$ and sends $\langle [\![a'_0]\!]_3, \ldots, [\![a'_{m-1}]\!]_3 \rangle$ to party 3.
4: Parties 2 and 3 agree on random $r_2 \in \mathbb{Z}_m$ and locally rotate their shares as $\langle [\![a'_{r_2}]\!]_p, \ldots, [\![a'_{m-1}]\!]_p, [\![a'_0]\!]_p, \ldots, [\![a'_{r_2-1}]\!]_p \rangle \leftarrow \langle [\![a'_0]\!]_p, \ldots, [\![a'_{m-1}]\!]_p \rangle$ and let $[h] \leftarrow [h] + r_2$.
5: Party 2 randomly generates $s'_i \in \mathbb{F}$ for $i \in [0, m-1]$ and sends $\langle s'_0, \ldots, s'_{m-1} \rangle$ to party 3, who consequently sets $[\![a''_i]\!]_3 = [\![a'_i]\!]_3 + s'_i$ for $i \in [0, m-1]$.
6: Party 2 sets $[\![a''_i]\!]_1 = [\![a'_i]\!]_2 - s'_i$ for $i \in [0, m-1]$ and sends $\langle [\![a''_0]\!]_1, \ldots, [\![a''_{m-1}]\!]_1 \rangle$ to party 1.
7: The value of $h$ mod $m$ is opened to parties 1 and 3 who set $[\![b]\!]_p = [\![a''_h]\!]_p$ for $p \in [1, 3]$.
8: return $[\![b]\!]_{13}$.
```

in $\mathbb{Z}_m$, computing h mod m is needed prior to opening the value. For example, with SSS, one might invoke efficient Mod protocol from [21] (integer division with public divisor). This is a one-time operation of cost at most $O(\log m)$ and does not have a significant impact on the performance of the overall protocol.

To show security in the three-party setting with a single corrupt party, we argue that the data set remains information-theoretically protected from any participant. In particular, it is always secret-shared among two parties. Furthermore, the value of j is also information-theoretically protected from the parties if r_1 and r_2 are chosen randomly (and otherwise is computationally protected). Thus, it can be shown that the simulated view with no access to real data is indistinguishable from a real run of the protocol. We provide a formal proof in the Section 4.3.

Lastly, to permit this construction to be used in conjunction with SSS techniques, we next provide conversion procedures to and from 2-out-of-2 additive

Protocol 6 $[\![a]\!]_{12} \leftarrow \mathsf{S2A}([a])$

```
1: $[r] \leftarrow \mathsf{PRSS}()$;
2: $[d] \leftarrow [a] - [r]$;
3: Open $r$ to party 1 who sets $[\![a]\!]_1 = r$;
4: Open $d$ to party 2 who sets $[\![a]\!]_2 = d$;
5: return $[\![a]\!]_{12}$;
```

Protocol 7 $[a] \leftarrow \mathsf{A2S}([\![a]\!]_{12})$

```
1: Party 1 creates Shamir secret shares of $[\![a]\!]_1$ and distributes them among the parties;
2: Party 2 creates Shamir secret shares of $[\![a]\!]_2$ and distributes them among the parties;
3: $[a] = [[\![a]\!]_1] + [[\![a]\!]_2]$;
4: return $[a]$;
```

secret sharing and SSS over the same field $\mathbb{F}$. The cost of converting a field element to or from additive SS is that of communicating two field elements. This means that for a read operation, the cost of converting the inputs and outputs of ArrayRead is communicating about $2m$ elements. The cost of the conversion, however, can be amortized among multiple operations if these operations are repeatedly called on the same data set without other intermediate operations.

We start from the SSS to additive SS conversion, which proceeds as shown in Protocol 6. If we assume that PRSS can be realized non-interactively as previously described, reconstructing r in step 3 and d in step 4 involves communicating one field element each. That is, party 2 or 3 sends its share of r to party 1 in step 3, from which party 1 recovers r. Thus, the cost is communicating 2 field elements in 1 round.

The conversion from additive SS to SSS is shown in Protocol 7. Steps 1 and 2 can be accomplished by communicating a single field element each in the computational setting similar to the approach taken in [18] (otherwise, the cost is 2 field elements in the information-theoretic setting). That is, party 1 shares a secret key with another party for generating that party's share as a pseudo-random value. The remaining shares are computed to be consistent with the pseudo-random share and the value of $[\![a]\!]_1$, which requires communication of a single share. Thus, the protocol's cost is communicating 2 field elements in 1 round.

4.3 Security Proof

In this section, we demonstrate the security of the protocols proposed in this chapter based on Definition 1.

Theorem 1. *The* ArrayRead *protocols of section 4.1 are t-private for any $t < n/2$ assuming security of sub-protocols* PRandM *and multiplication.*

Proof. As in definition 1, let I denote the set of corrupt parties for any t $< n/2$. We build a simulator S_I, which simulates the view of the parties in I in the ideal model without access to private data. Note that in the case of ArrayRead operations, each corrupt party contributes no private input and learns no private output. Thus, the simulator needs to construct their view without access to any private data. Our simulator S_I proceeds as follows:

- In step 1 of the protocol, S_I invokes the simulator for PRandM, which simulates the view of the corrupt parties.
- During the computation of AllOr, S_I invokes the simulator for each multiplication of secret-shared values called by that protocol.
- During step 4, S_I broadcasts shares of c on behalf of honest parties such that all shares reconstruct to random c of the desired length. In particular, S_I can wait for the corrupt parties to transmit their shares and fix a desired

c (which corresponds to a share evaluated at point 0). These $t+1$ values define a unique degree-t polynomial, which S_I reconstructs through polynomial interpolation and consequently computes and broadcasts shares on behalf of the honest parties.

Note that S_I is not required to wait for the corrupt parties to transmit their shares. Instead, S_I can use its access to the corrupt parties' inputs in the protocol, including randomness that they use throughout the computation. Based on that information and the messages that each corrupt party receives prior to step 4, S_I can correctly compute the share that each party in I is to broadcast in step 4. From that point, it creates shares of c on behalf of the remaining parties as specified above.

- In step 6 of the protocol, S_I invokes the simulator for the multiplication operations (or the simulator for the dot product if available as a separate primitive).

Now we need to compare the view that S_I produces with the view of the corrupt parties in the real protocol execution. Notice that most steps invoke simulators for the respective building blocks which we assume secure. This means that the views produced by those simulators are indistinguishable from the parties' views in the real protocol execution. The only value that S_I produces on its own is c. Because S_I samples the value of c from the same distribution as the protocol does, the only difference between the real and simulated values can come from the fact that adding j to the random value prior to its opening in the real protocol execution may result in overflow and thus be distinguishable. The probability of this happening, however, is negligible in the statistical security parameter ρ and is beyond the adversarial control. Therefore, the real and simulated views are statistically indistinguishable, which completes the proof. □

Theorem 2. *Custom three-party* ArrayRead *of section 4.2 are 1-private.*

Proof. We prove that our custom three-party array read protocols are secure in the presence of a single corrupt party based on definition 1. We consider corruption of party 1, 2, and 3 in turn.

Party 1 is corrupt. First, let party 1 be corrupt, and we build the corresponding simulator S_1. As before, the protocol has no private inputs or outputs for any party and therefore the simulator is not given any private values. Our S_1 proceeds as follows:

- If updating $[h]$ in steps 1 and 4 of the protocol or computation of $[h \bmod m]$ in step 7 involves interaction, S_1 invokes simulators for the corresponding computation.
- In step 2, S_1 receives shares $s_0, \ldots, s_{m-1}$ from party 1 on behalf of party 2.
- In step 6, S_1 generates random shares $[\![a''_0]\!]_1, \ldots, [\![a''_{m-1}]\!]_1$ and sends them to party 1 on behalf of party 2.
- In step 7, S_1 participates in opening shares of $h \bmod m$ on behalf of party 3. To do so, S_1's behavior depends on the choice of the secret sharing used. For additive sharing over $\mathbb{Z}_m$, S_1 simply generates random shares in $\mathbb{Z}_m$ and sends them to party 1. For Shamir SS over $\mathbb{F}$, S_1 computes the remaining shares for its choice of random $h \in \mathbb{Z}_m$ taking into account its knowledge of party 1's share. The mechanism is the same as what S_I used in the proof of Theorem 1.

To analyze party 1's view, we see that party 1 has access to 1 out of 2 shares for each element a_i at step 1 and later receives a new share of each a_i after its re-sharing in step 6. Party 1 also has access to r_1 and obtains the value of $h \bmod m = (j + r_1 + r_2) \bmod m$ in step 7. In the simulated view, party 1 receives random shares in step 6 and a random $h \in \mathbb{Z}_m$ in step 7. Now notice that the shares of a_is that party 1 receives in step 6 are distributed uniformly in $\mathbb{F}$ and are therefore distributed identically to the values that the party receives in the simulated view. Also, because the values that $(r_2 + j) \bmod m$ takes is distributed as a random (or pseudo-random) element of $\mathbb{Z}_m$ in the real execution, $h = (r_1 + r_2 + j) \bmod m$ in the real execution is indistinguishable from random $h \in \mathbb{Z}_m$ in the simulated execution.

Party 2 is corrupt. We next construct simulator S_2 for the case of corrupt party 2. In this case, the simulator is simple: besides accepting messages from party 2 in steps 5 and 6, S_2 only needs to simulate the message party 1 sends to party 2 in step 2. To do so, S_2 randomly generates shares $s_0, \ldots, s_{m-1}$ and sends them to party 2. Also, if computation of h and $h \bmod m$ in steps 1, 4, and 7 is interactive, S_2 would need to invoke the corresponding simulators.

From the above, it is clear that the simulated view is identical to the real view, i.e., values $s_0, \ldots, s_{m-1}$ are distributed identically in both views. Party 2 also has access to r_1 and r_2, which are independent of private data.

Party 3 is corrupt. Lastly, we consider party 3 to be corrupt and construct simulator S_3 which works as follows:

- As before, if the computation of h and $h \bmod n$ is interactive, S_3 invokes the corresponding simulators.
- S_3 generates and sends random shares $[\![a'_0]\!]_3, \ldots, [\![a'_{m-1}]\!]_3$ to party 3 in step 3.
- S_3 generates and sends random shares $s'_0, \ldots, s'_{m-1}$ to party 3 in step 5.
- In step 7, S_3 participates in opening a random value $h \in \mathbb{Z}_m$ in the same way it was accomplished by S_1.

To analyze the real and simulated views, we first note that the shares s'_i communicated to party 3 are chosen and distributed identically in both views. Furthermore, while S_3 used random values for $[\![a'_i]\!]_3$s, but in the real protocol execution these shares were computed differently, the real execution shares were protected by uniformly random values not accessible to party 3 and thus both have uniformly random distributions. Lastly, party3 learns random $h \in \mathbb{Z}_m$ in the simulated view, while in the real execution the opened value is $(j + r_1 + r_2) \bmod m$ with r_2 known by party 3, the views are still indistinguishable. This is because r_1 is a random or pseudo-random element of $\mathbb{Z}_m$ and $(r_1 + j) \bmod m$ has the same distribution is that of r_1.

We conclude that our custom three-party construction is secure in the presence of a single semi-honest party. $\square$

4.4 Performance Evaluation

We have implemented the proposed array read and multiplication operations in C using single invocation as well as batched execution. Because the custom 3-party array read is asymmetric, our batched execution of that protocol used 3 threads, each taking on the role of a different party and with the workload divided evenly across the threads. We used the GNU Multiple Precision Arithmetic Library (GMP) [78] for field arithmetic and executed SSS constructions within the PICCO compiler framework [76]. We also execute original array read with private index and multiplication operations as previously implemented in PICCO. All of our protocols are evaluated in the three-party setting with a single corrupt party. For comparison, we also include runtimes of two-party Floram CPRG [29] using their implementation from [79]. This is one of the best perform-

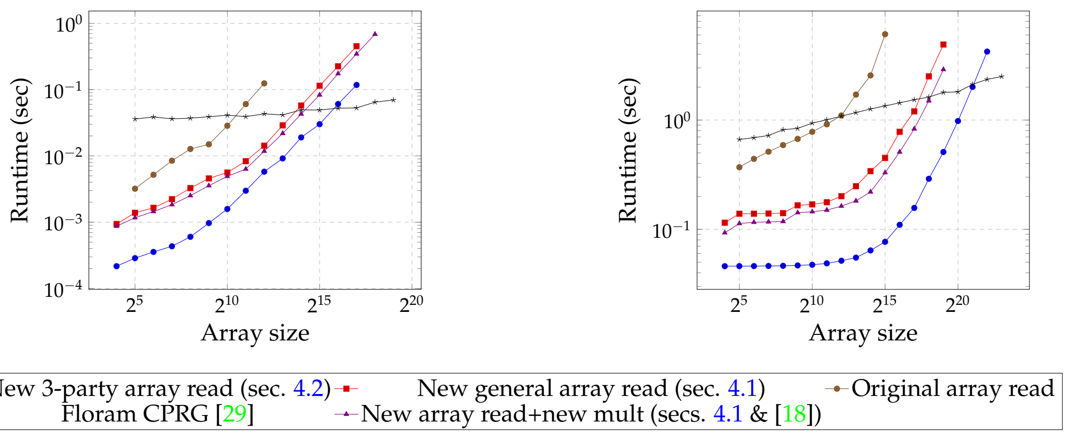

Figure 4.1: Performances of array read with private index on a LAN (left) and WAN (right).

ing ORAM constructions among two- and three-party implementations and its performance tells us at which array sizes ORAM techniques outperform linear scan. Note that ORAM use might involve additional overhead beyond what we report, e.g., for initializing ORAM or converting between different data representations.

We provide experiments in the LAN and WAN configurations. Our LAN experiments were carried out on identical machines with a 2.1GHz processor connected via 1Gbps Ethernet with one-way latency of 0.15ms. Our WAN experiments used local machines and one remote machine with a 2.4GHz processor. One-way latency between the remote and local machines was 23ms. We note that although the machine configurations were slightly different, we do not expect this to introduce inconsistencies in the experiments. In particular, computation time is dictated by the slower machines which do not change across our experiments and the introduced slowdown is attributed to the longer round-trip times and lower bandwidth in WAN experiments. All experiments except Floram used a single core and all experiments (except Floram) were executed over a 64-bit finite field and averaged over 100 executions.

	Original array read				New array read (sec. 4.2)				New 3-party array read (sec. 4.1)			
	1	10	10^2	10^3	1	10	10^2	10^3	1	10	10^2	10^3
2^4	0.0022	0.0058	0.025	0.24	0.00087	0.0021	0.0095	0.096	0.00022	0.00039	0.00084	0.0069
2^7	0.0085	0.028	0.26	2.33	0.0018	0.0071	0.044	0.46	0.00043	0.00075	0.0057	0.048
2^{10}	0.029	0.28	2.9	27.2	0.0049	0.028	0.29	2.98	0.0016	0.0039	0.036	0.37
2^{13}	0.27	2.77	28.8	276	0.022	0.22	2.2	22.5	0.0092	0.027	0.28	3.21
2^{16}	2.67	27.8	267	2,689	0.174	1.75	17.6	180	0.061	0.23	2.41	26.1

Table 4.1: Performance of array read with private index for varying array sizes and in batches of varying size (from 1 to 10^3) on a LAN in seconds. General constructions used (3, 1) setting.

Performance of array read is shown in Figures 4.1 in both LAN and WAN settings. We see that the custom three-party construction significantly outperforms other options and further improvements are possible with parallel execution (which we discuss later in this section). We also see that linear scan constructions outperform ORAM-based solutions for arrays of size up to 2^{16} in the LAN setting and up to 2^{21} in the WAN setting. The figure also shows the difference in the performance of our general array read protocol using the original multiplication protocol as implemented in PICCO (with 6 field elements communicated per multiplication) and the new multiplication protocol from [18](with 3 field elements per multiplication).

We further note that a flatter curve in Figure 4.1 indicates that round complexity or another portion of the computation sub-linear in the array size (Floram or linear scans for arrays of small sizes in the WAN setting) is the bottleneck. A steeper curve indicates that work linear in the array size (e.g., $O(m)$ communication in the case of linear scans) is the bottleneck.

We also provide measurement results for parallel execution of array read in Table 4.1. We compare the original PICCO multiplexer-based implementation with (i) our new general array read with new multiplication from [18] and (ii) our custom 3-party array read from section 4.2. Substantial runtime reduction over single execution is observed for arrays of relatively small size and improvement is present for all sizes in the case of the custom 3-party array read. The largest difference between the original and our general solution is by a factor of 16 with array size of 2^{16} and batch size of 10 and the largest difference between the original and our custom 3-party solution is by a factor of over 120 for the same configuration.

We also attempted to compare performance of our array read protocols with that of the parallel array access protocols from [35], which is designed to do many simultaneous read or write operations in a batch. Because the protocols were implemented in the Sharemind setting using different underlying arithmetic and building blocks, a direct comparison is not possible. Furthermore, the results were plotted in the log-scale and therefore extracted precise numbers is difficult and we can only offer approximate insights. The experiments in [35] were run on a cluster of three 12-core 3GHz computers on a 1Gbps LAN. Our conclusion was that our solutions significantly outperform that from [35] when either the array size is rather small or when the number of parallel invocations is low (or both). For example, performing 5 parallel reads from an array of size 5 costs $>$ 10ms in [35], which is 5 and 25 times slower than executing 10 reads from an array of size 2^4 in our general and 3-party solutions, respectively (recall that Sharemind-based implementation in [35] also works only with three parties). Performing 100 and 1 simultaneous reads from an array of size of 100 takes around 100ms and 50ms, respectively, which is 2 and respectively $>$ 25 times slower than the same number of reads from an array of 2^7 in our general protocol, and $>$ 17 and 115 times slower than our 3-party protocol. Executing a single read is always faster in our solution for all available data points by a significant amount (1–3 orders of magnitude). Where the construction of [35] can offer advantage over our solutions is when both the number of parallel reads and the array size are large. The largest advantage we can observe for 1000 simultaneous reads from an array of size 2^{16}, where our general construction is slower than the results from [35] by about a factor of 18 while our three-party construction is only slower by about 2.5 times.

Chapter 5

Private Binary Search

We are interested in the computation associated with a binary search when we search a private sorted dataset using a private key. In more detail, we are given a data set consisting of m private items $a_0, \ldots, a_{m-1}$, possibly consisting of multiple fields, one of which is the key field. The items are sorted by their keys. On input private search value b, we want to retrieve the element a_i whose key is equal to the searched key b or the smallest one that bigger than b. We also consider update operations where instead of retrieving an element, we update the chosen item.

In this chapter, we initiate the study of binary search protocols in secure multi-party computation, where on input a private sorted dataset and a private search key, one retrieves or updates the closest element that matches the search. We first explore the existing methods and possible improvements for performing an private binary search. Next, we develop a suite of protocols with different properties and structure and further combine them to into hybrid solutions to improve performance and asymptotic complexity. Our fastest binary search protocol uses $O(\sqrt{m})$ communication for a dataset of size m. It is noted that the protocols proposed in this chapter also apply in the malicious setting. We will discuss it in the Section 5.5.

Our performance evaluation demonstrates that our solutions outperform existing ORAM constructions for dataset sizes up to a billion, even after optimizations to improve performance of ORAM schemes specifically in the context of binary search. We hope that this work will inspire others to work on this topic and make further progress in binary search protocols and in particular designing sublinear-cost constructions.

5.1 Initial Construction

5.1.1 Linear-Comparison based Construction

Recall that we are given a set $[a_0], \ldots, [a_{m-1}]$ sorted in the increasing order by their keys (we assume no duplicate key values) and would like to search for value $[b]$. In a linear-scan-type solution, we compare b to each $a_i.key$ by executing a protocol for $[b] \le [a_i.key]$, which results in an array of m bits. We then search for the position j in the array where the bits switch from 0s to 1s and return the corresponding item a_j.

A possible comparison-based binary search CompBS is given as Protocol 8, with linear times of less or equal comparisons LE being called on k-bit arguments. In this protocol, after performing the comparisons on line 2, we compute d_i as $c_i \wedge \neg c_{i-1}$. Note that $d_j = 1$ indicates that $b > a_{j-1}.key$ and $b \le a_j.key$ and thus a_j is the desired element that we want to retrieve. Because there is only one index j such that $d_j = 1$, we retrieve the corresponding element a_j using computation $z = \bigvee_{i=0}^{m-1}(d_i \wedge a_i) = \sum_{i=0}^{m-1} a_i \cdot d_i$ on line 9.

Protocol 8 $[z] \leftarrow \mathsf{CompBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
1: for $i = 0, \ldots, m-1$ in parallel do
2:     $[c_i] \leftarrow \mathsf{LE}([b], [a_i.key])$;
3: end for
4: $[d_0] = [c_0]$;
5: for $i = 1, \ldots, m-2$ in parallel do
6:     $[d_i] \leftarrow [c_i] \cdot (1 - [c_{i-1}])$;
7: end for
8: $[d_{m-1}] = 1 - [c_{m-2}]$;
9: $[z] \leftarrow \sum_{i=0}^{m-1} [a_i] \cdot [d_i]$
10: return $[z]$;
```

CompBS is written to return a single element of the set even if the searched value is outside of the key range for all a_is. In particular, if $b \leq a_0.key$, a_0 is returned; if $a_{i-1}.key < b \leq a_i.key$ for some i, a_i is returned; and if $b > a_{m-1}.key$, a_{m-1} is returned (this means that a_{m-1} is returned if b is greater than the key of a_{m-2}). However, any desired variant of the algorithm can be easily supported.

Because round complexity is crucial to performance of secure protocols based on secret sharing, our protocol is written to run as many interactive operations in parallel as possible. The overall cost is dominated by m LE comparisons. Detailed costs of this and other protocols are provided in Table 5.1 assuming classical building blocks. In particular, the dot product costs 1 interactive operation with communication independent of the input size (e.g., implemented as a generalization of multiplication from [80]), and LE comparisons are instantiated as in [21, 81] on k-bit inputs $a_i.key$ and b, which cost $4k-2$ interactive operations in 4 rounds, one of which can be precomputed.[1] Security analysis of this and other protocols is provided in section 5.4.

[1]Note that there are newer comparison protocols such as those that use edaBits for random bit generation [20] of noticeably lower communication, but higher round complexity. As will be seen later, our final solution uses a relatively small number of comparison operations and a lower-round version of the comparison operation is preferred.

Protocol 9 $[z] \leftarrow \mathsf{OramBS}(\mathsf{ORAM}([a_0], \ldots, [a_{m-1}]), [b])$

```
1: $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
2: $[d] = \lfloor m/2 \rfloor$;
3: $[p] \leftarrow [c] \cdot ([a_{\lfloor m/2 \rfloor}] - [a_{m-1}]) + [a_{m-1}]$;
4: **for** $i = 0, \ldots, \log(m) - 1$ **do**
5:     $[d] = [d] + (1 - 2[c]) \cdot \lfloor m/2^{i+2} \rfloor$;
6:     $[a] \leftarrow \mathsf{ORAMR}([d])$;
7:     $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
8:     $[p] \leftarrow [c] \cdot ([a] - [p]) + [p]$;
9: **end for**
10: **return** $[z] = [p]$;
```

5.1.2 ORAM-based Search

We next discuss binary search that uses ORAM. Unless noted otherwise, we let the array size be $m = 2^k - 1$ for some integer k, i.e., the set can be represented as a perfect binary tree. Also, we use notation $\log(\cdot)$ to denote $\lceil \log_2(\cdot) \rceil$.

The algorithm for performing an ORAM-based binary search follows the same structure as that of a conventional binary search with the difference that the array access to a known index is replaced with ORAM access to a secret index. Due to the access pattern hiding properties of ORAM, no information about the decision (i.e., go left or right) will be leaked after each round of comparisons. Given an ORAM set up over sorted dataset $[a_0], \ldots, [a_{m-1}]$, we denote an ORAM read to a logical address $[d]$ as $\mathsf{ORAMR}([d])$. Note that for an entry $[a_i]$, the index i represents its logical address rather than its physical address in the ORAM. Furthermore, typical realizations of ORAM hide the type of operation (read or write) on each access, but in our context the operation type is public knowledge. Thus, we reveal the type of the operation in the ORAM notation (in the case that knowledge of the operation can permit performance optimizations).

The ORAM-based binary search, OramBS, is given as Protocol 9. It starts by comparing the search key to the element at position $\lfloor m/2 \rfloor$. Because the location of the first access is fixed and known, we store $[a_{\lfloor m/2 \rfloor}]$ outside of ORAM to save a costly ORAM access. The value of d stores the (private) location which should be read in each round, and after the total of j comparisons moves $\lfloor m/2^j \rfloor$ positions left or right depending on the last comparison result c.

In the last iteration, the search key b is compared to the element at position d to determine whether to return a_d or a_{d+1}. While the element at position d was just retrieved, retrieving a_{d+1} requires another ORAM access. We, however, observe that this extra ORAM read is not needed because a_{d+1} is guaranteed to be retrieved in an earlier ORAM access. That is, because we know $b > a_d$ based on the last comparison and $b \leq a_{d+1}$ based on correctness of the search, the element at position $d+1$ has to reside on the path from the root of the binary search tree $a_{\lfloor m/2 \rfloor}$ to a_d (except when $d = m-1$ is the last position). This means that a_{d+1} has been previously read and all we need is to maintain a copy of it instead of invoking another ORAM access. This is what `OramBS` does: variable p stores the last element on the path from the root when the search proceeded left, i.e., the searched value was smaller than the element on the path (p is conditionally updated on line 8 during each loop iteration). This guarantees that p will be equal to a_{d+1} if the comparison in the last round results in incrementing d. In the event that the path never goes left, p is initialized to a_{m-1} on line 3.

The computation on lines 3 and 8 uses conditional statements of the type `if` ($[c]$) `then` $[p] = [x]$ `else` $[p] = [y]$ expressed as $[p] = [c] \wedge [x] \vee [\neg c] \wedge [y] = [c] \cdot [x] + (1 - [c]) \cdot [y]$. We rewrite them as $[p] = [c]([x] - [y]) + [y]$ to save one multiplication each. Furthermore, because updating the value of p can be done together with the next interactive operation, updating p contributes to the round complexity only in the last round. Thus, the cost of the ORAM-based binary search is heavily dominated by $\log m$ ORAM accesses and $\log m$ LE comparisons. Security is discussed in Section 5.4.

Gentry et al. [27] suggested a formula for performing greater-than comparisons, $x > y$, on bit-decomposed k-bit values written as $x = x_{k-1} \dots x_0$ and $y = y_{k-1} \dots y_0$. The computation is

$$g(x_{k-1} \dots x_0, y_{k-1} \dots y_0) = (x_{k-1} - y_{k-1})x_{k-1} + (x_{k-1} - y_{k-1} + 1)g(x_{k-2} \dots x_0, y_{k-2} \dots y_0). \tag{5.1}$$

Note that a straightforward implementation of this function would result in k rounds of computation, which becomes prohibitive in an application like binary search where comparisons are executed sequentially. We notice that the formula can be rewritten in a different form to support constant-round evaluation as $g(x, y) = \sum_{i=0}^{k-1} x_i z_i \prod_{j=i+1}^{k-1} w_j$, where $z_i = x_i - y_i$ and $w_i = z_i + 1$. However, note that the original formula was written for computation in $\mathbb{Z}_2$ and does not produce correct output when the computation is over a larger field. Furthermore, if we rewrite the formula to be correct, it can no longer be represented in a compact form suitable for constant-round evaluation. This means that we do not further consider it as competitive for our application compared to other options (such as LE from [21] mentioned above, which also avoids bit decomposition).

5.1.3 ORAM-based Optimizations

Gentry et al. [27] suggested an optimization to tree-based ORAM constructions that allows ORAM-based binary search to have asymptotically the same cost as that of a recursive tree-based ORAM access. Briefly, the optimization eliminates the need for a recursive position map to translate a logical address to its physical address in ORAM by adding pointers to each block that store the locations of the next access in the ORAM. As a result, the total cost of a binary search is equal to that of a single recursive ORAM access instead of a logarithmic number of full ORAM accesses in general tree-based ORAM protocols. The optimized solution was not empirically evaluated in [27] and therefore it is difficult to tell how its performance might compare to other constructions in the literature. We, however, note that this optimization is expected to apply to other tree-based ORAM constructions which recursively outsource the position map. In particular, we determined that one of the latest efficient ORAMs, 3PC ORAM [33], is tree-based and uses recursive position maps for each ORAM access. Thus, the optimization is applicable to 3PC ORAM and we use the resulting optimized 3PC ORAM in the performance evaluation of our constructions.

We also note that it is possible to optimize performance of ORAM-based binary search regardless of the internal structure of the underlying ORAM construction. The general idea behind our optimization is that we partition the original dataset $a_0, \ldots, a_{m-1}$ into $\log m$ layers of exponentially increasing size. The ith layer will correspond to the elements which can be accessed during the ith step of the binary search computation. That is, for m of the form $2^k - 1$, layer 0 contains only a single element $a_{\lfloor m/2 \rfloor}$, layer 1 contains two elements $a_{\lfloor m/4 \rfloor}$ and $a_{\lfloor 3m/4 \rfloor}$, etc., and layer $\log(m) - 1$ contains $(m+1)/2$ elements at even positions i. With this division, we can set up a separate ORAM for each layer, significantly decreasing the work associated with the first accesses to ORAM and thus leading to practical improvements in the performance of binary search.

This optimization can lead to varying impacts on the asymptotic complexity of the resulting binary search for different ORAM constructions. In particular, for constructions with polylogarithmic (in dataset size) complexities of ORAM access, there might be no asymptotic gain in applying this optimization. For example, if ORAM access costs $O(\log^2 N)$ for an ORAM set up for N elements, then making $\log m$ accesses of cost $O(\log^2 m)$ and making $\log m$ accesses with exponentially increasing sizes from $O(1)$ to $O(m)$ will both result in $O(\log^3 m)$ overall cost. On the other hand, ORAM constructions of larger asymptotic complexities can see pronounced improvements in the asymptotic cost. For example, we can look at Floram [29], which is one of the fastest two-party ORAM constructions. Its asymptotic complexity per access is $O(\sqrt{N})$ communication and $O(N)$ local work. After applying our optimization, the total work associated with binary search decreases from $O(m \log m)$ using full-size ORAM for each access to only $O(m)$. We empirically evaluate the associated performance gain and report it in section 5.5.

5.2 Hierarchical Construction

In this section we describe two new approaches to binary search, both of which are hierarchical and use only a logarithmic number of comparisons. The high-level structure is similar to the one used with ORAM-based search, but we replace the mechanism for protecting true accesses at each iteration. Because of the cost of an ORAM access, it can be beneficial to replace it with alternative computation, including solutions of higher asymptotic complexity linear in the size of a layer. Our findings are in line with those in [18] that demonstrated that linear-time constructions for accessing an element at a private location outperform ORAM performance in practice unless the size of the array becomes very large.

In this section, we offer two solutions: The first one is based on array rotation to hide accessed locations and the second generates tags for all elements in a layer to mark the true path. As with the previous, ORAM-based, construction, we divide the array into layers and process one layer at a time. The conceptual difference is how true accesses are protected for each layer.

5.2.1 Rotation-based Construction

Our first construction rotates all elements in a layer to hide the true access pattern and is based on the following high-level idea: after comparing an element to the search key, we privately determine whether we are jumping left or right and compute the location d to read in the next layer. We next rotate the next layer of size 2^i by a random private amount $r \in [0, 2^i - 1]$ and disclose the protected location $(d + r) \bmod 2^i$. Once we know the location, we can retrieve the desired element and perform the next comparison without knowing what the true index d was. Note that the rotation operation should be oblivious and performed once per search for each layer to ensure that no information about the accessed locations is revealed.

Our algorithm for rotation-based binary search, RotBS, is given as Protocol 10. As before, we are given an ordered set $[a_0], \ldots, [a_{m-1}]$, where m is of the form $2^k - 1$, and divide it into layers as in the optimized ORAM-based solution, i.e., with layer 0 consisting of only one element $a_{\lfloor m/2 \rfloor}$ and layer $\log(m) - 1$ containing all elements at even indices.

Let $[\ell^{(i)}]$ denote the elements stored at layer i and $[\ell_j^{(i)}]$ denote the jth element stored at layer i. We first randomly generate a secret random offset $r^{(i)} \in [0, 2^i)$ for each layer $i > 0$ and rotate all layers by those offsets in parallel to mini-

Protocol 10 $[z] \leftarrow \mathsf{RotBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
1: let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor}], [a_{\lfloor 3m/2^{i+1} \rfloor}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor}] \rangle$ for $i = 1, \ldots, \log(m) - 1$;
2: for $i = 1, \ldots, \log(m) - 1$ in parallel do
3:     $\langle [\hat{\ell}^{(i)}], [r^{(i)}] \rangle \leftarrow \mathsf{Rotate}([\ell^{(i)}])$;
4: end for
5: $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor}.key])$;
6: $[p] \leftarrow [c] \cdot ([a_{\lfloor m/2 \rfloor}] - [a_{m-1}]) + [a_{m-1}]$;
7: $[d] = 0$;
8: for $i = 1, \ldots, \log(m) - 1$ do
9:     $[d] = 2[d] + [c]$;
10:    $[w] \leftarrow \mathsf{RandInt}(\kappa + 1)$;
11:    $s' = \mathsf{Open}([d] + [r^{(i)}] + 2^i [w])$;
12:    $s = s' \bmod 2^i$;
13:    $[a] = [\hat{\ell}_s^{(i)}]$;
14:    $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
15:    $[p] \leftarrow [c] \cdot ([a] + [p]) + [p]$;
16: end for
17: return $[z] = [p]$;
```

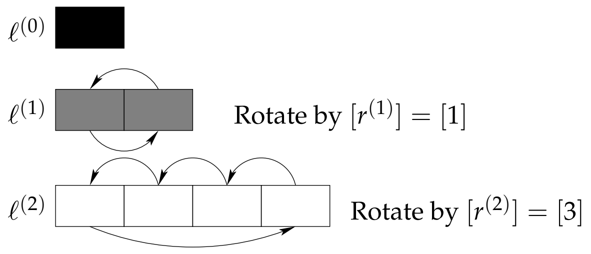

Figure 5.1: Illustration of the rotation-based construction.

mize round complexity (lines 2–4). This is illustrated in Figure 5.1. The rotation algorithm Rotate, which also chooses the amount of rotation, is described afterwards.

After rotating all layers, we proceed with comparisons and first retrieve the (only) element from the top layer $[\ell^{(0)}]$ and compare it with the target value b (line 5). We continue by jumping left or right as before and privately computing the index d to access next using local computation (line 9). Note that unlike OramBS that computed this index d as an index in the entire array, this time we compute it as a position in a layer. This means that we start from two possible positions in layer 1 and update d in the next layer as $2[d] + [c]$, where $[c]$ is the result of the current comparison.

To safely disclose the desired (protected) position in the rotated array, we must open $(d + r^{(i)}) \bmod 2^i$ instead of simply $d + r^{(i)}$. While the latter is a location operation, computing the remainder modulo a power of 2 in this framework is a rather expensive operation, with the same number of rounds as in comparisons and a number of interactive operations linear in the size of the modulus, i.e., in i (see Mod2m in [21]). Fortunately, we were able to get around this cost and safely disclose $(d + r^{(i)}) \bmod 2^i$ using only local computation (prior to the opening). In particular, we mask the $(i+1)$st bit of the sum $d + r^{(i)}$, i.e., the carry bit, by a randomly chosen integer, which allows us to safely open the result. To achieve this, we rely on statistical secrecy and choose an integer of κ bits longer than the value we are protecting, where κ is a statistical security parameter (line 10). The corresponding protocol is called RandInt and takes an argument that specifies the bitlength of an integer to generate. It can be realized non-interactively as described in [21]. The reader may notice that this approach requires that the field can represent integers κ bits longer than the bitlength of the key values. This, however, is already required by the LE protocol.

Because each layer has been rotated by a one-time random offset, we can safely proceed by revealing the value of $(d + r^{(i)}) \bmod 2^i$ in each layer, retrieving that element in the rotated layer, and performing the comparison until we reach the last layer. In addition to maintaining the current element used in the comparison, we also keep track of the last retrieved element p which was $\leq$ the target b, in the same way as in OramBS. It will be retrieved in the last round if the returned element should not be the one used in the last comparison. As

Protocol 11 $(\langle[\hat{a}_0],\ldots,[\hat{a}_{2^u-1}]\rangle,[r]) \leftarrow \mathsf{Rotate}([a_0],\ldots,[a_{2^u-1}])$

```
1: for $i = 0,\ldots,u-1$ in parallel do
2:     $[r_i] \leftarrow \mathsf{RandBit}()$;
3: end for
4: let $[a_j^{(0)}] = [a_j]$ for $j \in [0, 2^u - 1]$;
5: for $i = 0,\ldots,u-1$ do
6:     for $j = 0,\ldots,2^u-1$ in parallel do
7:         $[a_j^{(i+1)}] \leftarrow [a_j^{(i)}] - ([a_j^{(i)}] - [a^{(i)}_{(j-2^i) \bmod 2^u}]) \cdot [r_i]$;
8:     end for
9: end for
10: $[r] = \sum_{i=0}^{u-1} 2^i [r_i]$;
11: let $[\hat{a}_j] = [a_j^{(u)}]$ for $j \in [0, 2^u - 1]$;
12: return $(\langle[\hat{a}_0],\ldots,[\hat{a}_{2^u-1}]\rangle,[r])$;
```

before, the default value of p is a_{m-1}.

The cost of this protocol consists of $\log(m)$ comparisons and other cheaper operations (i.e., multiplications and openings) and is dominated by the cost of rotating all layers. Because some interactive operations could be combined and executed in the same round, the overall round complexity is that of rotation (see below) plus $4 \log m$. Security is shown in Section 5.4.

We next describe our Rotate protocol. It takes as input an array $[a_0],\ldots,[a_{2^u-1}]$ of size 2^u, generates a random u-bit integer r, circularly rotates the elements of the array by r positions, and outputs r together with the rotated array. Our solution is conceptually simple and is given as Protocol 11. As the first step, we generate u random bits using protocol RandBit (line 2) and use them to assemble u-bit offset r (line 10). RandBit can be implemented using 1 interactive operation [21]. Despite having a higher cost than $\mathsf{RandInt}(u)$, generating r from random bits is important for two reasons: (i) we use the bits in the computation that follows and (ii) it allows us to generate a value in the exact range $[0, 2^u - 1]$, while RandInt generates values slightly larger than of the specified bitlength.

The next step is to rotate the array elements by the generated offset, which we do in u iterations: in iteration i, the elements are shifted by $2^i \cdot r_i$ positions right, i.e., we conditionally perform the shift if the ith bit of r is set. After u iterations, the array elements are shifted by the value of r, as desired. In more detail, on line 7 we either keep the current element a_j at position j or replace it with the element at position $a_{j-2^i \bmod 2^u}$ based on the value of r_i.

When $u = \log(m)$, Rotate requires $m \log m$ interactive operations in $\log(m) + 1$ rounds. Recall that we use it in RotBS and execute rotations for all layers in parallel in the beginning. However, rotation of only the smallest layer 1 needs to finish prior to its use in the first loop iteration on line 13. This means layer rotations do not increase the round complexity of RotBS. As before, we summarize performance of our binary search constructions in Table 5.1.

On using ring $\mathbb{Z}_{2^k}$. Before we conclude, we comment on executing our protocols over ring $\mathbb{Z}_{2^k}$ instead of a finite field. All protocols described so far except RotBS work unmodified when instantiated with building blocks over ring $\mathbb{Z}_{2^k}$. The difference is that RotBS protects a value, opens it, and uses i least significant bits in further computation (i.e., lines 10–12 of RotBS). We note that this operation can become even easier over ring $\mathbb{Z}_{2^k}$ because all values are automatically reduced modulo a power of 2. In particular, instead of prepending a large random value to statistically hide the overflow from i bits, we can directly open the sum $[r^{(i)}] + [d]$ and use the result as s, as long as all shares are reduced modulo 2^i prior to the opening.

$$\begin{tikzcd}[column sep=tiny]
 & & & q^{(0)} = 1 \arrow[dll] \arrow[drr] & & & \\
 & q_0^{(1)} = q^{(0)} \wedge c^{(0)} \arrow[dl] \arrow[dr] & & & & q_1^{(1)} = q^{(0)} \wedge \neg c^{(0)} \arrow[dl] \arrow[dr] & \\
q_0^{(2)} = q_0^{(1)} \wedge c^{(1)} & & q_1^{(2)} = q_0^{(1)} \wedge \neg c^{(1)} & & q_2^{(2)} = q_1^{(1)} \wedge c^{(1)} & & q_3^{(2)} = q_1^{(1)} \wedge \neg c^{(1)}
\end{tikzcd}$$

Figure 5.2: Illustration of the tag-based construction.

5.2.2 Tag-based Construction

Our second hierarchical solution utilizes a different mechanism for retrieving an element at a private location from each layer. While a generic protocol for reading an element at a private location could be used (e.g., multiplexor-based or as described in [18]), in the context of binary search we notice that the location read at layer $i+1$ is highly correlated to the location previously read at layer i. This observation allows us to generate binary tags for each layer using tags of the layer before, where in each layer the tag of a single position is set to 1 and the tags at all other positions are set to 0. This representation consequently permits us to retrieve the element at the marked position using efficient dot product computation.

Our tag-based construction, TagBS, is given as Protocol 12. As before, let $[\ell^{(i)}]$ denote the elements at layer i and let us use similar notation $[q^{(i)}]$ to denote binary tags for layer i.

Initially, layer 0 has a single element and its tag is set to 1. The next layer has two elements, one of which will be set to 1 based on the result of the first comparison and the other will be set to 0. We continue computing tags for the current layer from the tags of the previous layer as follows: if the "parent" tag at position j is 0, both "children" tags at positions $2j$ and $2j+1$ will be 0. If the "parent" tag is 1, one of the "children" tags will be 0 and one will be 1 based on the result of the current comparison c. This process is illustrated in Figure 5.2.

Protocol 12 $[z] \leftarrow \mathsf{TagBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
1: let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor - 1}], [a_{\lfloor 3m/2^{i+1} \rfloor - 1}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor - 1}] \rangle$ for $i = 1, \ldots, \log(m) - 1$;
2: $[q^{(0)}] = \langle 1 \rangle$;
3: $[c] \leftarrow \mathsf{LE}([b], [a_{\lfloor m/2 \rfloor - 1}.key])$;
4: $[p] = [c] \cdot ([a_{\lfloor m/2 \rfloor - 1}] - [a_{m-1}]) + [a_{m-1}]$;
5: for $i = 1, \ldots, \log(m) - 1$ do
6:     for $j = 0, \ldots, 2^i - 1$ in parallel do
7:         if $j \bmod 2 = 0$ then
8:             $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot [c]$;
9:         else
10:            $[q_j^{(i)}] = [q_{\lfloor j/2 \rfloor}^{(i-1)}] \cdot (1 - [c])$;
11:        end if
12:    end for
13:    $[a] = \sum_{j=0}^{2^i-1} [q_j^{(i)}] \cdot [\ell_j^{(i)}]$;
14:    $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
15:    $[p] = [c] \cdot ([a] - [p]) + [p]$;
16: end for
17: return $[z] = [p]$;
```

Then to retrieve the marked element at the current layer, we compute the dot product of the elements and their tags in the current layer (line 13). Also, as before, we maintain another element in variable p, which will be used at the end if the result of the last comparison is false.

The overhead of this protocol is given in Table 5.1. Note that TagBS uses fewer interactive operations compared to rotation-based RotBS, but the latter has fewer rounds. Thus, we expect that TagBS will be a faster choice in typical circumstances, but RotBS can be beneficial for high-latency connections. In addition, comparison-based CompBS has constant round complexity, but is expected to be slower for larger datasets due to its communication volume.

Protocol	Rounds	Interactive operations
CompBS	5	$(4k-1)m-1$
OramBS	$(\log m)(3+\mathsf{ORAMRead})$	N/A
RotBS	$4\log m$	$4k\log(m)+$ $m\log(m)-1$
TagBS	$5\log m$	$4k\log(m)+m-2$
LayHBS, $\tilde{\imath}<\log(m)-1$	$5(\log(m)-\tilde{\imath})+4$	$(4k-3)2^{\tilde{\imath}}+m+2$
SubHBS, $\alpha\cdot\beta=m$	$\mathsf{MBS}(\alpha)+\mathsf{BS}(\beta)+1$	$\mathsf{MBS}(\alpha)+\mathsf{BS}(\beta)+\beta$

Table 5.1: Performance of binary search using field-based building blocks in the semi-honest model with honest majority.

5.3 Hybrid Construction

In this section we discuss how combining multiple constructions in a single solution can be used to further improve performance of binary search. Section 5.3.1 discusses a solution in which portions of the binary search tree are processed using different algorithms, and section 5.3.2 presents a solution where previously developed constructions are applied only to a subset of the tree nodes resulting in sublinear communication cost. In particular, while all of our constructions including the one in section 5.3.1 require $O(m)$ communication, the solution of section 5.3.2 lowers communication to $O(\sqrt{m})$ by relying on an efficient dot product protocol.

5.3.1 Composition of Layers

The binary search constructions described so far have their own advantages. For example, comparison-based CompBS has constant round complexity, which is the lowest across all protocols. Its communication cost, however, is rather high and is linear in $m\cdot k$. These properties make it a good choice for datasets of small size, but performance is expected to deteriorate as the dataset size grows. In contrast, tag-based TagBS has the lowest communication complexity as the dataset size m increases, but the largest round complexity. Because it invokes only a logarithmic number of comparisons, it is expected to outperform other constructions for larger values of m, but perform relatively worse for very small values of m. In particular, the first three rounds of comparisons in that construction process only 7 elements, but use 15 rounds. This can be contrasted with the total of 5 rounds in CompBS. Coupled with the fact that communication latency is the major overhead when the dataset is small, TagBS is sub-optimal during processing of the top layers.

The hybrid construction we propose here combines flat and hierarchical structures to take advantage of the benefits of both of them. More precisely, we replace the top layers of the hierarchy with a flat structure and design a transition to feed the results of evaluating the flat structure to the next layer in the hierarchy. Although the idea is straightforward, its realization requires careful design because the constructions have different interfaces and rely on different intermediate results. We illustrate a transition mechanism on the example of combining CompBS and TagBS.

Recall that CompBS computes an array of bits $[c_0], \ldots, [c_{m-1}]$, where the bits in the beginning of the array are 0s and switch to 1s at the location of the searched element $[b]$. The array is consequently used to compute another bit array $[d_0], \ldots, [d_{m-1}]$, in which all elements are 0 except for the location of the switch, and the non-zero bit is used to retrieve the desired element of the array $[a_0], \ldots, [a_{m-1}]$. TagBS, on the other hand, proceeds in a hierarchical manner and prior to moving to layer i expects state $[c]$, $[p]$ that indicates the position of the desired element in the already processed layers and computed tags $[q_j^{(i-1)}]$ for layer $i-1$. Our transition computes the value of $[p]$ and tags $[q_j^{(i)}]$ from the bit arrays computed by CompBS and no explicit $[c]$, as maintained by TagBS, is

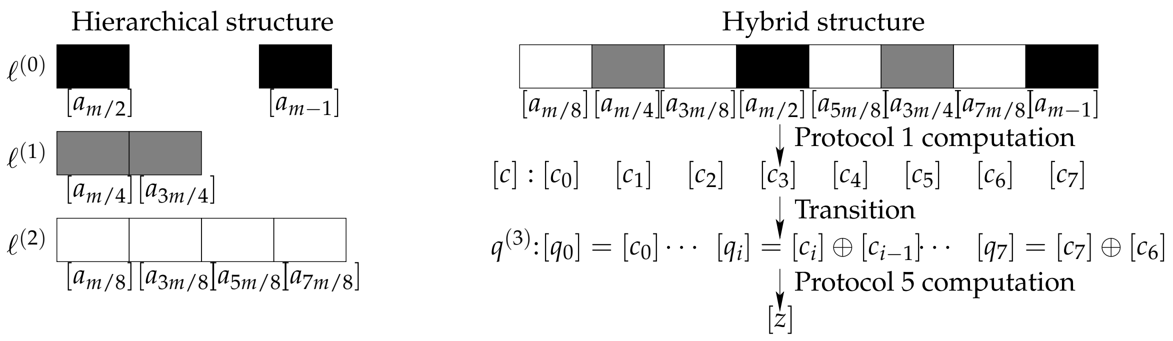
Hierarchical structure
$\ell^{(0)}$
$[a_{m/2}]$
$[a_{m-1}]$
$\ell^{(1)}$
$[a_{m/4}]$ $[a_{3m/4}]$
$\ell^{(2)}$
$[a_{m/8}]$ $[a_{3m/8}]$ $[a_{5m/8}]$ $[a_{7m/8}]$
Hybrid structure
$[a_{m/8}]$ $[a_{m/4}]$ $[a_{3m/8}]$ $[a_{m/2}]$ $[a_{5m/8}]$ $[a_{3m/4}]$ $[a_{7m/8}]$ $[a_{m-1}]$
Protocol 1 computation
$[c]:[c_0]$ $[c_1]$ $[c_2]$ $[c_3]$ $[c_4]$ $[c_5]$ $[c_6]$ $[c_7]$
Transition
$q^{(3)}:[q_0]=[c_0]\cdots$ $[q_i]=[c_i]\oplus[c_{i-1}]\cdots$ $[q_7]=[c_7]\oplus[c_6]$
Protocol 5 computation
$[z]$

Figure 5.3: Illustration of the layer-based hybrid construction.

used in the process.

Suppose we would like to process the first $\tilde{\imath}$ layers, or equivalently $\tilde{m} = 2^{\tilde{\imath}}$ elements, using the flat structure. We execute the main portion of CompBS on elements associated with layers 0 through $\tilde{\imath}-1$, use transition to generate $[p]$ and the tags at layer $\tilde{\imath}$, and continue with the remaining computation as in TagBS. Figure 5.3 illustrates the process for $\tilde{\imath} = 3$. Note that the very last element of the original dataset, a_{m-1}, is appended to the elements that CompBS processes and is treated as an element of layer 0.

We determined that the relationship between $c_0, \ldots, c_{\tilde{m}-1}$ computed by CompBS and $q_0^{(\tilde{\imath})}, \ldots, q_{\tilde{m}-1}^{(\tilde{\imath})}$ needed in TagBS is rather simple. In particular, we have:

$$q_j^{(\tilde{\imath})} = \begin{cases} c_0 & \text{if } j = 0 \\ c_{j-1} \oplus c_j & \text{if } 0 < j \le \tilde{m}-1 \end{cases} \tag{5.2}$$

Somewhat surprisingly, each $q_j^{(\tilde{\imath})}$ is computed in the same way, while we expect differences in the computation of even and odd elements because only half of the values are associated with layer $\tilde{\imath}-1$. XOR of each $[c_{j-1}]$ and $[c_j]$ is easily computed as $[c_{j-1}] + [c_j] - 2[c_j] \cdot [c_{j-1}]$. This is equivalent to $2[d_j] - [c_j] + [c_{j-1}]$, where the $[d_j]$s are as computed by CompBS. Because availability of $[d_j]$s makes the computation of $[q_j^{(\tilde{\imath})}]$s local, a notable implication for us is that combining the two constructions reduces the overall cost below that of running CombBS and TagBS on the respective portions of the data.

It is also necessary to compute the appropriate value of $[p]$ during the transition. This computation comes first (i.e., to finish processing layer $\tilde{\imath} - 1$), and $[p]$ is computed in the same way as $[z]$ in CompBS, i.e., the first element which was determined to be $\leq b$ so far will be used as the next larger element if the remaining search returns that b is greater than all other elements.

In practice, the best choice of $\tilde{\imath}$ depends on the setup. When the network latency between the computational parties is small, the round complexity may have less impact on performance and therefore a lower $\tilde{\imath}$ is preferred. In contrast, if the network latency is high, a larger $\tilde{\imath}$ reduces the round complexity and therefore could be a better choice. We provide additional comments in section 5.5. The exact cost is listed in Table 5.1 and this time is a function of $\tilde{\imath}$.

5.3.2 Composition of Subtrees

All solutions presented so far, except OramBS, have communication linear in the size of the dataset m. In this section we show how performance of previously presented constructions can be further improved to $O(\sqrt{m})$ communication using only standard building blocks.

The high-level idea behind this solution is as follows. Recall that, on input m elements, CompBS works by generating a bit array $[d_0], \ldots, [d_{m-1}]$ with a single element set to 1 and that array is used to retrieve the searched element using a dot product (which costs 1 interactive operation). Now suppose that instead of retrieving a single element at the end of the computation, we use the computed

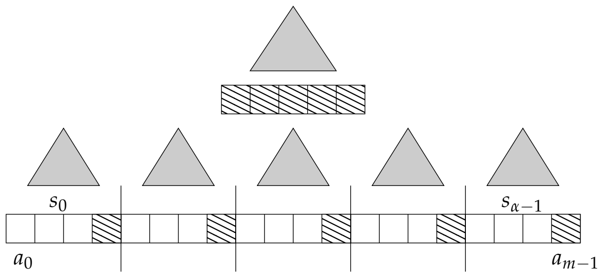

Figure 5.4: Illustration of the subtree-based hybrid construction. Shaded array elements are used to form the top tree of size α. Subtrees s_0 through $s_{\alpha-1}$ have size β.

bits to retrieve a desired subset of the elements, or a subtree if the elements are organized in a hierarchy. The high-level structure of this approach is illustrated in Figure 5.4.

We organize the dataset in a hierarchy and run a modified binary search (which produces a bit array) on the top portion of the tree of size α. We consequently use dot products to obliviously localize the search to the relevant subtree of size $\beta = m/\alpha$, call binary search on that tree, and use its output as the result of the search. In other words, the top-level search allows us to determine in what portion of the dataset the searched element falls and the second low-level search determines the exact position and returns the desired element.

The solution, SubHBS, is formalized as Protocol 13. We use a modified version of a binary search protocol that produces a unity bit array, denoted as MBS. The second call to binary search, on the other hand, uses the conventional interface. We note that communication savings are possible because of the use of dot product protocols, which on input of two vectors of arbitrary size require only one interactive operation. This means that when we set $\alpha = \beta = O(\sqrt{m})$, communication complexity of this solution reduces from $O(m)$ to $O(\sqrt{m})$. We note that local computation remains $O(m)$, but this component of interactive protocols is much easier to speed up than communication, e.g., by employing

Protocol 13 $[z] \leftarrow \mathsf{SubHBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b], \alpha)$

```
1: let $\beta = m/\alpha$;
2: let $[s^{(i)}] = \langle [a_{i \cdot \beta}], [a_{i \cdot \beta + 1}], \ldots, [a_{i \cdot \beta + \beta - 1}] \rangle$ for $i = 0, \ldots, \alpha - 1$;
3: $\langle [d_0], \ldots, [d_{\alpha-1}] \rangle \leftarrow \mathsf{MBS}(\langle [s^{(0)}_{\beta-1}], [s^{(1)}_{\beta-1}], \ldots, [s^{(\alpha-1)}_{\beta-1}] \rangle, [b])$;
4: for $i = 0, \ldots, \beta - 1$ in parallel do
5:     $[u_i] \leftarrow \sum_{j=0}^{\alpha-1} [s^{(j)}_i] \cdot [d_j]$;
6: end for
7: $[z] \leftarrow \mathsf{BS}(\langle [u_0], \ldots, [u_{\beta-1}] \rangle, [b])$;
8: return $[z]$;
```

more powerful hardware and/or using multi-threading.

It should be clear that the call to binary search BS on line 7 of SubHBS can be instantiated with any binary search construction described so far. This includes hybrid LayHBS and even SubHBS itself, which in that case would be called recursively. Similarly, the first call to binary search on line 3 can be instantiated with any binary search protocol modified to produce a bit array instead of a single element. So far we only showed that CompBS can be naturally used for this purpose, but below we also show how to modify TagBS. This would imply that LayHBS could additionally be used for that purpose. Lastly, if we replace the call to BS on line 7 of SubHBS with a call to MBS, SubHBS itself can be invoked on line 3. Combined with the ability to choose parameter α (and consequently β), these options provide a rather significant performance optimization space in practice.

What remains is to discuss our solution for modifying TagBS to implement the interface for MBS. Recall that TagBS generates tags $[q^{(i)}]$ for each layer i in the hierarchy and maintains variable $[p]$. The tags have a useful structure, which each tag being a bit and having only a single tag set to 1 per layer. However, to generate a desired bit array $[d_0], \ldots, [d_{m-1}]$, we need to have a single bit set to 1 out of the entire dataset and not per layer. Instead of trying to combine different

Protocol 14 $\langle [d_0], \ldots, [d_{m-1}] \rangle \leftarrow \mathsf{TagMBS}(\langle [a_0], \ldots, [a_{m-1}] \rangle, [b])$

```
1: let $[\ell^{(i)}] = \langle [a_{\lfloor m/2^{i+1} \rfloor}], [a_{\lfloor 3m/2^{i+1} \rfloor}], \ldots, [a_{\lfloor (2^{i+1}-1)m/2^{i+1} \rfloor}] \rangle$ for $i = 0, \ldots, \log(m) - 1$;
2: $[q_0^{(0)}] = 1$;
3: for $i = 0, \ldots, \log(m) - 1$ do
4:     $[q^{(i)}] = \langle [q_0^{(i)}], \ldots, [q_{2^i-1}^{(i)}] \rangle$;
5:     $[a] = \sum_{j=0}^{2^i-1} [q_j^{(i)}] \cdot [\ell_j^{(i)}]$;
6:     $[c] \leftarrow \mathsf{LE}([b], [a.key])$;
7:     for $j = 0, \ldots, 2^i - 1$ in parallel do
8:         if $j \bmod 2 = 0$ then
9:             $[q_j^{(i+1)}] = [q_{\lfloor j/2 \rfloor}^{(i)}] \cdot [c]$;
10:        else
11:            $[q_j^{(i+1)}] = [q_{\lfloor j/2 \rfloor}^{(i)}] \cdot (1 - [c])$;
12:        end if
13:    end for
14: end for
15: return $\langle [d_0], \ldots, [d_{m-1}] \rangle = \langle [q_0^{(\log m)}], \ldots, [q_{m-1}^{(\log m)}] \rangle$;
```

layers into a single array, out solution is to continue with the structure of TagBS and generate one more layer $[q^{(\log m)}]$, which this time will have a tag for each element of the original dataset and only a single element will be set to 1. We also remove the code for creating and maintaining the value of $[p]$ because it is no longer used in the protocol. The final solution TagMBS is formalized as Protocol 14. As noted before, this variant will allow for a variety of solutions to be used with SubHBS.

Performance of hierarchical SubHBS is shown in Table 5.1. Because the algorithm makes calls to binary search algorithms on datasets of smaller sizes, we use notation $\mathsf{BS}(x)$ and $\mathsf{MBS}(x)$ to denote the cost of binary search and modified binary search, respectively, invoked on input of size x.

5.4 Security Proof

In this section, we use the standard definition of security in the presence of semi-honest participants. The definition requires that an adversary corrupting at most t participants is unable to learn any information about private inputs. This is modeled by showing that a simulator without access to the private inputs is able to simulate the adversarial view, which the adversary is not able to distinguish from the actual view of a protocol execution.

Note that showing a protocol secure under this definition will also guarantee that the computation is data-oblivious, i.e., does not depend on any private data and can be simulated without access to private data.

As mentioned earlier, for concreteness, we instantiate our constructions using classical Shamir secret sharing and corresponding building blocks for operations such as multiplication, comparisons, etc. This setting requires honest majority, i.e., $t < n/2$. We next proceed with showing our protocols secure according to definition 1.

Theorem 3. *Let λ be a computational security parameter and κ be a statistical security parameter. Assuming the existence of secure protocols for multiplication, dot product and comparison* LE*, denoted by Π_{mult}, Π_{dot}, Π_{le}, respectively, Protocol 8 is secure according to Definition 1.*

Proof. For this protocol, the parties hold no private inputs and obtain no output. The implication is that it is required that no information about private a_is and b is revealed to corrupt parties I during the protocol execution. This input formulation also means that, upon protocol initiation, the input shares available to the corrupt parties I information-theoretically reveal no information about the underlying input, i.e., they can be reconstructed to any possible values with the same probability.

We build our simulator S_I as follows:

- In step 1, S_I invokes the simulator for Π_{le} m times and simulates the view of the parties in I.
- In step 6, S_I invokes the simulator for Π_{mult} $m-2$ times and simulates the view of the parties in I.
- In step 9, S_I invokes the simulator for Π_{dot} of two m-element vectors and simulates the view of the parties in I.

Next, we need to show that the real and simulated view are indistinguishable. The simulated view consists of the input shares available to the corrupt parties and the messages received during simulation of Π_{le}, simulation of Π_{mult}, and simulation of Π_{dot}. Because the input shares of t or fewer participants are distributed uniformly at random within the field, that component of the view has identical distributions in the real and simulated views and are therefore indistinguishable. Also, because protocols Π_{le}, Π_{mult}, and Π_{dot} are secure (meaning that their simulators are guaranteed to produce indistinguishable views), we get that the remaining components of the view are also indistinguishable, resulting in the overall indistinguishability of the simulation.

The type of indistinguishability that we obtain depends on the indistinguishability guarantees of protocols Π_{le}, Π_{mult}, and Π_{dot}. That is, if at least one of them provides computational guarantees, our simulation will also be computationally secure. Similarly, if at least one of them uses statistical security, our simulation will inherit that requirement as well. Note that many interactive protocols (including Π_{le}, Π_{mult}, Π_{dot}) assume secure pair-wise communication channels, which are normally implemented using computationally secure techniques.

Note that because this protocol produces no private output to any of the corrupt parties, output correctness of the output in the simulated view does not need to be shown. However, if this protocol is followed by another operation that discloses a value to one of more corrupt participants (e.g., open operation), it will always be possible for the simulator to generate the view that will lead to the corrupt parties reconstructing the right output. This is due to the properties of (n, t) secret sharing that will allow any combination of shares that t or fewer participants holds to reconstruct to any possible value in the field. □

Theorem 4. *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication, comparison* LE, *and ORAM read access* ORAMR, *denoted by Π_{mult}, Π_{le}, Π_{oramr}, respectively, Protocol 9 is secure according to Definition 1.*

Proof. As before, we prove security of Protocol 9 in the presence of semi-honest participants according to Definition 1. Recall that the parties hold no private inputs and obtain no output. Our simulator S_I works as follows:

- In step 1, S_I invokes the simulator for Π_{le} and simulates the view of the parties in I.
- In step 3, S_I invokes the simulator for Π_{mult} and simulates the view of the parties in I.
- During evaluation of the loop, S_I iteratively invokes simulators for Π_{oramr} (step 6), Π_{le} (step 7), and Π_{mult} (step 8) $\log m$ times to simulate the view of the parties in I.

The simulated view now consists of the input shares available to the corrupt parties and the messages received during the simulation of Π_{le}, Π_{mult}, and Π_{oramr}. As before, we simulate the input shares by distributing random field elements to the corrupt parties, which have the same distribution as during the real execution and are therefore indistinguishable. In addition, because of security of protocols Π_{le}, Π_{mult}, and Π_{oramr}, their simulators are guaranteed to produce indistinguishable views. Thus, all components of the view are indistinguishable and, as a result, the overall simulation is indistinguishable as well. □

Theorem 5. *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication and random bit generation* RandBit*, denoted by Π_{mult} and $\Pi_{randbit}$, respectively, Protocol 11 is secure according to Definition 1.*

Proof. As before, it is straightforward to prove security of Protocol 11 in the presence of semi-honest participants based on Definition 1. Our simulator S_I needs to invoke the simulator for $\Pi_{randbit}$ u times in step 2 and the simulator for Π_{mult} u times in batches of size 2^u in step 7 to simulates the view of the parties in I.

As before, indistinguishability follows from information-theoretic security of the secret sharing scheme in the presence of at most t corrupt participants and security of protocols Pi_{mult} and $\Pi_{randbit}$ which must come with simulators that produce indistinguishable views. □

So far security of Protocols 8, 9, and 11 was straightforward to show because they only combine secure building blocks. We next discuss Protocol 10, which involves more interesting analysis because it opens values during the computation.

Theorem 6. *Let λ and κ be computational and statistical security parameters, respectively. Assuming the existence of secure protocols for multiplication, random integer generation* RandInt*, comparison* LE*, and rotation* Rotate*, denoted by Π_{mult}, $\Pi_{randint}$, Π_{le}, Π_{rotate}, respectively, Protocol 10 is secure according to Definition 1.*

Proof. As before, we prove security of Protocol 10 in presence of a semi-honest adversaries according to Definition 1, by building a simulator and shoring indistinguishability of the real and simulated views. Recall that the parties hold no private inputs and obtain no output.

Our simulator S_I can be built as follows:

1. In step 3, S_I invokes the simulator for Π_{rotate} on the layers of varying sizes $\log m - 1$ times and simulates the view of parties in I.
2. In step 5, S_I invokes the simulator for Π_{le} and simulates the view of parties in I.
3. In step 6, S_I invokes the simulator for Π_{mult} to simulate the multiplication view.
4. Simulation of the for loop on lines 8–16 proceeds as follows:
   (a) Assuming non-interactive implementation of $\Pi_{randint}$, there is nothing for S_I to simulate for step 10.
   (b) To simulate opening of s' shares, S_I generates a random integer $s' \in \mathbb{Z}_{2^{\kappa+i+1}}$, uses its knowledge of the corrupt parties' shares prior to the opening[2] to compute the remaining shares which would reconstruct to the chosen s', and communicates the computed shares to the parties in I according to the specification of Π_{open}.
   (c) Consequently, S_I invokes the simulators for Π_{le} and Π_{mult} to simulate the view of the parties in I for steps 14 and 15, respectively.

   This portion is repeated and performed $\log(m) - 1$ times to simulate the

[2]Maintaining shares of the corrupted parties for the variables used in the computation of s' would require the simulator to remember and maintain additional information, but this can be accomplished in our setting. In particular, this is because we know the corrupt parties' inputs and messages they receive, because the corrupt parties follow the computation, and because we are working with a redundant secret sharing scheme that permits extraction of corrupt minority's shares from the shares they distribute to the simulated parties during the execution of interactive building blocks.

view of the parties in I.

Next, we demonstrate that the real and simulated view are indistinguishable. Besides the simulations of interactive building blocks such as Π_{mult}, Π_{le}, and Π_{rotate}, which are required to come with simulators that guarantee indistinguishability, this simulation includes an important component which makes it different from prior proofs. In particular, in the simulated view the parties receive a random $(\kappa + i + 1)$-bit integer s', while in the real view it was computed according to the protocol specification. That is, in the protocol s' is computed as $d + r^{(i)} + 2^i w$, where $r^{(i)}$ is a secret one-time i-bit random value and w is a secret one-time $(\kappa + 1)$-bit random value. The purpose of $r^{(i)}$ was to perfectly protect d when disclosing $d + r \bmod 2^i$ and the purpose of w was to statistically protect the carry bit after addition of i-bit d and $r^{(i)}$.

We next argue that the distribution of s' in real execution is statically close to the uniform distribution over $(\kappa + i + 1)$-bit integers (as used in the simulation) and thus we can simulate the view of corrupt parties during protocol execution by drawing a random element from that space and obtain statistical indistinguishability. In more detail, because we protect the $(i + 1)$st carry bit of $d + r^{(i)}$ with a $(\kappa + 1)$-bit random integer, information about the carry bit can only be revealed with at most negligible probability in the security parameter κ. The remaining i bits of $d + r^{(i)}$ and s' are distributed uniformly at random over the range $[0, 2^i - 1]$ and reveal no information information about d, i.e., perfect secrecy is achieved.

Combining this argument with the fact that the simulators of the interactive building blocks produce indistinguishable views, we obtain that the real and simulated views are indistinguishable in the presence of at most t semi-honest adversaries. $\square$

Security of other protocols is not difficult to show using the same logic as that in the proofs of Protocols 8, 9, and 11. Namely, because they only invoke secure building blocks, a composition of secure building blocks will result in security of the overall protocols and indistinguishable simulation can be built by invoking simulators associated with the building blocks.

5.5 Performance Evaluation

We implemented several of our algorithms and carried out experiments on both LAN and WAN. For LAN experiments, we used three machines with 2.1GHz processors connected via 1Gbps Ethernet (934Mbps throughput) with a one-way latency of 0.13ms. Our WAN experiments used local machines and one remote machine with a 2.4GHz processor. The link between the remote and local machines had throughput of 76–85Mbps and a one-way latency of 21ms. While the machine configurations are slightly different, we examine the times to ensure that the differences do not introduce inconsistencies in the experiments. That is, the computation time is determined by the slower machines, and the introduced slowdown on WAN is due to the higher latency and lower bandwidth in the WAN experiments. All experiments used a single thread and the dataset elements had a single field (the key) represented as a 32-bit integer.

5.5.1 Cost in Different Settings

Table 5.1 shows costs of our constructions when instantiated with Shamir secret sharing in the semi-honest setting. If one would like to utilize a different underlying framework, for example, 3-party replicated secret sharing over ring $\mathbb{Z}_{2^k}$,

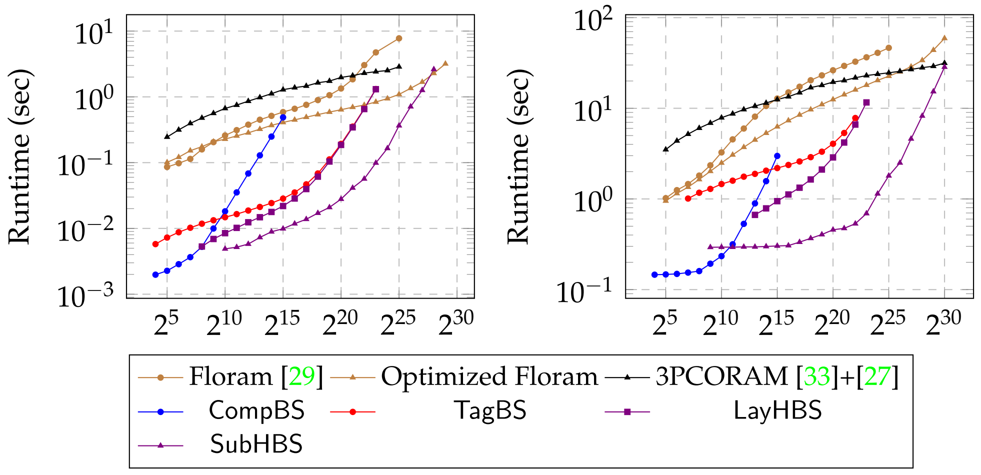

Figure 5.5: Performance of ring-based binary search in the semi-honest model on LAN (left) and WAN (right).

the costs of the building blocks can change which will have an impact on the total cost. Using a different realization of LE would likely most significantly impact CompBS, but we expect that relative performance of different construction will remain similar.

If we realize the constructions in the malicious adversarial setting, many frameworks do not support dot product of communication independent of the input size. Note that all of CompBS, RotBS, TagBS, and the hybrid constructions utilize dot product operations and thus their communication will be impacted. The amount of impact varies based on the relationship of m to the original communication cost. For example, this is a relatively small increase for CompBS, close to doubling for TagBS, and significant increase for SubHBS. The order of the protocols in terms of their communication volumes, however, does not change.

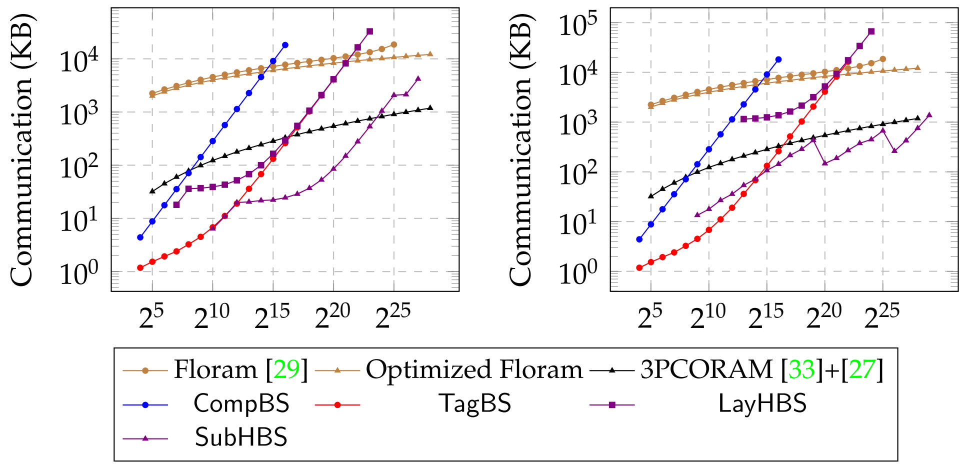

Figure 5.6: Communication costs (per party) of binary search in the semi-honest model on LAN (left) and WAN (right).

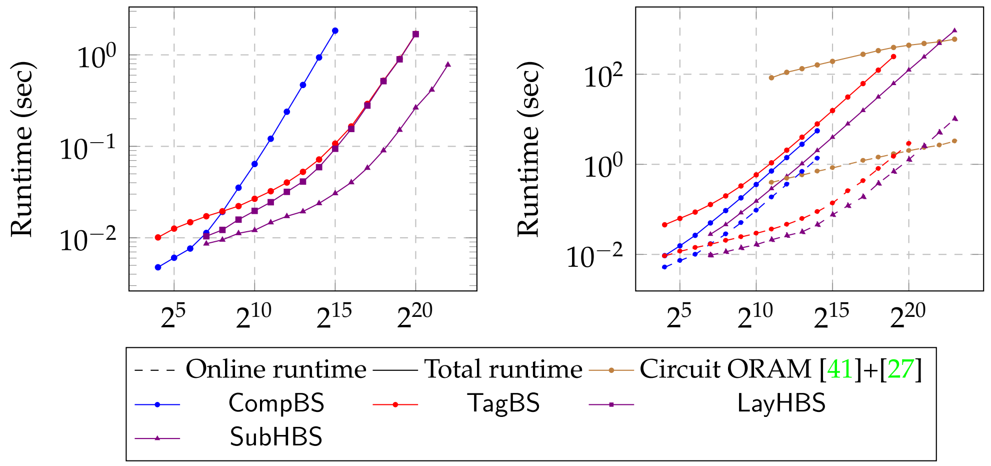

Figure 5.7: Performance of binary search in the malicious model with (left) and without (right) honest majority on LAN.

5.5.2 Performance in the Semi-Honest Model

Our implementation in the semi-honest model is in the honest majority setting (i.e., $t < n/2$). Because ring-based computation is faster than computation over a field, we use replicated secret sharing over $\mathbb{Z}_{2^{32}}$ with three parties. Implementation of comparisons is adapted from field-based LT [21, 81] as described in [82, 19].

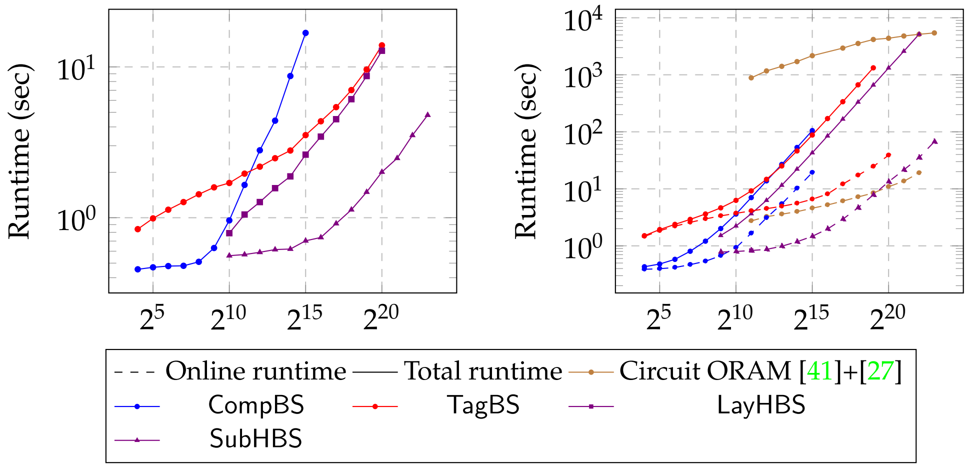

Figure 5.8: Performance of binary search in the malicious model with (left) and without (right) honest majority on WAN.

We compare performance of our constructions to the state-of-the-art Floram [29] and 3PC ORAM [33] after optimizing them. In particular, we run Floram's binary search implementation from [79] and also execute an optimized version using the results of section 5.1.3. We also apply Gentry et al.'s optimization [27] to 3PC ORAM which makes the cost of binary search asymptotically equal to a single ORAM access and run it using the implementation from [83]. Note that 3PC ORAM uses custom techniques not compatible with standard secure computation building blocks. For that reason we approximate performance of optimized 3PC ORAM binary search using comparisons and multiplications as implemented in this work. The exact times might be higher due to the need to convert between different representations. Furthermore, all ORAM schemes come with significant initialization costs not captured in our experiments.

The results of our experiments on both LAN and WAN are given in Figure 5.5. First note that optimized Floram binary search is on par with the original Floram binary search for small datasets and the difference starts to show when m becomes larger than 2^{10}. Also, the optimization applied to 3PC ORAM removes recursion and makes the construction faster.

Our CompBS outperforms all other protocols when the dataset size is small because the round complexity is the bottleneck with small m. With larger m, communication is dominant and CompBS is not competitive, as expected, but maintains its advantage longer on WAN because of the latency. TagBS has lower communication and shows a better runtime than optimized Floram for sizes up to 2^{22} on LAN and larger on WAN, but its linear communication becomes the bottleneck for large sizes.

Hybrid LayHBS can improve performance of TagBS by only a constant. The maximum improvement is achieved when $\tilde{\imath} = 6$ on LAN and $\tilde{\imath} = 12$ on WAN, where the runtime gap between CompBS and TagBS is the largest. Nevertheless, the difference in performance of TagBS and LayHBS diminishes as the size increases.

Our SubHBS shows significant advantage over all other protocols. It outperforms all other options for sizes is up to 2^{27} on LAN and 2^{11}–2^{30} on WAN due to its low communication. As the size increases, its curve becomes steep indicating that local computation is the bottleneck. In particular, the dot-product computation of $O(m)$ local work consumes 94% (45%) of the total time with $m = 2^{25}$ on LAN (resp., WAN). This means that the performance could be significantly improved for large sizes via multi-threading. We also would like to note that while it appears that SubHBS's curve is significantly steeper than that of (optimized) Floram binary search, this will not be the case if we keep increasing m. Both SubHBS and optimized Floram search require $O(m)$ local computation, but computation becomes the bottleneck for SubHBS at smaller sizes because of its superior performance for medium values of m. This is consistent with findings in [29] which show that Floram's curve becomes steep close to 2^{30} in a setting with 500Mbps bandwidth.

Recall that implementing SubHBS involves several optimization choices including the algorithms for the sub-searches and the values for parameter α. While using $\alpha = O(\sqrt{m})$ gives us theoretically the lowest communication, the network environment and the speed of sub-protocols also affect this decision. For each experiment, we vary the value of α and select the best performing protocols for the sub-searches to determine the best configuration. Specifically, for LAN, we used $\alpha = 2^6$ and modified CompBS for the first sub-search for m from 2^{10} to 2^{14} and $\alpha = 2^7$ and TagMBS for larger m; the second sub-search used the fastest protocol for the corresponding β. On WAN, we set $\alpha \approx \sqrt{m}$ for $m < 2^{16}$ and use $\alpha = 2^8$ for m between 2^{16} to 2^{19}. For larger m, we let $\beta = 2^{13}$ or 2^{14} and use SubHBS with $\alpha = 2^8$ recursively for the second sub-search. TagBS and TagMBS were not helpful in WAN experiments because CompBS is faster for small to medium sizes and SubHBS that combines CompBS variants becomes a better choice. We expect the best configurations to vary based on computation and communication resources.

The above choices give us that on LAN each party sends 22.9KB when $m = 2^{15}$, 65KB when $m = 2^{20}$, and 2.12MB when $m = 2^{25}$ per SubHBS. On WAN, communication becomes 107.5KB, 116.5KB, and 554KB for the same sizes. The differences are because we opt for CompBS with fewer rounds on WAN and also use α closer to $\sqrt{m}$ when m is large. As is clear from the above, our protocol configurations were optimized for runtime and do not always use minimally achievable communication. Compared to ORAM solutions, SubHBS uses less communication than Floram binary search, both asymptotically and in practice. For example, for $m = 2^{20}$, Floram search communicates 10MB per party. Communication of optimized 3PCORAM (of polylogarithmic cost) is 543KB for

$m = 2^{20}$, but predictably will get closer to that of SubHBS and outperform it when the size grows to very large. Memory footprint of our constructions is proportional to the computation size (i.e., linear in m for most protocols).

Figure 5.6 illustrates communication costs of our and ORAM-based binary search protocols in the semi-honest setting. The communication costs of the ORAM-based solutions and our CompBS and TagBS are the same on both LAN and WAN, while communication of hybrid LayHBS and SubHBS vary since we adjust the settings in different environments to achieve the best runtime results. CompBS has the steepest curve among all solutions as it has the worst communication complexity, which is dominated by m instances of LE operations. TagBS uses only $\log m$ LE comparisons, but also invokes m multiplications, which make its curve relatively flat at small sizes and steep at larger sizes. LayHBS generates significantly more communication on WAN than on LAN at small sizes. This is because we used a larger $\tilde{\imath}$ on WAN, which means that a larger portion is processed by CompBS, inevitably increasing the communication cost. However, larger $\tilde{\imath}$ reduces the total number of communication rounds since CompBS has a constant round complexity, and round complexity is the bottleneck at small sizes on WAN. Thus, a larger $\tilde{\imath}$ can improve the overall runtime of LayHBS on WAN.

The curves of our SubHBS on LAN and WAN are significantly different because different settings are used, especially when the dataset size is large. As mentioned in Section 5.5.2, TagBS or LayHBS are of little use as sub-protocols of SubHBS on WAN because CompBS is faster for small to medium sizes. As a result, we mainly use CompBS in SubHBS on WAN, but TagBS and LayHBS on LAN. This makes the communication costs of SubHBS on WAN larger than that on LAN when the data set size is smaller than 2^{22}. For larger sizes, local computation becomes the dominating component of LAN experiments. The setting we use, i.e., $\alpha = 7$, is not the best choice for optimizing the communication cost, but it results in faster runtime as we can recursively use SubHBS for the second sub-search.

One can notice from Figure 5.6 that communication of SubHBS fluctuates as the size changes. This is due to the changes in the settings and the choices of sub-search protocols. More specifically, we used CompBS to do the second sub-search on LAN at small sizes and switched to TagBS and SubHBS at $m = 2^{12}$ and 2^{26}, respectively, which also reduced communication. Similarly, we replaced CompBS with a recursive invocation of SubHBS in the second and first sub-search on WAN at $m = 2^{20}$ and 2^{26} because CompBS's performance degrades at large sizes due to its high communication.

To summarize, our best protocols outperform alternative solutions for sizes up to $m = 2^{27}$ on LAN and $m = 2^{30}$ on WAN. SubHBS is up to $49\times$ faster than binary search based on best performing ORAM (optimized Floram) on LAN and up to $27\times$ on WAN.

5.5.3 Performance in the Malicious Model

To demonstrate that our constructions can be used in the malicious setting, we also evaluate our constructions in the malicious model on both LAN and WAN. Because relying on dot product protocols of constant communication is important for our constructions, we start with the techniques of Dalskov et al. [84] which have this property. The solution uses replicated secret sharing with three or four parties one of which is corrupt and we use the three party version. To evaluate performance when dot products involve linear communication, we use

SPD$\mathbb{Z}_{2^k}$ [85, 82]. It is in the dishonest majority setting and our experiments use two parties, which often gives the best performance. Lastly, we also run binary search experiments using maliciously secure ORAM [41], which is the only ORAM in the malicious model we are aware of, and use its BMR+Circuit ORAM variant with two parties. Because it is tree-based recursive ORAM, Gentry et al.'s optimization is applied in the experiments. All implementations are from MP-SPDZ [86] and ORAM is available only for $m \geq 2^{11}$.

Figure 5.7 and 5.8 show the results. The left plots with honest majority depict the total time, just like Figure 5.5. The right plots with dishonest majority show online and total runtime separately. The total runtime includes both online and estimated offline times, where the offline cost is computed based on the amount of precomputation and offline computation speed for our solutions in MP-SPDZ and the same triple generation speed is used for our protocols and Circuit ORAM. LayHBS is not shown in the right plot for clarity and has almost the same performance as TagBS for larger sizes.

Overall, our constructions show similar trends to those in Figure 5.5. Because the dot product has linear communication in the dishonest majority setting, both TagBS and SubHBS have linear communication, which narrows the gap between them. Due to the high latency, the gap is larger on WAN because SubHBS has a lower round complexity than TagBS. TagBS outperforms CompBS for $m > 2^8$ on LAN and $m > 2^{13}$ on WAN, with the biggest gap at $m = 2^6$ on LAN and $m = 2^9$ on WAN. Thus, we use $\tilde{\imath} = 6$ on LAN and $\tilde{\imath} = 9$ on WAN for our LayHBS in both malicious settings. Also interesting to note that the total time in the three-party honest majority setting is similar to the online time only with dishonest majority (with the exception of SubHBS when m is large, which is expected).

Online time of our dishonest majority SubHBS is up to 18 times smaller than that of Circuit ORAM on LAN and up to 2 times on WAN; Circuit ORAM becomes faster on LAN when m reaches 2^{21} (resp., 2^{20} on WAN). Circuit ORAM shows its advantage earlier on WAN because of its low round complexity, which is consistent with how it is reported in [41]. The total runtime of Circuit ORAM is up to 140 and 240 times slower than that of SubHBS on LAN and WAN, respectively, and performance of our construction and Circuit ORAM becomes similar at 2^{22}. One may also observe that the total runtimes on WAN are an order of magnitude slower than on LAN. This is because not only does the low bandwidth affect the online computation, but it also slows down precomputation needed to generate multiplication triples.

Chapter 6

Private and Accurate Floating-Point Summation

Floating-point numbers are the most widely used data type for approximating real numbers, and floating-point numbers are used in a wide variety of applications [87, 88, 89]. A well-known issue with floating-point arithmetic is that it is not exact. For example, summing two floating-point numbers can have a roundoff error and these roundoff errors can propagate and even become larger than a computed result when performing a sequence of many floating-point additions.

Calculating exact summation of n private floating-point numbers is a more challenging because of constraints of security and obliviousness requirements. In this chapter, we explore the problem of performing private floating-point summation in secure computation environment and develop an accurate and secure summation solution based on secret sharing.

In more details, given private floating-point number $\{[x_0], [x_1], \ldots, [x_{m-1}]\}$, we want to calculate a faithful rounding, one of the immediate floating-point neighbors, of their exact sum. In this work, we use a controversial way to represent a private floating-point number that is shared among parties. Instead of using a single integer, we represent a secret shared floating-point number x using a tuple $\langle [b], [p], [v] \rangle$ of secret shared integers such that $x = [b] \cdot [v] \cdot 2^{[p]}$, where $[b]$ represents the sign bit, $[p]$ represents the exponent, and $[v]$ represents the normalized mantissa.

6.1 Accurate Floating-Point Number Summation

In this section, we explore two potential solutions of calculating floating-point number summations, i.e., the expand-and-sum solution and superaccumulator solution. It is noted that the solutions discussed in this section are used to correctness verification and are not designed for the secure computation environment.

6.1.1 The Expand-and-Sum Solution

There is a simple solution for exactly summing m floating-point numbers, $\{x_1 ,x_2, \ldots, x_m\}$, which we refer to as the *expand-and-sum* solution. This solution is reasonable for low-precision floating-point representations.

Protocol 15 $s \leftarrow \mathsf{ExpandAndSum}(\{x_1, x_2, \ldots, x_m\})$

```
1: for each $x_i$ do
2:     $y_i \leftarrow \mathsf{ConvertToFixed}(x_i)$;
3:     $y_i \leftarrow 2^{u+e} \cdot y_i$   // convert to integer
4: end for
5: $t \leftarrow \sum_{i=1}^{m} y_i$;   // exact integer addition
6: $t \leftarrow t/2^{u+e}$   // convert to fixed-precision
7: $s \leftarrow \mathsf{ConvertToFloat}(t)$;
8: return $s$;
```

That is, for each floating-point number, x_i, we convert the representation of

x_i into a fixed-point number, y_i, with as many bits as is possible based on the type of floating-point being used for the x_i's. Then we sum these exactly using integer addition (viewing these numbers as integers with the decimal point understood) and then convert the result back into a floating-point number. For example, the y_i's would have the following number of bits based on the respective IEEE 754 formats:

- *Half*: a half-precision floating-point number in the IEEE 754 format has 1 sign bit, an 5-bit exponent, and a 10-bit mantissa. Thus, representing this as a fixed-point number requires $1 + 2^5 + 10 = 43$ bits.

- *Single*: a single-precision floating-point number in the IEEE 754 format has 1 sign bit, an 8-bit exponent, and a 23-bit mantissa. Thus, representing this as a fixed-point number requires $1 + 2^8 + 23 = 280$ bits.

- *Double*: a double-precision floating-point number in the IEEE 754 format has 1 sign bit, an 11-bit exponent, and a 52-bit mantissa. Thus, representing this as a fixed-point number requires $1 + 2^{11} + 52 = 2101$ bits.

- *Quad*: a quad-precision floating-point number in the IEEE 754 format has 1 sign bit, a 15-bit exponent, and a 112-bit mantissa. Thus, representing this as a fixed-point number requires $1 + 2^{15} + 112 = 32881$ bits.

Further, there are also even higher-precision floating-point representations, which would require even more bits to represent as fixed-precision numbers; see, e.g., [90, 78, 91, 92, 93]. Thus, implementing a summation method using a fixed-precision representation could require many operations, such as in a secret-sharing scheme, to be performed on very large numbers when applied to high-precision floating-point numbers. Of course, applications with high-precision floating-point numbers are likely to be applications that require accurate summations; hence, we desire solutions that can work efficiently for such applications without requiring ways of summing very large integers. For example, summing very large integers requires techniques for dealing with cascading carry bits during the summations, and performing all these operations securely is challenging for very large integers. Thus, since we desire a solution that can be implemented using off-the-shelf primitives for privacy and confidentiality, we consider this expand-and-sum approach for summing n floating-point numbers as fixed-point precision numbers to be limited to low-precision floating-point representations.

6.1.2 Superaccumulators

An alternative approach, which is better suited for use with off-the-shelf secure summation systems when applied to high-precision floating-point formats, is to use a a *superaccumulator* to represent floating-point summands, e.g., see [94, 54, 53]. This approach also uses integer arithmetic but with much smaller integers. More importantly, it avoids cascading carry-bit propagation, which is inefficient and vulnerable to timing attacks.

In a superaccumulator, rather than representing the bits of a floating-point number as a single expanded (very-large) integer, we represent that integer as a sum of small components, which we maintain separately. That is, we represent the expanded integer, y, representing a floating-point number, x, so that y is represented as a vector of $2w$-bit integers, $(y_h, y_{h-1}, \ldots, y_l)$, where

$$y = \sum_{j=l}^{h} 2^{wj} y_j,$$

such that l and h are chosen to cover all possible exponents. In our case, we choose w based on the system we plan to employ to achieve security and privacy. For example, if we have an off-the-shelf scheme that supports summations of 32-bit integers, then we can choose w to be 32.

In addition, we say that s is *regularized* if $-2^w < y_j < 2^w$, for all $j = l, \ldots, h$. In our scheme, we start with a regularized representation for each floating-point number, x_i, and after we perform groups of summations we regularize the partial sums. As we show, this approach allows us to limit how carries propagate after performing a group of sums, which allows us to achieve a constant number of rounds of communication for the secure constructions.

Suppose we are given n floating-point numbers, $\{x_1, x_2, \ldots, x_m\}$, each represented as a regularized superaccumulator,

$$x_i = \sum_{j=l}^{h} 2^{wj} y_{i,j}.$$

Further, suppose $n \leq 2^{w-2}$. We sum all the x_i's by first summing the corresponding terms,

$$s_j = \sum_{i=1}^{n} y_{i,j}.$$

Then, we split the binary representation of each s_j into c_{j+1} and r_j, so that

$$s_j = c_{j+1} 2^{w-1} + r_j,$$

where $-2^{w-1} < r_j < 2^{w-1}$. Then, we update each s_j as follows:

$$s_j \leftarrow r_j + c_j,$$

for $j = l, \ldots, h+1$. As we show, because of the way that we regularize superaccumulators, the "carry" values, c_j, will not propogate in a cascading way, and the result of the above summation will be regularized. This allows us to complete the sum in a single communication round.

Further, for practical values of w, the constraint that $n \leq 2^{w-2}$ is not restrictive. For example, if $w = 32$, this implies we can sum up to one billion floating-point numbers in a single communication round. For the case that $n > 2^{w-2}$, we could easily split the input set into multiple batch with size smaller than 2^{w-2} and calculate the sum superaccumulator separately.

Next, as shown in Theorem 7, we prove that the result of summing multiple regularize superaccumulators is accurate and will still be a regularize superaccumulator.

Theorem 7. *If $n \leq 2^{w-2}$, then summing n regularized superaccumulators using the above algorithm will produce a regularized result.*

Proof. Let $x_1, x_2, \ldots, x_m$ be the set of input superaccumulators to sum, where $n \leq 2^{w-2}$, and

$$x_i = \sum_{j=l}^{h} 2^{wj} y_{i,j},$$

for $i = 1, 2, \ldots, m$. Recall that we sum all the x_i's by summing the corresponding terms,

$$s_j = \sum_{i=1}^{m} y_{i,j}.$$

Since each x_i is regularized, $-2^w < y_{i,j} < 2^w$, for all i, j. Thus, $-2^w m < s_j <$

$2^w m$, for all j; hence, $-2^{2w-2} < s_j < 2^{2w-2}$, since $m \leq 2^{w-2}$. Recall that we split the binary representation of each s_j into c_{j+1} and r_j, so that

$$s_j = c_{j+1}2^{w-1} + r_j,$$

where $-2^{w-1} < r_j < 2^{w-1}$. Thus,

$$s_j = c_{j+1}2^{w-1} + r_j < c_{j+1}2^{w-1} + 2^{w-1} = (c_{j+1} + 1)2^{w-1} < 2^{2w-2}$$

and

$$s_j = c_{j+1}2^{w-1} + r_j > c_{j+1}2^{w-1} - 2^{w-1} = (c_{j+1} - 1)2^{w-1} > -2^{2w-2}.$$

Therefore,

$$-2^{w-1} + 1 < c_{j+1} < 2^{w-1} - 1,$$

for each j. So, when we update each s_j as $s_j \leftarrow r_j + c_j$, then

$$s_j = r_j + c_j < 2^{w-1} + 2^{w-1} - 1 = 2^w - 1$$

and

$$s_j = r_j + c_j > -2^{w-1} - 2^{w-1} + 1 = -2^w + 1.$$

Therefore, the result is regularized. $\square$

6.2 Secure Constructions for Floating-Point Number Summation

In this section, we propose our secure and accurate floating-point number summation protocol based on the superaccumulator structure proposed in Section 6.1.2. We assume the input floating-point number $[x_i]$ are in the form of a tuple $\langle[b_i]$, $[v_i]$, $[p_i]\rangle$, where $[b_i]$ represents the 1-bit sign bit, $[v_i]$ represents the m-bit mantissa, and $[p_i]$ represents the e-bit exponent.

Our superaccumulator-based construction is based on the following high-level idea: we first convert floating-point numbers into superaccumulators. Then, we calculate the sum of all superaccumulators in a fan-in manner and produce the result in a format of regularized superaccumulator. At the end, convert the result superaccumulator back to floating-point number.

6.2.1 High-level Ideas and New Building Blocks

Recall that the propagation of roundoff errors is the reason that leads to highly inaccurate results when summing multiple floating-point numbers. The straightforward idea is to convert floating-point numbers to fixed-point numbers and then sum in that format, which could avoid doing roundoff during the summation. However, this only works for low-precision floating-point numbers, such as half-precision floating-point numbers, which only need 43 bits to be represented as fixed-point numbers. A higher precision floating-point number would require more bits for the fix-point format. It would significantly increase computation efforts and inevitably introduce carry-propagation during the computation, as no native primitive data types support such a long bit length integer.

Protocol 16 $[s] \leftarrow \mathsf{FLSumLarge}(\langle [b_0], [v_0], [p_0] \rangle, \ldots, \langle [b_{m-1}], [v_{m-1}], [p_{m-1}] \rangle, e, u, w)$

```
1: $\alpha = \lceil \frac{2^e+u}{w} \rceil, \beta = \lceil \frac{u+1}{w} \rceil + 1;$
2: for $i = 0, \ldots, m-1$ in parallel do
3:     $\langle [y_{i,\alpha-1}], \ldots, [y_{i,0}] \rangle \leftarrow \mathsf{FL2SA}([b_i], [v_i], [p_i], e, u, w, \alpha, \beta);$
4: end for
5: $\langle [y_{\alpha-1}], \ldots, [y_0] \rangle \leftarrow \mathsf{SuperSum}(\langle [y_{0,\alpha-1}], \ldots, [y_{0,0}] \rangle, \ldots, \langle [y_{m-1,\alpha-1}], \ldots, [y_{m-1,0}] \rangle);$
6: $\langle [b'_0], [v'_0], [p'_0] \rangle \leftarrow \mathsf{SA2FL}(\langle [y_{\alpha-1}], \ldots, [y_0] \rangle);$
7: return $\langle [b'_0], [v'_0], [p'_0] \rangle;$
```

Protocol 17 $\langle [b_0], \ldots, [b_{u-1}] \rangle \leftarrow \mathsf{B2U}([a], u, k)$

```
1: $q = \lceil \log u \rceil;$
2: $[r], [r_{q-1}]_1, \ldots, [r_0]_1 \leftarrow \mathsf{edaBit}(q)$
3: $\langle [b_0]_1, \ldots, [b_{2^q-1}]_1 \rangle \leftarrow$
   $\mathsf{AllOr}([r_{q-1}]_1, \ldots, [r_0]_1);$
4: for $i = 0, \ldots, u-1$ in parallel do
5:     $[b_i]_1 = 1 - [b_i]_1;$
6:     $[b_i] = \mathsf{B2A}([b_i]_1, k);$
7: end for
8: $c \leftarrow \mathsf{Open}_{2^q}([a] + \sum_{i=0}^{q-1} 2^i [r_i]);$
9: for $i = 0, \ldots, u-1$ in parallel do
10:     $[b_i] = [b_{(c-i) \bmod 2^q}];$
11: end for
12: return $\langle [b_0], \ldots, [b_{u-1}] \rangle;$
```

Thus, we propose the superaccumulator-based solution to achieve secure and accurate floating-point number summation.

Our summation solution is based on the superaccumulator structure, which requires a conversion for all input floating-point numbers. It is non-trivial to design the conversion from the floating-point number into the format of superaccumulator. We need to make sure that the conversion procedure is input independent so that no information about the input data will be leaked. While ensuring the obliviousness, we would like to improve the efficiency of the conversion since the performance of this conversion is the bottleneck of the entire computation. As we need to process each input floating-point number, the cost of this stage depends linearly on the size of the input.

The high-level idea of the conversion from floating-point number into superaccumulator is that we write the mantissa of a floating-point number into the appropriate blocks in the superaccumulator. Because the position of the mantissa in the superaccumulator depends on its exponent, the writing procedure should be oblivious to prevent any information leakage about the updated position in the superaccumulator. For obliviously writing a value into a private location in a superaccumulator, we generate a tag vector with the same size of the superaccumulator, where the tag representing the target position will be $[1]$ to and others are $[0]$. We design the building block $\mathsf{B2U}$) to convert a private index into the corresponding tag vector.

Binary to unary representation $\langle [b_0], \ldots, [b_{u-1}] \rangle \leftarrow \mathsf{B2U}([a], u, k)$. This is a conversion procedure from a binary to unary representation. $\mathsf{B2U}$ converts the private argument integer $[a]$ into an array of bits, where only the $[a]$-th bit is $[1]$ and all others are set to $[0]$. It is noted that our $\mathsf{B2U}$ is modified based on the $\mathsf{ArrayAccess}$ protocol in [95], as shown in Protocol 17, where the $\mathsf{PRandBit}$ is replaced by edaBit. The main improvement is that the subsequent AllOr is now working with binary shares instead of arithmetic shares, which would reduce the corresponding communication cost.

During the computation, we may need to do conversion from binary shares to arithmetic shares. We also design a new protocol for binary to arithmetic conversion $\mathsf{B2A}$ that works in the three-party setting.

Binary to Arithmetic Conversion $[r]_k \leftarrow \mathsf{B2A}([r]_1, k)$. This building block converts a binary share $[r]_1$ in $\mathbb{Z}_2$ into its corresponding arithmetic share $[r]_k$ within a given ring $\mathbb{Z}_{2^k}$. This protocol is designed to work only with three-party setting, but offers superior performance compared to the general construction. In what follows, we use notion $\mathsf{G_i}$ to denote a pseudorandom generator, which is initialized by a secret seed $\mathsf{key_i}$. Because a replicated secret shared value $[r]$ is split into three shares, i.e, $\langle [r]^{(1)}, [r]^{(2)}, [r]^{(3)} \rangle$, we let the notation include the number of each share. Each party P_j holds $\{[r]^{(i)} | i \in [1,3], i \neq j\}$.

In our construction, the input of B2A is a secret shared bit in $\mathbb{Z}_2$ and an integer k that represents the bit length of the result arithmetic share in $\mathbb{Z}_k$. We use the fact that $r = [r]^{(1)} \oplus [r]^{(2)} \oplus [r]^{(3)}$, which is calculated via a custom procedure in two rounds. In particular, we initialize three secret shared variables, i.e., $[r1]_k = \langle [r]_1^{(1)}, 0, 0 \rangle$, $[r2]_k = \langle 0, [r]_1^{(2)}, 0 \rangle$, and $[r3]_k = \langle 0, 0, [r]_1^{(3)} \rangle$. It is easy to see that $r = [r1]_k \oplus [r2]_k \oplus [r3]_k$, which could be calculated mainly with two rounds of multiplication. This, however, appears redundant because two of the shares of these variables are zero, which means that most of the parties' the local computation results will be zero and makes the subsequent re-sharing meaningless. For example, for the computation $[r1]_k \cdot [r2]_k$, P_1 and P_2's local computation will be zero as they don't hold the $[r]^{(1)}$ and $[r]^{(2)}$, respectively. Thus, we simplified the two rounds of multiplication, which reduces the communication cost to only one element per party.

As shown in Protocol 18, the secret variables $[r1]_k$, $[r2]_k$, and $[r3]_k$ are formed with shares of input $[r]_1$ and shared among three parties. In addition, each party i also holds two pseudorandom generators, i.e., $\{\mathsf{G}_j | j \in [1,3], i \neq j\}$. For calculating the first xor operation $[r1]_k \oplus [r2]_k$, we first calculate the multiplication $[s]_k = [r1]_k \cdot [r2]_k$ as shown in Protocol 18 Line 3-5. Because only $P3$ holds the substantial shares of $[r1]_k$ and $[r2]_k$, it could generate the multiplication result locally and share the result to other parties. It is noted that $P3$ only needs to send $[s]_k^{(1)}$ to P_2, because P_1 is able to generate $[s]_k^{(2)}$ using G_2. After this step, all parties hold their corresponding shares of $[s]_k$ and continue to compute $[s']_k$ in local, which is the result of $[r1]_k \cdot [r2]_k$. For calculating the second xor operation that $[s']_k \oplus [r3]_k$, P_1 and P_2 compute $[s']_k^{(2)} \cdot [r]_1^{(3)}$ and $[s']_k^{(1)} \cdot [r]_1^{(3)}$ separately and re-share their results with other parties as shown in Protocol 18 Line 8-10. Af-

Protocol 18 $[r]_k \leftarrow \mathsf{B2A}([r]_1, k)$

```
1: Setup: $[r1]_k, [r2]_k, [r3]_k$ consist of three shares, i.e., $\langle [r]_1^{(1)}, 0, 0\rangle$, $\langle 0, [r]_1^{(2)}, 0\rangle$,and $\langle 0, 0, [r]_1^{(3)}\rangle$, respectively; Party $Pi$ holds $\{[r]_1^{(j)}, \mathsf{G}_j | j \in [1,3], i \neq j\}$.
2: $[s]_k = [r1]_k \cdot [r2]_k$:
    3: $P_3$ computes $[r]_1^{(1)} \cdot [r]_1^{(2)}$, sets $[s]_k^{(2)} = \mathsf{G}_1.\mathsf{next}$, $[s]_k^{(1)} = [r]_1^{(1)} \cdot [r]_1^{(2)} - \mathsf{G}_2.\mathsf{next}$ and sends $[s]_k^{(1)}$ to $P_2$;
    4: $P_2$ sets $[s]_k^{(1)}$ to the received value and $[s]_k^{(3)} = 0$;
    5: $P_1$ computes $[s]_k^{(2)} = \mathsf{G}_2.\mathsf{next}$ and set $[s]_k^{(3)} = 0$;
6: $[s']_k = [r1]_k + [r2]_k - 2[s]_k$;
7: $[t]_k = [s']_k \cdot [r3]_k$:
    8: $P_2$ computes $g_1 = \mathsf{G}_1.\mathsf{next}$, $g_3 = \mathsf{G}_3.\mathsf{next}$, and $[t'] = [s']_k^{(1)} \cdot [r]_1^{(3)} - g_1$. Then, set $[t]_k^{(1)} = g_1$ and $[t]_k^{(3)} = [t'] + g_3$, and sends $[t']$ to $P_1$;
    9: $P_1$ sets $[t']$ to the received value and sample $g_3 = \mathsf{G}_3.\mathsf{next}$. Then, $P_1$ computes $[t]_k^{(2)} = [s']_k^{(2)} \cdot [r]_1^{(3)} - g_3$ and $[t]_k^{(3)} = [t'] + g_3$, and sends $[t]_k^{(2)}$ to $P_3$;
    10: $P_3$ sets $[t]_k^{(2)}$ to the received value and sample $[t]_k^{(1)} = \mathsf{G}_1.\mathsf{next}$.
11: $[r]_k = [s']_k + [r3]_k - 2[t]_k$;
12: **return** $[r]_k$
```

ter another round of local computation, the final result $[r]_k$ is shared among all three parties.

The cost of our B2A is mainly communicating one ring element per party in two rounds, where the communication cost is as same as the cost of a multiplication.

6.2.2 Conversion from a Floating-Point Number to a Superaccumulator

Given a normalized floating-point number $[x] = [b] \cdot [v] \cdot 2^{[p]}$, we first convert it into the superaccumulator format. A regularized superaccumulator is a vector that consists of α w-bit integers, where $\alpha = \lceil \frac{2^e + u}{w} \rceil$. We first locate where

Protocol 19 $[s] \leftarrow \mathsf{FL2SA}([b], [v], [p], e, u, w, \alpha, \beta)$

```
1: $[p^{high}] \leftarrow \mathsf{Trunc}([p], e, \log w)$;
2: $[p^{low}] = [p] - [p^{high}] \cdot w$;
3: $[z] = \mathsf{EQZ}([p])$;
4: $[v] = [v] + 2^u \cdot (1 - [z])$;
5: $\langle [v_i^{(0)}], \ldots, [v_i^{(\beta-1)}] \rangle \leftarrow \mathsf{FL2SAPart1}([b], [v], [p^{low}], e, u, w, \beta)$;
6: $\langle [y_{\alpha-1}], \ldots, [y_0] \rangle \leftarrow \mathsf{FL2SAPart2}(\langle [v_i^{(0)}], \ldots, [v_i^{(\beta-1)}] \rangle, [p^{high}], e, u, w, \alpha, \beta)$;
7: **return** $[s] = \langle [y_{\alpha-1}], \ldots, [y_0] \rangle$;
```

the floating-point number's mantissa will fall in the superaccumulator. Then, we split the the mantissa $[v]$ into multiple pieces and write them into the corresponding blocks of the initially empty superaccumulator.

Our solution for converting a normalized floating-point number into a superaccumulator is given as Protocol 19. We first truncate the exponent $[p]$ into two parts, i.e., $[p^{low}]$ and $[p^{high}]$, where the high-order $e - \log w$ bits, denoted as $[p^{high}]$, indicates the index of the first chunk in a superaccumulator that the mantissa fall in and the low-order $\log w$ bits, , denoted as $[p^{low}]$, indicates the position of the least significant bit of $[v]$ in the corresponding chunk. In addition, we check whether $[p]$ is zero. If $[p]$ is not a zero[1], it indicates that the input floating-point number is normalized and we need to denormalize it in the subsequent operation. Next, we describe our conversion in two separate parts, i.e., dividing the mantissa into multiple pieces in Protocol 20 and writing them into the superaccumulator in Protocol 21.

As described in Protocol 20, we first left shift the mantissa $[v]$ by the private offset $[p_i^{low}]$ so that $[v'] = [v] \cdot 2^{[p_i^{low}]}$. This shift could move the bits in the mantissa to their proper position in the superaccumulator, as shown in Figure 6.1. Then, we split the $[v']$ into chunks of w-bit starting from the least significant bits.

[1]Although we assume that the input is normalized floating-point number, this check is still necessary as there is a case that a number could be too small to be normalized, in which its exponent will be zero.

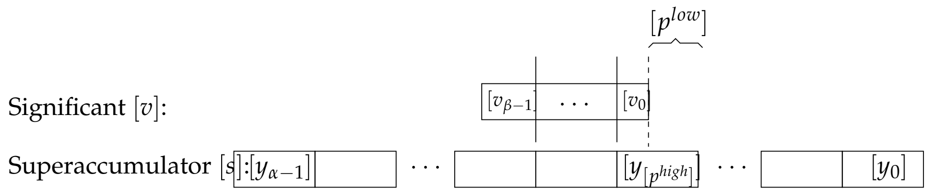

Figure 6.1: Illustration of converting floating-point number into superaccumulator.

Protocol 20 $\langle [v_0], \ldots, [v_{\beta-1}] \rangle \leftarrow \mathsf{FL2SAPart1}([b], [v], [p^{low}], e, u, w, \beta)$

```
1: $[v'] \leftarrow \mathsf{LeftShift}([v], [p^{low}], \log w)$;
2: $\{[v_0], \ldots, [v_{\beta-1}]\} = \mathsf{Split}([v'], \beta, w, u)$;
3: for $i = 0, \ldots, \beta - 1$ in parallel do
4:     $[v_i] \leftarrow ([1] - 2 \cdot [b]) \cdot [v_i]$;
5: end for
6: return $\langle [v_0], \ldots, [v_{\beta-1}] \rangle$;
```

In particular, we recursively truncate w least significant bits from the $[v']$ until the bit length of the rest part is less than w. As the mantissa may fall into more than one piece of chunks in the superaccumulator, we use β to denote the max possible of chunks that the mantissa may affect, where $\beta = \lceil \frac{u+1}{w} \rceil + 1$. To ensure the obliviousness of the computation, we always split the $[v']$ into β parts. At the end, we multiply the sign bit $([1] - 2 \cdot [b])$ with each $[v_i]$ before writing them into the superaccumulator.

We next describe how to obliviously write $[v_i]$ into the corresponding block of a superaccumulator $[s] = \langle [y_{\alpha-1}], \ldots, [y_0] \rangle$. Although we already extracted $[p^{high}]$ from which we could calculate each $[v_i]$'s corresponding location in $[s]$, it is non-trivial to write in a private array with a private index. We cannot reveal $[p^{high}]$ to any party as it will leak information about the private input floating-point number. To obliviously write $[v_i]$ into a superaccumulator, we first use B2U with input $[p^{high}]$ to generate an array $\langle [d_{\alpha-1}], \ldots, [d_0] \rangle$, where only the $[d_{[p^{high}]}]$ is $[1]$ and all other elements are $[0]$. In particular, as shown in Protocol 21, we calculate a dot product for each block $[y_i]$ to get its updated value.

Protocol 21 $\langle [y_\alpha], \ldots, [y_0] \rangle \leftarrow \mathsf{FL2SAPart2}(\langle [v_0], \ldots, [v_{\beta-1}] \rangle, [p^{high}], e, u, w, \alpha, \beta)$

```
1: $[s] = \langle [y_{\alpha-1}], \ldots, [y_0] \rangle$;
2: $\langle [d_{\alpha-1}], \ldots, [d_0] \rangle = \mathsf{B2U}([p^{high}], \alpha)$;
3: for $i = 0, \ldots, \alpha - 1$ in parallel do
4:     if $0 <= i <= \beta - 2$ then
5:         $[y_i] \leftarrow \sum_{k=0}^{i} [d_{i-k}] \cdot [v_k]$;
6:     end if
7:     if $\beta - 1 <= i <= \alpha - \beta$ then
8:         $[y_i] \leftarrow \sum_{k=0}^{\beta-1} [d_{i-k}] \cdot [v_k]$;
9:     end if
10:    if $\alpha - \beta + 1 <= i <= \alpha - 1$ then
11:        $[y_i] \leftarrow \sum_{k=0}^{\alpha-1-i} [d_{i-\beta+1+k}] \cdot [v_{\beta-1-k}]$;
12:    end if
13: end for
14: return $\langle [y_\alpha], \ldots, [y_0] \rangle$;
```

Protocol 22 $\{[z_{k-1}], \ldots, [z_0]\} \leftarrow \mathsf{Split}([a], k, w, u)$

```
1: $[a^{(0)}] = [a]$;
2: for $i = 0, \ldots, k - 2$ in parallel do
3:     $[a^{(i+1)}] \leftarrow \mathsf{Trunc}([a^{(i)}], u - i \cdot w, w)$;
4:     $[z_i] = [a^{(i)}] - [a^{(i+1)}] \cdot 2^w$;
5: end for
6: $[z_{k-1}] = [a^{(k-1)}]$;
7: return $\{[z_{k-1}], \ldots, [z_0]\}$;
```

If i is in range $[[p^{high}], [p^{high}] + \beta - 1]$, $[y_i]$ will be updated to its corresponding value. Otherwise, $[y_i]$ remains zero. For example, the block $[y_0]$ may be updated to $[v_0]$ only if $[d_0] = [1]$, while block $[y_1]$ may be updated to $[v_0]$ or $[v_1]$ in the case of $[d_1] = [1]$ or $[d_0] = [1]$, respectively. Overall, we could batch all the dot product operations in a single round with a communication cost of α ring elements.

6.2.3 Superaccumulator Summation

We next present the solution for summing over superaccumulators in Protocol 23. After the conversion, all the n floating-point numbers $\langle [b_0], [v_0], [p_0] \rangle$, …, $\langle [b_{m-1}], [v_{m-1}], [p_{m-1}] \rangle$ have been converted to the format of superaccumulator $\langle [y_{0,\alpha-1}]$, …, $[y_{0,0}] \rangle, \ldots, \langle [y_{m-1,\alpha-1}], \ldots, [y_{m-1,0}] \rangle$. Because the bits of the

Protocol 23 $\langle [y_{\alpha-1}], \ldots, [y_0] \rangle \leftarrow \mathsf{SuperSum}(\langle [y_{0,\alpha-1}], \ldots, [y_{0,0}] \rangle, \ldots, \langle [y_{m-1,\alpha-1}], \ldots, [y_{m-1,0}] \rangle)$

```
1: for $i = 0, \ldots, \alpha - 1$ in parallel do
2:     $[y_i] = \sum_{j=0}^{m-1} [y_{j,i}]$;
3:     $[b_i] = \mathsf{MSB}([y_i])$;
4:     $[y'_i] = [y_i] \cdot ([b_i] - 1)$;
5:     $[c_{i+1}] = \mathsf{Trunc}([y'_i], 2w, w)$;
6:     $[r_i] = [y'_i] - [c_{i+1}] \cdot 2^w$;
7:     $[y_i] = [r_i] \cdot [b_i] + [c_i] \cdot [b_{i-1}]$
8: end for
9: return $\langle [y_{\alpha-1}], \ldots, [y_0] \rangle$;
```

floating-point number's mantissa are already aligned according to the exponent, the summation of n superaccumulators can be easily achieved by calculating the sum of all corresponding elements with the same index, i.e., $[y_i] = \sum_{i=0}^{m-1} [y_{i,j}]$ for $i \in [0, \alpha - 1]$. Once the summation is done, we proceed to regularize the result superaccumulator. As shown in Protocol 23 Line 3, we first truncate the each superaccumulator block $[y_i]$ to calculate the carry value $[c_{i+1}]$ and rest value $[r_i]$. Then, we could compute the regularized $[y_i]$ by adding $[r_i]$ with carry value from previous block. As proven in Section 6.1.2, the result superaccumulator are already regularized.

It is noted that the ring size will affect the number of superaccumulators that can be processed in one batch. If computation is over $\mathbb{Z}_{2^{w+w'}}$, for instance, the $\mathsf{SuperSum}$ can only calculate the sum of $2^{w'}$ inputs. Calculating summation for more than $2^{w'}$ inputs may cause overflow in some blocks as the size of bit length of each input block is w. For the case that n is greater than $2^{w'}$, a straightforward option is to use a larger ring size to process all input in a single batch, which could reduce the round complexity. However, using a smaller ring size might not always be a bad option as using a smaller ring size could potentially reduce the runtime of local operation as well as communication cost.

Protocol 24 $\langle[b],[p],[v]\rangle \leftarrow \mathsf{SA2FL}(\langle[y_{\alpha-1}],\ldots,[y_0]\rangle)$

```
1:  for $i = \beta, \ldots, \alpha-1$ in parallel do
2:      $[c_i] = 1 - \mathsf{EQZ}([y_i])$;
3:  end for
4:  $[d_{\alpha-1}] = [c_{\alpha-1}]$;
5:  for $i = \alpha-2, \ldots, \beta$ do
6:      $[d_i] = [d_{i+1}] + [c_i]$;
7:  end for
8:  for $i = \beta, \ldots, \alpha-2$ in parallel do
9:      $[c'_i] = 1 - \mathsf{EQZ}([d_i])$;
10: end for
11: $[c'_{\alpha-1}] = [c_{\alpha-1}]$;
12: for $i = \beta, \ldots, \alpha-2$ in parallel do
13:     $[d'_i] = [c'_i] - [c'_{i+1}]$;
14: end for
15: $[d'_{\beta-1}] = [1] - [d_\beta]$;
16: $[d'_{\alpha-1}] = [c_{\alpha-1}]$;
17: for $i = 0, \ldots, \beta-1$ in parallel do
18:     $[v'_i] \leftarrow \sum_{j=i}^{\alpha-\beta+i} [d'_{j+\beta-1-i}] \cdot [y_j]$;
19:     $[v''_i] \leftarrow \mathsf{Convert}([v'_i], 2w, w\beta)$;
20: end for
21: $[v'] = \sum_{i=0}^{\beta-1} [v''_i] \cdot 2^{w \times i}$;
22: $\langle[b],[p'],[v]\rangle \leftarrow \mathsf{Normalize}([v], w\beta, u)$;
23: $[p] \leftarrow [p'] + \sum_{i=0}^{\alpha-\beta} [d'_{i+\beta-1}] \cdot i \cdot w$;
24: return $\langle[b],[p],[v]\rangle$;
```

6.2.4 Conversion from a Superaccumulator to a Floating-Point Number

We next describe the last phase, Protocol 24, in our superaccumulator based floating-point number summation solution. Once the final superaccumulator is regularized, we need to convert it back to the format of floating-point number. The high-level idea of this conversion is that we extract β blocks starting from the first non-zero block. If all blocks in $\langle[y_{\alpha-1}],\ldots,[y_0]\rangle$ are zero or the number of eligible blocks is less than β, the computation will return the $\langle[y_{\beta-1}],\ldots,[y_0]\rangle$ for privacy concerns. This will make sure that the number of extracted blocks is always β.

Without loss of generality, we denote k as the index of the first non-zero block, where $k \in [\alpha-1,\beta]$. Given a regularized superaccumulator $\langle[y_{\alpha-1}],\ldots,[y_0]\rangle$, we check all blocks in $\langle[y_{\alpha-1}],\ldots,[y_\beta]\rangle$ to find out the first non-zero block. We first execute EQZ on all blocks and then locally calculate prefix-sum $\langle[d_{\alpha-1}],\ldots,[d_\beta]\rangle$ over the EQZ results staring from $[c_{\alpha-1}]$ towards $[c_\beta]$. The prefix-sum results

Protocol 25 $\langle[b],[v'],[p]\rangle \leftarrow \mathsf{Normalize}([v],k,k')$

```
1:  $[b'] \leftarrow \mathsf{MSB}([v])$;
2:  $[b] = 1 - 2[b']$;
3:  $[v'] = [b] \cdot [v]$
4:  $\langle [v'_{k-1}], \ldots, [v'_0] \rangle \leftarrow \mathsf{BitDec}([v'], k)$;
5:  $[d_{k-1}] = [v'_{k-1}]$;
6:  for $i = k-2, \ldots, k'+1$ do
7:      $[d_i] = [v'_i] + [d_{i+1}]$;
8:  end for
9:  for $i = k'+1, \ldots, k-2$ in parallel do
10:     $[c_i] = 1 - \mathsf{EQZ}([d_i])$;
11: end for
12: $[c_{k-1}] = [v'_{k-1}]$;
13: for $i = k'+1, \ldots, k-2$ in parallel do
14:     $[d'_i] = [c_i] - [c_{i+1}]$;
15: end for
16: $[d'_{k'}] = 1 - [c_{k'+1}]$;
17: $[d'_{k-1}] = [c_{k-1}]$;
18: for $i = 0, \ldots, k'-1$ in parallel do
19:     $[u_i] \leftarrow \sum_{j=i}^{k-k'-1+i} [d'_{i+k'-i}] \cdot [v'_j]$;
20: end for
21: $[v'] = \sum_{i=0}^{k'-1} [u_i] \cdot 2^i$;
22: $[d'_{k'}] = [d'_{k'}] \cdot [v'_{k'}]$;
23: $[p] = \sum_{i=0}^{k-k'-1} i \cdot [d'_{j+k'}]$
24: return $\langle [b], [v'], [p] \rangle$;
```

$\langle [d_{\alpha-1}], \ldots, [d_{k+1}] \rangle$ remain zero, while all the rests become non-zero values. We continue executing EQZ on $\langle [d_{\alpha-1}], \ldots, [d_{\beta}] \rangle$ again and computing the tag array $\langle [d'_{\alpha-1}], \ldots, [d'_{\beta-1}] \rangle$ as describe in Protocol 24 Line 11-15. In the end, we get a tag array that only the k-th element is $[1]$ and all others are $[0]$. The $[d'_{\beta-1}]$ will be set as $[1]$ obliviously if all the $\langle [d'_{\alpha-1}], \ldots, [d'_{\beta}] \rangle$ are zero. Next, we could extract the blocks $\langle [y_k], \ldots, [y_{k-\beta+1}] \rangle$ along with the exponent $[p']$.

After extracting the β blocks, we assembly them into a single integer $[v'] = \sum_{i=0}^{\beta-1} [v'_i] \cdot 2^{w \times i}$ and normalize $[v']$ to get the final results.

6.3 Performance Evaluation

In this section, we demonstrate the performance of our proposed constructions and compare it with the floating-point summation protocol in [74]. We implemented the protocols in C++ using the ring setting. We run all experiments in a three-party setting, where each machine has a 8-core 2.1GHz CPU and 64GB of

Table 6.1: Performance of Superaccumulator-based floating-point summation in milliseconds

Protocol	Setting	Input size					
		2^4	2^6	2^8	2^{10}	2^{12}	2^{14}
FL2SA	Single	6.98	8.56	15.1	42.1	125	467
	Double	11.3	16.6	34.7	121	405	1661
SuperSum	Single	1.57	1.56	2.23	2.19	2.13	3.09
	Double	2.56	2.67	2.74	3.17	3.91	8.98
SA2FL	Single	7.55	7.51	7.4	7.13	6.95	6.96
	Double	8.87	8.48	8.75	8.6	8.13	8.41
Total	Single	16.1	17.7	25.4	51.4	135	478
	Double	22.7	27.8	46.2	132	417	1679

(a) Performance with $w = 16$

Protocol	Setting	Input size							
		2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
FL2SA	Single	5.94	8.37	16.5	46.5	143	566	2294	8962
	Double	9	14.2	32.5	108	379	1501	6198	23775
SuperSum	Single	1.72	1.71	2.45	2.17	2.26	2.95	4.07	9.31
	Double	2.64	2.81	2.93	3.12	3.96	5.56	12.9	41.5
SA2FL	Single	8.44	8.41	8.36	8.17	8.11	8.16	8.17	7.97
	Double	9.76	9.57	9.41	9.51	9.54	9.47	9.28	9.61
Total	Single	16.1	18.5	27.4	56.8	154	578	2307	8980
	Double	21.4	26.7	44.8	121	393	1517	6220	23826

(b) Performance with $w = 32$

RAM. All experiments only used a single thread. The machines were connected in a LAN network by 1 Gbps Ethernet link with one-way latency of 0.08ms.

The performance of our superaccumulator based floating-point number summation protocols are shown in Figure 6.2 and Figure 6.3. Additional numbers are available in Table 6.1. From the break down of the runtime in both single and double precision experiments, the bottleneck of the summation is the conversion FL2SA, especially when the input size is large. This is as expected because we need to convert all input floating-point number into the format of superaccumulator. In contrast, the conversion from superaccumulator back to floating-point number SA2FL has a constant runtime for all batch sizes, which is

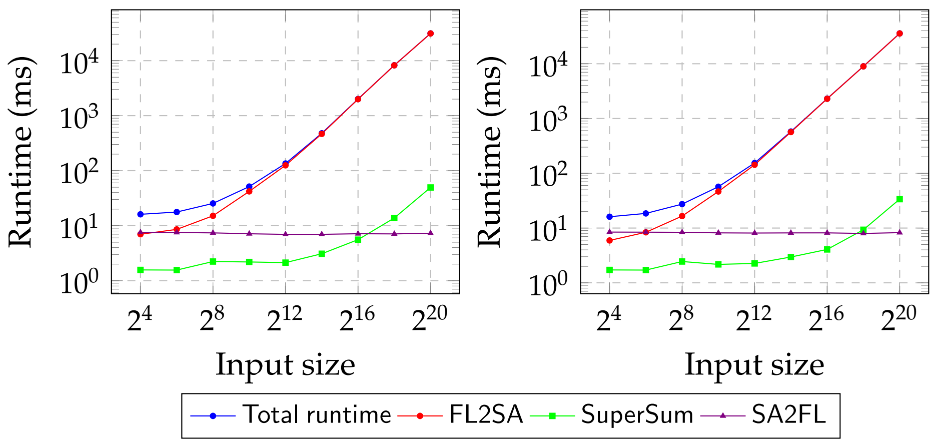

Figure 6.2: Performance of Superaccumulator-based floating-point summation for single precision with $w = 16$ (left) and $w = 32$ (right).

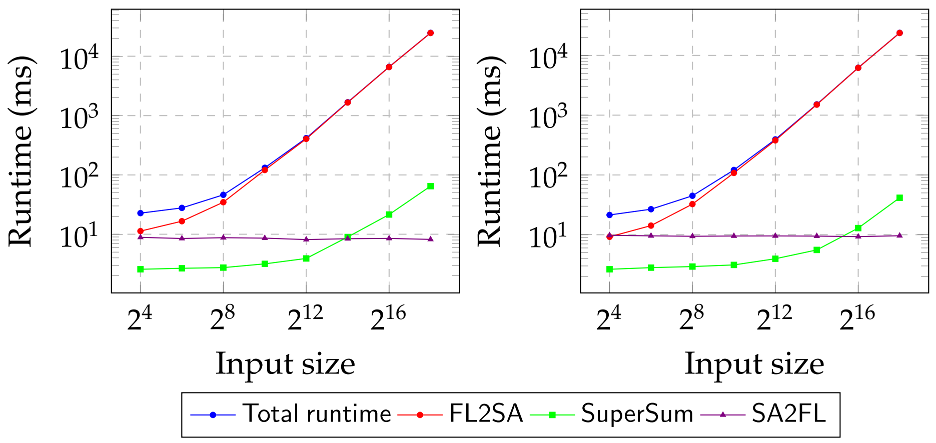

Figure 6.3: Performance of Superaccumulator-based floating-point summation for double precision with $w = 16$ (left) and $w = 32$ (right).

because we only need to convert the final superaccumulator so that the workload of SA2FL is constant through all experiments. Although our superaccumulator summation protocol SuperSum has constant online complexity, its local computation complexity linearly depends on the input size, which makes its runtime increase as the input sizes.

Comparing the experiments using different values of w, using $w = 16$ could bring a better overall runtime for the single precision experiments, while the double precision experiments recorded the better performance with $w = 32$.

Protocol	Precision	Input size			
		10	20	50	100
Ours	Single	15.3	16.1	17.2	19.2
	Double	20.7	22.2	25.2	29.4
[75]	Single	19.3	33.1	71.6	136
	Double	34.7	55.9	118.6	198

Table 6.2: Performance comparison (ms)

The performance differences from different choice of w mainly come from the conversion FL2SA. As the value of parameters α and β directly depends on the precision and w, a larger w will bring a smaller α and a larger β. In other words, increasing the value of w will increase the complexity of Protocol 20 but decreasing the complexity of Protocol 21. Thus, for a better over all runtime, the choice of w should be decided based on the computational environment as well as the target precision.

We compare the performance of our superaccumulator based summation protocols with the reported runtime of a floating-point summation protocol from [74, 75]. The experiments in [75] were run on 3 computers with 3.6 GHz CPU connected by 1 Gbps Ethernet LAN. As shown in Table 6.2, although our experiments were run on slower machines, the best of ours still has a better runtime. The experiments with input size 100 recorded the largest gap, where ours is 7 and 6.7 times faster for the single and double precision, respectively. We believe the advantage of ours will be larger as the batch size increases.

Chapter 7

Conclusion

We conclude this dissertation in this chapter.

7.1 Private Array Access

We present optimized protocols for reading or writing an element of an array at a private index. Most of our constructions are based on Shamir secret sharing with the exception of one array access construction. The latter uses 2-out-of-2 additive secret sharing in the three-party setting with honest majority, but offers superior performance compared to general constructions. To be compatible with computation based on Shamir secret sharing, we also provide conversion procedures to convert between the two representations. We implement the presented constructions in the setting with three computational parties and show that they offer attractive performance in both LAN and WAN settings.

7.2 Private Binary Search

We design a suite of binary search protocols with different properties and structure in the multi-party setting based on secret sharing. In addition, we further improve our solutions to obtain hybrid schemes which outperform the individual constructions and lower binary search communication from $O(m)$ in our prior constructions to $O(\sqrt{m})$ for a dataset of size m. Our performance evaluation demonstrates that our solutions outperform existing ORAM constructions for dataset sizes up to a billion, even after optimizations to improve performance of ORAM schemes specifically in the context of binary search.

7.3 Private and Accurate Floating-point Summation

We propose a superaccumulator based floating-point number summation protocol by applying the oblivious data access solution. The proposed solution is the first one that we are aware of that could calculate the faithful rounding of the exact sum of many private floating-point numbers in secure computation. Our solution could calculate the exact summation of unlimited number of floating-point numbers within constant rounds. Performance evaluations show that our solution outperforms existing work while providing better accuracy.

Bibliography

[1] A. Yao. Protocols for secure computations. In *Symposium on Foundations of Computer Science*, pages 160–164, 1982.

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

[3] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[4] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267. Springer, 2009.

[5] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay-secure two-party computation system. In *USENIX Security Symposium*. San Diego, CA, USA.

[6] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *International workshop on public key cryptography (PKC)*, pages 160–179. Springer, 2009.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, 2010.

[8] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE symposium on security and privacy*, pages 334–348. IEEE, 2013.

[9] B. Hemenway, S. Lu, R. Ostrovsky, and W. Welser Iv. High-precision secure computation of satellite collision probabilities. In *International Conference on Security and Cryptography for Networks*, pages 169–187. Springer, 2016.

[10] D. Bogdanov, L. Kamm, S. Laur, P. Pruulmann-Vengerfeldt, R. Talviste, and J. Willemson. Privacy-preserving statistical data analysis on federated databases. In *Annual Privacy Forum*, pages 30–55. Springer, 2014.

[11] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(4):345–364, 2017.

[12] E. A. Abbe, A. E. Khandani, and A. W. Lo. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review*, 102(3):65–70, 2012.

[13] D. Bogdanov, L. Kamm, B. Kubo, R. Rebane, V. Sokk, and R. Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016(3):117–135, 2016.

[14] R. Ostrovsky. Efficient computation on oblivious RAMs. In *ACM STOC*, pages 514–523, 1990.

[15] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

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

[17] M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structures. In *IEEE Globecom*, pages 99–102, 1987.

[18] M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 377–397, 2020.

[19] A. Baccarini, M. Blanton, and C. Yuan. Multi-party replicated secret sharing over a ring with applications to privacy-preserving machine learning. IACR Cryptology ePrint Archive Report 2020/1577, 2020.

[20] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, pages 823–852, 2020.

[21] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[22] E. Boyle and M. Naor. Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 357–368, 2016.

[23] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM STOC*, pages 182–194, 1987.

[24] E. Shi, T-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[25] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)*, pages 299–310, 2013.

[26] S. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)*, pages 513–524, 2012.

[27] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, pages 1–18, 2013.

[28] X. S. Wang, Y. Huang, T. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 191–202, 2014.

[29] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 523–535, 2017.

[30] M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT*, pages 506–525, 2014.

[31] I. Damgard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[32] S. Faber, S. Jarecki, S. Kentros, and B. Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, pages 360–385, 2015.

[33] S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 360–378, 2018.

[34] P. Laud. A private lookup protocol with low online complexity for secure multiparty computation. In *ICICS*, pages 143–157, 2014.

[35] P. Laud. Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2015(2):188–205, 2015.

[36] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy*, pages 218–234, 2016.

[37] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient RAM-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638, 2014.

[38] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *ACM Conference on Computer and Communications Security (CCS)*, pages 215–226, 2014.

[39] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky. Efficient 3-party distributed ORAM. In *Security and Cryptography for Networks (SCN)*, pages 215–232, 2020.

[40] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Advances in Cryptology – EUROCRYPT*, pages 640–658, 2014.

[41] M. Keller and A. Yanai. Efficient maliciously secure multiparty computation for RAM. In *Advances in Cryptology – EUROCRYPT*, pages 91–124, 2018.

[42] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *ACM Conference on Computer and Communications Security (CCS)*, pages 850–861, 2015.

[43] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen. S^{3}ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir secret sharing. In *ACM Conference on Computer and Communications Security (CCS)*, pages 491–505, 2017.

[44] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, 2018.

[45] H. Chen, I. Chillotti, and L. Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *ACM Conference on Computer and Communications Security (CCS)*, pages 345–360, 2019.

[46] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi. Optorama: Optimal oblivious RAM. In *EUROCRYPT*, pages 403–432, 2020.

[47] D. Cash, A. Drucker, and A. Hoover. A lower bound for one-round oblivious RAM. In *TCC*, pages 457–485, 2020.

[48] T. Toft. Secure data structures based on multi-party computation. In *ACM Principles of Distributed Computing (PODC)*, pages 291–292, 2011.

[49] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[50] E. Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *IEEE Symposium on Security and Privacy (S&P)*, pages 449–465, 2020.

[51] Z. Jafargholi, K. Larsen, and M. Simkin. Optimal oblivious priority queues. In *ACM-SIAM Symposiym on Discrete Algorithms (SODA)*, pages 2366–2383, 2021.

[52] C. Rao, K. Singh, and A. Kumar. Oblivious stable sorting protocol and oblivious binary search protocol for secure multi-party computation. *Journal of High Speed Networks*, 27(1):67–82, 2021.

[53] R. M. Neal. Fast exact summation using small and large superaccumulators. *arXiv ePrint*, abs/1505.05571, 2015.

[54] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. A Reproducible Accurate Summation Algorithm for High-Performance Computing. In *Proceedings of the SIAM EX14 workshop*, 2014.

[55] J. Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[56] Y.-K. Zhu and W. B. Hayes. Correct rounding and a hybrid approach to exact floating-point summation. *SIAM Journal on Scientific Computing*, 31(4):2981–3001, 2009.

[57] Y.-K. Zhu and W. B. Hayes. Algorithm 908: Online Exact Summation of Floating-Point Streams. *TOMS*, pages 1–13, 2010.

[58] J. Demmel and Y. Hida. Accurate and efficient floating point summation. *SIAM Journal on Scientific Computing*, 25(4):1214–1248, 2004.

[59] J. Demmel and Y. Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37(1-4):101–112, 2004.

[60] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part i: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[61] D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th IEEE Symp. on Computer Arithmetic (ARITH)*, pages 132–143, Jun 1991.

[62] M. A. Malcolm. On accurate floating-point summation. *Commun. ACM*, 14(11):731–736, November 1971.

[63] H. Leuprecht and W. Oberaigner. Parallel algorithms for the rounding exact summation of floating point numbers. *Computing*, 28(2):89–104, 1982.

[64] E. Kadric, P. Gurniak, and A. DeHon. Accurate parallel floating-point accumulation. In *21st IEEE Symp. on Computer Arithmetic (ARITH)*, pages 153–162, April 2013.

[65] J. Demmel and Hong Diep Nguyen. Parallel reproducible summation. *IEEE TC*, 64(7):2060–2070, July 2015.

[66] O. Catrina and C. Dragulin. Multiparty computation of fixed-point multiplication and reciprocal. In *2009 20th International Workshop on Database and Expert Systems Application*, pages 107–111. IEEE.

[67] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *International Conference on Financial Cryptography and Data Security*, pages 35–50, 2010.

[68] O. Catrina and S. De Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *European Symposium on Research in Computer Security (ESORICS)*, pages 134–150, 2010.

[69] M. Franz and S. Katzenbeisser. Processing encrypted floating point signals. In *Proceedings of the 13th ACM multimedia workshop on Multimedia and security*, pages 103–108, 2011.

[70] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[71] K. Sasaki and K. Nuida. Efficiency and accuracy improvements of secure floating-point addition over secret sharing. In *International Workshop on Security*, pages 77–94, 2020.

[72] L. Kamm and J. Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.

[73] V. Dimitrov, L. Kerik, T. Krips, J. Randmets, and J. Willemson. Alternative implementations of secure real numbers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 553–564, 2016.

[74] O. Catrina. Optimizing secure floating-point arithmetic: Sums, dot products, and polynomials. In *Proceedings of the Romanian Academy*, volume 21, pages 21–28, 2020.

[75] O. Catrina. Performance analysis of secure floating-point sums and dot products. In *International Conference on Communications (COMM)*, pages 465–470. IEEE, 2020.

[76] Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *ACM CCS*, pages 813–826, 2013.

[77] F. Bayatbabolghani, M. Blanton, M. Aliasgari, and M. Goodrich. Secure fingerprint alignment and matching protocols. arXiv Report 1702.03379, 2017.

[78] GMP – The GNU multiple precision arithmetic library. `http://www.gmplib.org`.

[79] Floram implementation. `https://gitlab.com/neucrypt/floram/tree/floram-release`.

[80] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM symposium on Principles of Distributed Computing*, pages 101–111, 1998.

[81] O. Catrina. Round-efficient protocols for secure multiparty fixed-point arithmetic. In *International Conference on Communications (COMM)*, pages 431–436, 2018.

[82] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1102–1120, 2019.

[83] 3PC ORAM implementation. `https://github.com/Boyoung-/circuit-oram-3pc`.

[84] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, 2021.

[85] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD$\mathbb{Z}_{2}^{k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, pages 769–798, 2018.

[86] MP-SPDZ implementation. `https://github.com/data61/MP-SPDZ`.

[87] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, pages 5–48, March 1991.

[88] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Springer Science & Business Media, 2009.

[89] L.-K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani. Benchmarks and performance analysis of decimal floating-point applications. In *25th International Conference on Computer Design*, pages 164–170, 2007.

[90] M. Tommila. Apfloat for Java. `http://www.apfloat.org/apfloat_java/`. Accessed 2015-12-16.

[91] Exact Geometric Computation in LEDA. In *SoCG*, pages 418–419, 1995.

[92] G. Hanrot, V. Lefévre, P. Pélissier, P. Théveny, and P. Zimmermann. The GNU MPFR library. `http://www.mpfr.org/`. Accessed 2015-12-16.

[93] MPFR: A multiple-precision binary floating-point library with correct rounding. *TOMS*, June 2007.

[94] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Full-speed deterministic bit-accurate parallel floating-point summation on multi- and many-core architectures. *HAL-CCSD, Tech. Rep. hal-00949355*, 2014.

[95] M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *International Conference on Applied Cryptography and Network Security*, pages 377–397. Springer, 2020.