Combining Secure Multiparty Computation and Oblivious Data Structures

Thomas Skovlund Hansen, 201509447
thomasskovlundhansen@gmail.com

Master's Thesis, Computer Science
August 2023
Advisor: Peter Scholl

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In an increasingly technological world, keeping digital data private has become a major concern in many scenarios. One such scenario is *secure multiparty computation* (MPC) where multiple parties want to perform computation on their joint data in order to learn a common result while ensuring that their respective data remains private. For instance, several competing companies might want to pool together their data related to customers and sales in order to get more accurate results from computation on the larger joint data set. However, they only want to do so if they can be sure that the joint computation does not reveal any business-critical information which the other companies can use to gain a competitive advantage.

Thanks to the recent introduction of new efficient and actively secure MPC protocols as well as the introduction of publicly available frameworks for writing MPC programs in high-level programming languages and executing these programs using state-of-the-art MPC protocols, it has become feasible to apply MPC to solving increasingly complex real-world problems. Still, a major remaining challenge consists of improving the efficiency of MPC programs with input-dependent memory access; in order to keep the inputs private when jointly evaluating such programs, every memory access naïvely ends up incurring a linear overhead – compared to insecure computation of the same program in the RAM model – since all memory addresses must be touched to avoid revealing which specific address depends on the input. To reduce this overhead, we can utilize *oblivious data structures* that guarantee a physical memory access pattern that is independent of the addresses that are logically accessed.

In this thesis, we survey the current state of MPC and oblivious data structures. Furthermore, we formalize a framework for specifying oblivious data structures in MPC. Finally, we present a new specification, implementation, and experimental evaluation of the performance of an oblivious priority queue called *Path Oblivious Heap* (POH) [33] which builds on the ideas of tree-based oblivious RAM. In addition, we adapt POH to enforce uniqueness of inserted values and to support updating of a key given its associated value. This allows us to compare the adapted variant, which we call *Unique POH*, to a baseline implementation of a min-heap on top of oblivious RAM which supports the same set of operations. Our evaluation shows compelling and promising practical performance of POH and Unique POH, improving upon the state of the art and opening up the possibility of applications such as oblivious sorting and oblivious evaluation of graph algorithms in MPC.

Resumé

I en verden hvor teknologien bliver mere og mere avanceret, er digital datasikkerhed en væsentlig bekymring i mange scenarier. Ét sådant scenarie er *secure multiparty computation* (MPC), hvor flere parter ønsker at opnå et fælles resultat ved at regne på foreningsmængden af deres respektive datasæt *uden at afsløre denne data for de andre parter*. Eksempelvis kunne nogle konkurrerende virksomheder have et ønske om at regne på deres forenede kunde- og salgsdata for at opnå mere præcise resultater, end de kunne have opnået hver især. Dog vil den enkelte virksomhed kun deltage i en sådan beregning, hvis det kan garanteres, at ingen af de andre virksomheder kan opsnappe dens private data og dermed opnå en potentiel konkurrencefordel.

Takket være indførelsen af nye højtydende og aktivt sikre MPC-protokoller såvel som offentligt tilgængelige frameworks, der muliggør skrivning af MPC-programmer i højniveau-programmeringssprog samt eksekvering af disse programmer ved brug af de bedste MPC-protokoller, er det blevet muligt at anvende MPC til at løse praktiske problemer af tiltagende kompleks karakter. Én tilbageværende udfordring er dog at forbedre udførselstiden af MPC-programmer, der tilgår hukommelse afhængig af deres input; i sådanne programmer ender en hukommelsestilgang naivt med at have en lineær udførselstid, hvorimod den samme tilgang i den usikre RAM-model har en konstant udførselstid. Årsagen til dette er, at man som udgangspunkt rører ved alle hukommelsesceller i MPC-programudførsel for at skjule den specifikke celle, der tilgås. For at reducere dette overhead kan vi benytte "oblivious datastrukturer", som garanterer et fysisk hukommelsesadgangsmønster, der er uafhængigt af det logiske.

I dette speciale kortlægger vi den nuværende tilstand af MPC og oblivious datastrukturer. Endvidere formaliserer vi et framework til at specificere oblivious datastrukturer i MPC. Endelig præsenterer vi en specifikation, implementation samt eksperimentel evaluering af ydeevnen af en oblivious prioritetskø kaldet *Path Oblivious Heap* (POH) [33], som er inspireret af træbaseret oblivious RAM. Ydermere tilpasser vi POH til at garantere, at indsatte værdier er unikke samt til at tillade opdatering af en nøgle givet den tilhørende værdi. Denne tilpassede variant, som vi kalder *Unique POH*, sammenligner vi med en grundlæggende implementation af en min-hob, der er bygget på oblivious RAM og understøtter de samme operationer. Ifølge vores evaluering udviser POH og Unique POH en overbevisende og lovende praktisk ydeevene, der er sammenlignelig med de bedste implementationer. Dette åbner op for anvendelser såsom oblivious sortering og eksekvering af oblivious grafalgoritmer i MPC.

Acknowledgments

First and foremost, I would like to extend my sincere thanks to my advisor Peter Scholl for his indispensable guidance throughout my work on this thesis project – and for introducing me to oblivious data structures in the first place. In addition, Mathias Rav has been immensely helpful every time I felt the need to discuss theoretical as well as practical details. I would like to thank him for listening, for asking the right questions, and for providing valuable feedback on my writing which has improved the readability of the final product considerably. Jakob Lysgaard Rørsted also deserves thanks for his advice on typesetting my thesis in LaTeX.

I wish to give my special thanks to Emilie for her love and support throughout my work on this thesis, for keeping my spirits high, and for cheering me up and helping me through the most difficult periods.

This thesis is the conclusion and culmination of my time as a computer science student at Aarhus University. It has been eight years of mostly studying, but at the same time I have been lucky enough to be able to lead a rich and fun social life, without which my time at the university would have probably been shorter – and definitely not as happy and memorable. Therefore, I find it fitting to thank all my friends at TÅGEKAMMERET.

Thomas Skovlund Hansen,
Aarhus, August 2023.

Contents

Chapter 1

Introduction

In today's highly technological world, a vast amount of digital data is constantly being transferred over the Internet, stored, and processed by many parties. The invention of the computer, and subsequently the computer-connecting Internet, has indisputably had a major global impact and is one of the great technological breakthroughs, if not the greatest breakthrough, of the 20th century. Today, it is feasible to transfer a TCP data packet around the globe in less than a second, enabling geographically separate parties to perform near-instant computation on their shared data. Furthermore, a party has the opportunity to outsource the task of storing or processing their data to more computationally capable parties in "the cloud". With this breakthrough, however, many new challenges arise. One particular challenge is the following: Digital data might be private and sensitive. For instance, it could be business-critical data belonging to a company, personal health data, or intelligence belonging to an intelligence agency. If a party wants to store and access such data at an untrusted location, or if they want to perform a joint computation on their and other parties' data, it is imperative that they can control – or at least know in advance – how much private information is revealed in the expected and worst case.

Keeping data private in scenarios such as the ones we have just described is one of the major concerns of the scientific field of *cryptography*. Specifically, joint computation on private data is called *secure multiparty computation* (MPC) in scientific literature. An example of a problem that can be solved using MPC is the following: several competing companies might want to pool together their data related to customers and sales in order to get more accurate results from computation on the larger joint data set. However, they only want to do so if they can be sure that the joint computation does not reveal any business-critical information that the other companies can use to gain a competitive advantage.

The problem of storing private data in an untrusted location is often solved by *encrypting* the data. However, when a program takes input from a user and reads certain pieces of the encrypted data based on the user input, the *data access pattern*, that is, the sequence of memory locations accessed, can reveal sensitive information about the user or about which pieces of data are more interesting than others. Hiding

the data access pattern in such a scenario is the objective of so-called *oblivious data structures*.

In recent years, the introduction of new MPC protocols, such as the *SPDZ* protocol by Damgård, Pastro, Smart, and Zakarias [7] (the name of the protocol is an acronym after the authors' last names and is pronounced "SPeeDZ"), has had the effect of making MPC feasible in many new practical scenarios. The quest towards making secure multiparty computation of general functions feasible is an active research area, and *generic MPC* frameworks that implement various protocols, for instance *Multi-Protocol SPDZ* (MP-SPDZ) by Keller [22], have been made publicly available during the last decade or so, allowing developers to write arbitrary MPC programs in high-level programming languages. This is a major step forward; as we have seen in regular, insecure (as opposed to MPC) software development, introducing high-level programming abstractions has the effect of speeding up development in general, and furthermore, it becomes easier to maintain an overview when writing increasingly complex programs. A challenge that still remains, however, is to improve the practical performance of arbitrary MPC programs written in high-level programming languages. Most programs access memory addresses that depend on the program input, which leads to a data access pattern that reveals sensitive information. A program that needs to access a single input-dependent memory address can avoid revealing sensitive information in a naïve way by touching all memory addresses, incurring a linear overhead in the size of the memory. By making use of oblivious data structures, we can reduce this asymptotic overhead significantly. Generic MPC protocols primarily work by translating a program into a Boolean or arithmetic *circuit* and evaluating that circuit. Since an input in MPC is supposed to remain private to one party, accessing a single input-dependent memory address translates into a circuit that uses all its inputs, which again leads to the aforementioned linear overhead. Interestingly, it turns out that oblivious data structures can also help us solve this problem in an MPC context, resulting in significantly less asymptotic overhead compared to the naïve approach we have just described. Still, implementing an oblivious data structure efficiently in MPC is not always a simple task; oftentimes, complex implementations of asymptotically efficient oblivious data structures in MPC turn out to suffer from large constant overhead, resulting in circuits with too many gates to be practical. Today, a substantial amount of research in the area [25, 26, 33, 37, 38, 42] is devoted to designing asymptotically efficient oblivious data structures with small constant overhead in MPC.

1.1 Problem Statement and Outline of Thesis

In this thesis, our high-level goal is to contribute to the existing volume of research that aims at improving the performance and availability of practical, generic MPC. We aim to achieve this by implementing and experimentally evaluating the performance of *Path Oblivious Heap*

(POH) – an oblivious priority queue introduced by Shi [33] that promises near-optimal performance in terms of the best known lower bounds [19]. We are going to implement and evaluate the performance of POH in the SPDZ MPC protocol [7] which efficiently generalizes to an arbitrary number of parties and is *actively secure* against a *dishonest majority* of participating parties (secure even in the presence of a majority of parties that try to cheat by deviating from the protocol and by cooperating).

After introducing basic notation and definitions in Chapter 2, we begin our journey by surveying the current state of generic MPC and oblivious data structures. First, in Chapter 3, we describe MPC in general and two major MPC protocols that are being used today in particular, namely *Yao's Garbled Circuits* [40] and *SPDZ* [7]. Next, in Chapter 4, we introduce oblivious data structures, and in particular, we present several approaches to constructing *oblivious RAM* (ORAM) as well as the POH oblivious priority queue [33]. Following this survey, in Chapter 5, we develop a framework for formalizing oblivious data structures via pseudocode in a generic MPC context, and we give an example of using the framework by formalizing and analyzing a variant by Zahur et al. [42] of the *square-root* ORAM scheme by Goldreich and Ostrovsky [14] which we have introduced in its original form in Chapter 4. Finally, in Chapter 6, we present an implementation and analysis of POH in the state-of-the-art MP-SPDZ framework [22], and we experimentally evaluate its performance in the SPDZ protocol compared to the performance of a priority queue by Keller and Scholl [25] which is a classic min-heap built on top of ORAM.

1.2 Technical Highlights

The major technical highlights that we present in this thesis are the following:

- We formalize a framework outlined by Zahur et al. [42] for specifying oblivious data structures, and programs in general, in a generic, reactive MPC setting. The framework is based on a so-called *arithmetic black box* functionality specified using the *universal composability* framework by Canetti [6].
- We present a detailed analysis of the efficiency of the adapted square-root ORAM scheme, also introduced by Zahur et al. [42].
- Finally, we implement POH and compare it to an ORAM min-heap [25] in the SPDZ protocol. In order to make POH comparable to the ORAM min-heap, we adapt it, using ideas by Keller and Scholl [25], to enforce uniqueness of inserted values and to support updating of a key given its associated value, at the cost of $\Theta(\log n)$ additional overhead. We call this adapted version Unique POH. To the best of our knowledge, no existing, practical oblivious priority queue implements the same functionality as asymptotically efficient in the worst case as Unique POH. Furthermore, we are the first to evaluate the performance of POH in the SPDZ protocol; Shi [33] only presents a brief

evaluation of concrete MPC performance in the semi-honest garbled circuits setting, whereas we give a more thorough evaluation in the dishonest setting. Our evaluation shows compelling and promising practical performance of POH and Unique POH, improving upon the state of the art and opening up the possibility of applications such as oblivious sorting and oblivious evaluation of graph algorithms in MPC.

1.3 Related Work

ORAM in MPC The idea of combining ORAM ond MPC was first formulated by Gordon et al. [16] in 2012 with an *ORAM-in-MPC* (ORAM-MPC) implementation based on the *tree ORAM scheme* by Shi et al. [34]. In this paper, ideas from an earlier work on *Private Information Storage* (hiding information through the use of multiple databases that do not interact with one another) by Ostrovsky and Shoup [31] are used in the MPC setting, resulting in an *asymmetric* ORAM-MPC approach where the so-called *server state* of an ORAM scheme is maintained in the clear by one MPC party while the *client state* is *secret shared* between all parties such that no party knows the client state. Subsequent works propose improved protocols, based on tree ORAM [9, 12, 25, 37, 38, 39] and square-root ORAM [42], all aimed at being practically efficient in MPC. These works are not limited to the asymmetric client-server scenario we have just described, but instead apply to general MPC where the entire state is secret shared between all parties. The square-root ORAM scheme is not as asymptotically efficient as the tree ORAM protocols – however, it serves as an example of how focusing on reducing constant overhead can still result in compelling practical performance for a range of parameters.

Oblivious priority queues in MPC In concurrent works, Keller and Scholl [25] and Wang et al. [38] propose novel, specialized oblivious data structures that asymptotically outperform generic oblivious data structures implemented on top of ORAM. Specifically, both papers present an oblivious priority queue based on a *non-recursive* tree ORAM structure. The priority queue of Wang et al. is presented in the classical ORAM model and does not directly translate to the MPC setting, while the priority queue of Keller and Scholl is not currently implemented nor evaluated (however, they do implement a simpler oblivious min-heap priority queue on top of tree ORAM that is a factor $\Theta(\log n)$ less efficient).

Recently, as we have already mentioned, Shi [33] proposed POH which works in the classical ORAM model as well as in generic MPC and asymptotically outperforms the priority queues of Keller and Scholl and of Wang et al. Shi evaluates the performance of POH in the semi-honest garbled circuits setting and compares it to the priority queue of Wang et al. as well as a priority queue proposed by Jafargholi, Larsen, and Simkin [21] which roughly matches the asymptotic efficiency of POH. Shi concludes that POH is the most practically efficient solution and that the priority queue of Jafargholi, Larsen, and Simkin is in fact less efficient in practice than the priority queue of Wang et al.

Oblivious algorithms in MPC Specialized oblivious data structures have the prospect of being important building blocks when implementing various oblivious algorithms in MPC. Keller and Scholl [25] and Wang et al. [38] apply their data structures to solving various problems, notably they evaluate the performance of finding the shortest path between nodes in general graphs [25] and planar graphs [38]. The priority queue of Wang et al. does not support decreasing the key associated with a value in the queue, which is needed by the general Dijkstra's algorithm [8]. Instead, they present a specialized algorithm that works for planar graphs. Keller and Scholl implement an oblivious version of the general Dijkstra's algorithm based on their min-heap priority queue implementation. They compare their implemented algorithm to the previously best known solution [2] which is not based on oblivious data structures. Even though the solution of Keller and Scholl is asymptotically more efficient, their benchmarks indicate that there is a significant overhead which is higher than the asymptotic advantage for all graph sizes that they could run their algorithm on.

Finally, *oblivious sorting* is another interesting application of oblivious priority queues. Shi [33] notes that POH immediately implies an efficient oblivious sorting algorithm which has currently not been implemented nor evaluated. In theory, this algorithm should be able to outperform *Oblivious Radix Sort* [17], which is currently the state-of-the-art implementation in MP-SPDZ [22] – especially for a large number m of MPC parties, since the complexity of Oblivious Radix Sort is exponential in m.

Chapter 2

Preliminaries

This chapter introduces notation, basic definitions, and concepts that the reader should be familiar with before proceeding to read the remaining part of the thesis.

2.1 Notation

Notation 2.1 (Hidden Values and Oblivious Statements)**.** We generally write $\langle x \rangle$ when the value x is *hidden*, for instance if it is *secret shared* (see Definition 3.11 on page 21). What exactly it means for x to be hidden will be clear from the context.

Similarly, in a pseudocode context, we write $\langle \mathbf{s} \rangle$ when the statement **s** is *oblivious*, meaning that it should not be revealed whether **s** is executed or not. For instance, in an $\langle \mathbf{if} \rangle$ statement, the body is always executed, regardless of whether the guard is true or false, but in the false case, the body of the statement becomes a dummy operation where the access pattern is *indistinguishable* (see Definition 2.14 on page 9) from the actual statement body. We elaborate on this last part when we use the notation later. △

Notation 2.2 (Computational Domains)**.** In a multiparty computation context, we usually write $\mathbb{F}$ for the computational domain and only specify it more accurately if it helps to convey the topic at hand. Usually, it suffices to think of $\mathbb{F}$ as a finite field of integers. To increase readability, we omit explicit modulo notation for integer arithmetic in a finite field $\mathbb{F}$. △

Example 2.3 (Bits as a Field)**.** One important example of a finite field is the set $\{0, 1\}$ equipped with binary XOR as addition and binary AND as multiplication. ▲

Notation 2.4 (UC Functionalities)**.** We present several *ideal functionalities* using the *universal composability* (UC) framework by Canetti [6]. However, we use the UC framework quite informally; for instance, we often do not explicitly require message *id*s, and we omit mentioning *influence* and *leakage ports* altogether. It should be understood that all presented functionalities can be formalized in this sense, but for the purpose of giving intuition, we find that a simplified presentation does a better job. △

Notation 2.5 (Natural Numbers)**.** Let $\mathbb{N}_0 = \{0, 1, \ldots\}$ and let $\mathbb{N} = \{1, 2, \ldots\}$. Furthermore, for $n \in \mathbb{N}$, let $[n]_0 = \{0, 1, \ldots, n-1\}$, let $[n] = \{1, \ldots, n\}$, and let $\mathbb{N}_n = \{n, n+1, \ldots\}$. △

Example 2.6. $\mathbb{N}_2 = \{2, 3, \ldots\}$ is an important example of Notation 2.5 for the multiparty computation setting. ▲

Notation 2.7 (Logarithms)**.** When we write log without an explicit base, it is shorthand for $\log_2$. △

Notation 2.8 (Concatenation)**.** For two strings x and y, $x \parallel y$ denotes concatenation of x and y. △

Notation 2.9 (Uniformly Random Sampling)**.** When we want to say that an element x is sampled uniformly at random from a set S, we write $x \in_R S$. △

2.2 Definitions

Definition 2.10 (Negligibility)**.** We say that a function $\mu(\cdot) : \mathbb{N} \to \mathbb{R}$ is *negligible* if for every positive polynomial $p(\cdot)$ there exists a non-negative constant $c_p \in \mathbb{N}_0$ such that for all $x > c_p$,

$$|\mu(x)| < \frac{1}{p(x)}.$$

△

Definition 2.11 (Statistical Distance)**.** Let P and Q be discrete probability distributions over the same set of events $\mathcal{E}$. We define the *statistical distance* between P and Q to be

$$\mathrm{SD}(P, Q) = \frac{1}{2} \sum_{e \in \mathcal{E}} |P(e) - Q(e)| .$$

Equivalently, for discrete random variables X and Y over sample space $\mathcal{S}$, we define the statistical distance between X and Y to be

$$\mathrm{SD}(X, Y) = \frac{1}{2} \sum_{s \in \mathcal{S}} |\Pr[X = s] - \Pr[Y = s]| .$$

△

Definition 2.12 (Families of Probability Distributions)**.** A *family of probability distributions* defined over a set of events $\mathcal{E}$ and indexed by a set $\mathcal{X}$ is a function $\mathcal{P}$ that maps elements from $\mathcal{X}$ to probability distributions over $\mathcal{E}$. For $x \in \mathcal{X}$, we use the notation $\mathcal{P}_x$ to denote the probability distribution $\mathcal{P}(x)$. △

Remark 2.13. One can think of $\mathcal{P}$ in the above definition as a probabilistic algorithm indexed by the (possibly infinite, but enumerable) set of all possible inputs $\mathcal{X}$. ▲

Definition 2.14 (Indistinguishability)**.** Let $\mathcal{P}$ and $\mathcal{Q}$ be two families of probability distributions over $\mathcal{E}$ indexed by the same set of bit strings $\mathcal{X}$. In addition, define the function $\mu\colon \mathbb{N}_0 \to \mathbb{R}$ by

$$\mu(\lambda) = \max_{x \in \mathcal{X}\colon |x| = \lambda} \mathrm{SD}(\mathcal{P}_x, \mathcal{Q}_x).$$

We say that $\mathcal{P}$ and $\mathcal{Q}$ are *perfectly indistinguishable*, written $\mathcal{P} \overset{\text{perf}}{\cong} \mathcal{Q}$, if $\mu(\lambda) = 0$ for all $\lambda \in \mathbb{N}_0$. Note that an equivalent requirement is that for all $x \in \mathcal{X}$ and for all $e \in \mathcal{E}$, $\mathcal{P}_x(e) = \mathcal{Q}_x(e)$. Furthermore, we say that $\mathcal{P}$ and $\mathcal{Q}$ are *statistically indistinguishable*, written $\mathcal{P} \overset{\text{stat}}{\cong} \mathcal{Q}$, if μ is only negligible as opposed to being zero everywhere. Finally, we say that $\mathcal{P}$ and $\mathcal{Q}$ are *computationally indistinguishable*, written $\mathcal{P} \overset{\text{comp}}{\cong} \mathcal{Q}$, if it holds for every $x \in \mathcal{X}$ that no probabilistic, polynomial-time (PPT) algorithm D is able to successfully distinguish between $\mathcal{P}_x$ and $\mathcal{Q}_x$ when given a sample from each distribution as input, except with probability negligible in $|x|$. We call this probability the *distinguishing advantage* of D. △

Observation 2.15. It is straightforward to check that

$$\mathcal{P} \overset{\text{perf}}{\cong} \mathcal{Q} \implies \mathcal{P} \overset{\text{stat}}{\cong} \mathcal{Q}$$

and

$$\mathcal{P} \overset{\text{stat}}{\cong} \mathcal{Q} \implies \mathcal{P} \overset{\text{comp}}{\cong} \mathcal{Q}.$$

▲

Definition 2.16 (Circuits)**.** A circuit over a finite field $\mathbb{F}$ on n inputs and m outputs is a tuple $C = (G, W, T, \ell, O)$ where

- G is a set of *gates*,
- $W \subseteq G \times G$ is a totally ordered[1] set of *wires* connecting the gates,
- T is a set of *gate operation types*, where each type must specify a *fan-in*,
- $\ell\colon G \to T$ is a *labeling function*, and
- $O = (o_i)_{i=1}^m$ where $o_i \in G$ is a choice of m (not necessarily distinct) *output gates*.

We require T to contain n *input labels* $x_1, \ldots, x_n$, all of fan-in 0. Furthermore, we require that (G, W) constitute a *directed acyclic graph*. Finally, we require the fan-in of every gate $g \in G$, according to W, to adhere to that specified by the operation type $\ell(g)$. We say that C computes a function $f\colon \mathbb{F}^n \to \mathbb{F}^m$ in a natural way: evaluation starts by assigning the inputs to the chosen input gates and then follows a topological

[1] In order for our circuit definition to support non-commutative gate operations, we must as a very minimum require all wires that are input to the same gate to have an internal ordering. In Definition 2.16, we simplify matters by requiring that the wires be totally ordered.

order of the gates, computing the operation of each gate on the gate inputs and outputting the result of the computation on the output wire of the gate, until all output gates have been evaluated and the outputs of the circuit can be obtained.

Usually, $T = \{x_1, \ldots, x_n, \oplus, \otimes\} \cup \mathbb{F}$ where $\oplus$ denotes *addition with fan-in 2*, $\otimes$ denotes *multiplication with fan-in 2*, and $\mathbb{F}$ denotes *constant gates with fan-in 0*. We call such a circuit over a finite field an *arithmetic circuit*. In the special case $\mathbb{F} = \{0, 1\}$, where $\oplus$ denotes binary XOR and $\otimes$ denotes binary AND, we call the circuit a *Boolean circuit*. △

Definition 2.17 (Message Authentication Codes)**.** A *message authentication code* (MAC) scheme is a triple of algorithms $\mathcal{M} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Verify})$ satisfying the following:

Gen is a randomized *key generation algorithm* that upon input of a *security parameter* λ outputs a key $\Delta \leftarrow \mathsf{Gen}(1^\lambda)$. Usually, Δ is sampled uniformly at random from a key space determined by λ.

Auth is an *authentication algorithm* that upon input of the key Δ and a message m produces a MAC $a \leftarrow \mathsf{Auth}_\Delta(m)$.

Verify is a *verification algorithm* that is able to verify a MAC a against a message m given the key Δ. It always outputs $v \leftarrow \mathsf{Verify}_\Delta(a, m)$ where $v \in \{\mathsf{accept}, \mathsf{reject}\}$.

We always require that $\mathcal{M}$ is correct in the sense that $\mathsf{Verify}_\Delta(\mathsf{Auth}_\Delta(m), m) = \mathsf{accept}$ for all m and Δ. We also require that $\mathcal{M}$ is secure against *chosen message attacks* (CMAs): We define a game where an adversary $\mathcal{A}$ is given access to an oracle $\mathcal{O}$ who initially stores $\Delta \leftarrow \mathsf{Gen}(1^\lambda)$. Now, $\mathcal{A}$ can ask $\mathcal{O}$ to compute as many MACs on as many messages chosen by $\mathcal{A}$ as $\mathcal{A}$ wants and to share these MACs with $\mathcal{A}$. Finally, $\mathcal{A}$ wins the game if $\mathcal{A}$ is able to output a valid pair a_0, m_0 of MAC and message, where $\mathcal{O}$ has not computed a MAC on m_0, such that $\mathsf{Verify}_\Delta(a_0, m_0) = \mathsf{accept}$. We say that $\mathcal{M}$ is information-theoretically CMA-secure (or simply *information theoretic*) if for any adversary $\mathcal{A}$, $\mathcal{A}$ cannot do any better in the above game than guessing uniformly at random. △

Chapter 3

Secure Multiparty Computation

Secure multiparty computation (MPC) is the process of multiple parties jointly executing an interactive protocol in order to compute a function of their respective private inputs. The goal is that some agreed-upon, non-empty subset of the parties learn the output of the computed function – and nothing more. Specifically, "secure" means that all parties' private inputs should remain private during and after the execution of an MPC protocol – even in the presence of *corrupt parties* taking part in the protocol execution. Since it is essentially always a required property, "secure" is often omitted when mentioning and discussing the topic. Another intuitive view of MPC is to say that, essentially, the goal is to replace a *trusted third-party* who learns all parties' inputs, computes the desired function, and reveals the output to the parties, with a peer-to-peer protocol where no one knows the complete truth.

Example 3.1 (Electronic Voting)**.** As an example of how MPC is relevant in practice, consider the canonical case of *electronic voting*: Parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ ($n > 1$) want to vote yes or no on some decision, and they would all like to learn what the majority has voted, while no party wants to reveal their own vote. In this case, the function to be computed can be formulated as

$$f(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i > \frac{n}{2}, \text{ and} \\ 0 & \text{otherwise,} \end{cases}$$

where $x_i \in \{0, 1\}$ for $i \in [n]$ and 1 means "yes" while 0 means "no". This is clearly an instance of MPC and arguably a very basic and relevant problem in many real-world scenarios. Of course, the *majority vote* function is quite a simple example (which is indeed why it is chosen to illustrate the basic MPC concept), but it is not hard to imagine significantly more complex functions that are interesting to compute in an MPC setting. ▲

Remark 3.2. Note that in Example 3.1, the *domain* is n bits and the *range* is one bit. In general, we can define MPC over any domain and range, e.g. finite fields of integers. The domain and range are important factors when deciding what MPC protocol is best suited to solve a particular instance of MPC. ▲

3.1 Security

Before discussing MPC protocols in more detail, we define two notions of security that an MPC protocol can satisfy. Our definitions are less formal versions of the ones by Hazay and Lindell [18]. First, we define the weaker notion of *passive security*:

Definition 3.3 (Passively Corrupt Parties)**.** A *passively corrupt* (or semi-honest) party taking part in the execution of an MPC protocol is assumed to follow the protocol specification but may attempt to pool information from their view of the protocol execution together with information from other corrupt parties' views in order to learn more than they would be able to by themselves. △

Definition 3.4 (Passive Security)**.** We say that an MPC protocol is secure in the presence of passively corrupt parties – or simply *passively secure* – if there exists a PPT *simulator* that for any party is able to generate a view of the protocol execution that is *indistinguishable* from the view of that party given only that party's input and the output. △

Intuitively, by requiring that the view of a party be simulatable given only the input and output, Definition 3.4 ensures that the view does not contain more information than the input and output. Defining the stronger notion of *active security* requires a somewhat greater effort:

Definition 3.5 (Actively Corrupt Parties)**.** An *actively corrupt* (or malicious) party taking part in the execution of an MPC protocol may deviate from the protocol specification in any way they want, and, like passively corrupt parties, they may also cooperate with other parties in order to learn more information than they are supposed to. △

Definition 3.6 (Active Security)**.** We say that an MPC protocol Π is *actively secure* if it is secure in the presence of actively corrupt parties in the following way: We first define execution in an *ideal world* where all parties send their inputs to a *trusted third party* that computes the desired function and returns to each party their respective output. We assume that all corrupt parties are controlled by a PPT adversary $\mathcal{A}$. $\mathcal{A}$ may also at any time instruct the trusted third party to abort the computation – in that case, the trusted third-party returns abort to all parties. Next, we define execution in the *real world* where Π is executed. In this case, the adversary $\mathcal{A}$ communicates on behalf of the corrupt parties and may follow any strategy while the honest parties follow Π. Finally, we say that Π enjoys active security if for every PPT adversary $\mathcal{A}$ in the real world, there exists a PPT simulator $\mathcal{S}$ playing the role of the adversary in the ideal world such that the views of ideal world executions with $\mathcal{S}$ are indisinguishable from the views of real world executions with $\mathcal{A}$. △

By requiring that such a simulator exist, we ensure that $\mathcal{A}$ in the real world cannot do more than what is allowed in the ideal world. Since malicious adversaries may deviate arbitrarily from a protocol, for instance by choosing not to use their inputs at

all, it is not sufficient to require that their views are simulatable given their inputs and outputs – hence the more involved security definition. In turn, the security guarantee is stronger and is desirable in many real-world scenarios where it is not realistic to assume that all parties follow the protocol.

3.2 Generic MPC

Many approaches to MPC exist. Some protocols are specialized and optimized for specific types of computation such as private set intersection[1]. Other protocols are *generic* in the sense that they allow for the computation of arbitrary functions, at the cost of being less efficient than the specialized counterparts due to not being able to exploit the specific structure of the function being computed. These generic MPC protocols have seen a recent rise in popularity due to their versatility and the fact that recent results have indicated that generic MPC is not only of theoretical interest but also something we can hope to use in practice [29, 30]. Some important ways in which the various generic MPC protocols differ are the following:

Degree of security MPC protocols can be actively or passively secure. In between these two notions of security, some protocols are secure in the presence of *covert parties* who are allowed to deviate from the protocol, but only in order to maximize some pre-determined *utility function*.

Furthermore, when corruption is detected, we distinguish between protocols with *guaranteed output delivery*, protocols with *identifiable abort* where the honest parties are able to identify a corrupt party (allowing the honest parties to exclude the identified corrupt party from future participation), and protocols that only guarantee *abort*.

Fraction of corrupt parties tolerated In particular, we distinguish between protocols that require an *honest majority* of parties, and protocols that allow a *dishonest majority* of corrupt parties.

Model of computation Some MPC protocols rely on *preprocessed data* that can be prepared before knowing the function to be computed or the input. Other protocols are *online-only*.

Computational domain Usually, MPC is evaluated as a circuit over some *finite field*. An important special case is *Boolean circuits*. We distinguish between MPC protocols working only on Boolean circuits and MPC protocols working on *arithmetic circuits* over small or large finite fields. Some protocols also allow *mixed computation* where different domains are used for different parts of the computation.

[1] In private set intersection, two parties P_1 and P_2 each have a private set S_1 and S_2, and they want to learn $S_1 \cup S_2$ without revealing anything else; for example, imagine an intelligence service (with a list of suspected terrorists) and an airline (with a list of flight passengers) wanting to work together, without revealing anything to the other party except the intersection of those two lists.

Asymptotic efficiency Important measures are *communication round complexity*, *communication bandwidth*, *computational overhead* relative to insecure computation, and *space consumption*.

Practical efficiency Since we use MPC in practice, we are interested in protocols that have small constants and perform well in certain parameter ranges, even if they are asymptotically worse than other alternatives.

Generalizability Many protocols are specialized or optimized for a specific number n of parties – often two or three – and do not naturally generalize. When we want to emphasize this property, we refer to these as nPC protocols.

3.3 Oblivious Data Access in MPC

As we have already noted, many generic MPC protocols represent the function to be computed as a Boolean or arithmetic circuit. This is an advantage in many cases, since circuits are a well-studied and expressive model of computation. However, for particular types of functions, it turns out that their circuit complexity grows too rapidly as a function of the input size in order for the function to be efficiently computable in MPC for a wide range of realistic inputs. Functions that rely on input-dependent memory access is one important case where the circuit size blows up. This happens due to the fact that in any circuit representation, performing a full linear scan of a data structure is the naïve way of accessing a single element *obliviously*, i.e. without revealing which element is accessed, even if the computation in the RAM model is sublinear, as pointed out in [12, 16, 25, 39, 42]. The following example illustrates this problem:

Example 3.7 (Binary Search on a Sorted, Secret Database)**.** Imagine two parties that want to carry out secure computation on the union of their private data. One way of implementing this is to store both parties' data in a sorted database column in MPC. Let n be the total number of database entries. If the parties want to look up an entry in the column, a natural way to do so is via binary search. In the RAM model of computation, the number of memory accesses of binary search is $O(\log n)$. However, since the lookup is performed in MPC, the circuit touches every entry in the database to hide which entry is accessed, so the number of memory accesses is exactly n. Refer to Figure 3.1 on the facing page for an illustration. ▲

Since there is a lot of work dedicated to making generic MPC widely accessible, the fact that many functions are still not feasible to compute in a generic MPC setting defines an interesting and important research area. One approach to improving the performance of generic MPC protocols evaluating functions with input-dependent memory access is to combine existing MPC protocols with *oblivious data structures* and use so-called *reactive computation*, which can make the resulting online secure computation complexity proportional to the sublinear RAM model computation complexity instead of the MPC circuit size. We will introduce oblivious data

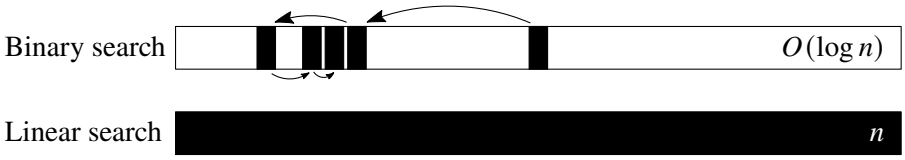


Figure 3.1: The memory access pattern for binary search vs. linear search. In binary search, the number of memory accesses is $O(\log n)$. In linear search, the number of memory accesses is exactly n.

structures in Chapter 4 and the approach of combining oblivious data structures and MPC in Chapter 5, but before we do that, let us describe some concrete MPC protocols that we will be working with in this thesis and define the *arithmetic black box model*, which allows us to work with certain types of MPC protocols in an abstract, reactive manner suitable for combining them with oblivious data structures.

3.4 Garbled Circuits

Yao [40] was the first to lay the foundations for MPC by introducing the 2PC protocol known as *Yao's Garbled Circuits* (GC) in 1986. In this protocol, one party acts as the *garbler* and another party acts as the *evaluator* in order to evaluate a Boolean circuit. The protocol builds on a *garbling scheme* which we now present based on the more recent definition by Bellare, Hoang, and Rogaway [5] from 2012.

3.4.1 Garbling Schemes

A garbling scheme enables evaluation of encrypted functions on encrypted inputs, resulting in an encrypted output that can be decrypted given some secret information. More formally:

Definition 3.8 (Garbling Scheme)**.** A *garbling scheme* is specified by a tuple $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ where

Gb is a randomized *garbling algorithm* that upon input of a security parameter $\lambda \in \mathbb{N}$ and the description $f \in \{0,1\}^*$ of a circuit outputs a triple of strings $(\langle f \rangle, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f)$ where $\langle f \rangle$ is a description of the *garbled circuit*, e is the *encoding information*, and d is the *decoding information*,

En is a deterministic *encoding algorithm* that uses encoding information e to map an input x to a *garbled input* $\langle x \rangle \leftarrow \mathsf{En}(e, x)$,

Ev is a deterministic *garbled evaluation* algorithm that evaluates a garbled circuit described by $\langle f \rangle$ on a garbled input $\langle x \rangle$ and produces a garbled output $\langle z' \rangle \leftarrow \mathsf{Ev}(\langle f \rangle, \langle x \rangle)$,

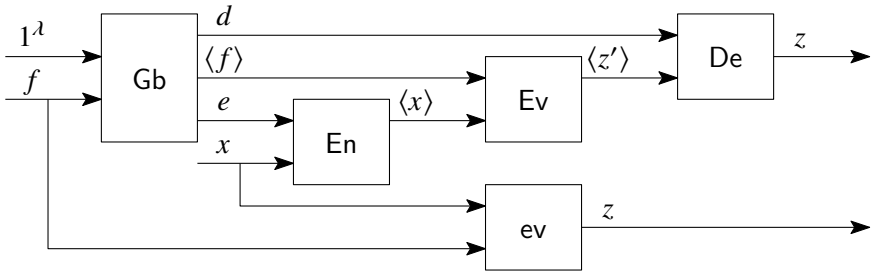


Figure 3.2: The data flow for resp. garbled circuit evaluation and regular non-garbled circuit evaluation. The outputs in both cases must be equal. This figure is heavily inspired by Bellare, Hoang, and Rogaway [5, Fig. 1].

De is a deterministic *decoding algorithm* that uses decoding information d to map a garbled output $\langle z' \rangle$ to a plaintext output $z \leftarrow \mathsf{De}(d, \langle z' \rangle)$, possibly with a small error probability, and

ev is a deterministic *plaintext evaluation algorithm* that evaluates a circuit described by f on an input x and produces an output $y \leftarrow \mathsf{ev}(f, x)$.

△

We always require a garbling scheme to be *correct*:

Definition 3.9 (Correctness of Garbling Scheme)**.** We say that a garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ enjoys *correctness* if for all $f \in \{0,1\}^*$ where $\mathsf{ev}(f, \cdot) \colon \mathbb{F}^n \to \mathbb{F}^m$ and for all $x \in \mathbb{F}^n$, the following probability

$$\Pr\left[\mathsf{ev}(f,x) \neq \mathsf{De}(d, \mathsf{Ev}(\langle f \rangle, \mathsf{En}(e,x))) : (\langle f \rangle, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f)\right]$$

is negligible in λ. △

See Fig. 3.2 for a diagram that illustrates the correctness requirement. Another required property is *circuit privacy*:

Definition 3.10 (Circuit Privacy of Garbling Scheme)**.** Let $\mathsf{struct}(f)$ denote some partial structure of the circuit described by f. We say that a garbling scheme $\mathcal{G}$ enjoys *circuit privacy* if there exists a PPT *simulator* S such that for all $f \in \{0,1\}^*$ where $\mathsf{ev}(f, \cdot) \colon \mathbb{F}^n \to \mathbb{F}^m$, and for all $x \in \mathbb{F}^n$,

$$(\langle f' \rangle, \langle x' \rangle, d') \leftarrow \mathsf{S}(1^\lambda, \mathsf{struct}(f), \mathsf{ev}(f,x))$$

is indistinguishable from

$$(\langle f \rangle, \langle x \rangle, d) \quad \text{where} \quad (\langle f \rangle, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f) \text{ and } \langle x \rangle \leftarrow \mathsf{En}(e,x)$$

except with probability negligible in λ. △

Definition 3.10 intuitively says that a garbled circuit, a garbled input, and a decoding string may only leak information about the plaintext output $\mathsf{ev}(f, x)$ and some partial information $\mathsf{struct}(f)$ about the structure of the circuit described by f. We ensure that this is the case by requiring that there exists an algorithm S (PPT in λ) that is able to *simulate* a garbled circuit, a garbled input, and a decoding string which has a joint distribution that is computationally close to that of $(\langle f \rangle, \langle x \rangle, d)$, computed from $\mathsf{Gb}(1^\lambda, f)$ and $\mathsf{En}(e, x)$, even though S only knows $\mathsf{struct}(f)$ and the output of the computation $\mathsf{ev}(f, x)$, but not necessarily f itself, and never the input x. The reason we parametrize this definition by $\mathsf{struct}(\cdot)$ is that it allows us to model different settings such as *secure function evaluation* where the structure of the circuit to be evaluated is public ($\mathsf{struct}(f) = f$) and *private function evaluation* where only the number of gates of the circuit is public ($\mathsf{struct}(f) = |G|$ where f describes a circuit with gates G). Note that in the case of secure function evaluation, circuit privacy is trivially satisfied, since S is allowed to know $\mathsf{struct}(f) = f$.

Projective Garbling Scheme A *projective garbling scheme* is an important concrete garbling scheme. In order to describe it, we make the following assumptions:

- We assume that $\mathsf{ev}(f, \cdot)\colon \{0,1\}^n \to \{0,1\}^m$, that is, f describes a Boolean circuit C which computes a function from n bits to m bits. Furthermore, we assume that C has W wires $\{w_i\}_{i=1}^W$ where
  - the first n wires $\{w_i\}_{i \in [n]}$ are the *input wires*,
  - the last m wires $\{w_i\}_{i \in \{W-m+1,\ldots,W\}}$ are the *output wires*, and
  - the remaining wires are called the *internal wires*.

  For simplicity, and without loss of generality, we also assume that C consists only of NAND gates $\{g_i\}_{i=n+1}^W$ where gate g_i has output wire w_i. This means that for all $i \in \{n+1, W\}$,

  $$w_i = \neg(w_{L(i)} \wedge w_{R(i)})$$

  where $L, R\colon \{n+1, \ldots, W\} \to [W]$ are two functions specifying the *left and right input wires* of g_i given the input i. We require that $L(i) \leq R(i) < i$ for all i.

- Finally, we need a *pseudorandom function*

  $$G\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \times [W] \to \{0,1\}^{2\lambda}$$

  to which we assume we have access. We require from G that the output is unpredictable as long as at least one of the two λ-bit inputs is unknown.

We are now ready to describe the projective garbling scheme:

Gb To generate a garbled circuit along with encoding and decoding information, $\mathsf{Gb}(1^\lambda, f)$ does the following:

1. For each wire w_i, choose two random λ-bit strings (K_0^i, K_1^i). Define the decoding information $d = \{(K_0^i, K_1^i)\}_{i\in\{W-m+1,\ldots,W\}}$, and define the encoding information $e = \{(K_0^i, K_1^i)\}_{i\in[n]}$.
2. For each *internal* wire w_i, define a *garbled table* $(C_0^i, C_1^i, C_2^i, C_3^i)$ as follows:
   (a) For all $(a, b) \in \{0,1\} \times \{0,1\}$ compute
   $$C'_{a,b} = G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus (K^i_{\neg(a\wedge b)} \,\|\, 0^\lambda)$$
   where 0^λ means the λ-bit all-zero string which is appended to $K^i_{\neg(a\wedge b)}$ before computing bit-wise the XOR (denoted by $\oplus$).
   (b) Choose a random permutation $\pi\colon \{0,1,2,3\} \to \{0,1\}\times\{0,1\}$ and add
   $$(C_0^i, C_1^i, C_2^i, C_3^i) = (C'_{\pi(0)}, C'_{\pi(1)}, C'_{\pi(2)}, C'_{\pi(3)})$$
   to the garbled function $\langle f\rangle$.

En The encoding function $\mathsf{En}(e, x)$ parses $e = \{K_0^i, K_1^i\}_{i\in[n]}$ and ouputs $\langle x\rangle = \{K^i_{x_i}\}_{i\in[n]}$.

Ev The evaluation function $\mathsf{Ev}(\langle f\rangle, \langle x\rangle)$ parses $\langle x\rangle = \{K^i\}_{i\in[n]}$ and does the following for all i from $n+1$ to W:

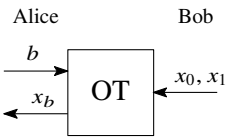
1. Recover $(C_0^i, C_1^i, C_2^i, C_3^i)$ from $\langle f\rangle$.
2. For $j \in 0, 1, 2, 3$, compute
   $$(K'_j, \tau_j) = G(K^{L(i)}, K^{R(i)}, i) \oplus C_j^i.$$
3. If there is a unique j such that $\tau_j = 0^\lambda$, define $K^i = K'_j$. Otherwise, output $\perp$ and abort.

Output $\langle z'\rangle = \{K^i\}_{i\in\{W-m+1,W\}}$.

De The decoding function $\mathsf{De}(d, \langle z\rangle)$ parses $d = \{(K_0^i, K_1^i)\}_{i\in\{W-m+1,W\}}$ and $\langle z\rangle = \{K^i\}_{i\in\{W-m+1,W\}}$ and defines each output bit $z_i = b$ where $K^i = K_b^i$. If this definition fails for any i, De outputs $\perp$ and aborts. Otherwise, De outputs $z = \{z_i\}_{i\in\{W-m+1,W\}}$.

3.4.2 2PC Protocol

We now present the *Garbled Circuits* 2PC protocol for $\mathbb{F} = \{0,1\}$. It builds on a projective garbling scheme and a $\binom{2}{1}$-*oblivious transfer* (OT) protocol which is an important cryptographic primitive that enables two parties, Alice and Bob, to solve the following problem: Alice has a bit $b \in \{0,1\}$, and Bob has two strings $x_0, x_1 \in \{0,1\}^*$. Now, they want Alice to learn x_b without learning x_{1-b}. Furthermore, Bob cannot learn b. An ideal functionality, $\mathcal{F}_{\mathsf{OT}}$, is specified as follows:



Functionality $\mathcal{F}_{\mathsf{OT}}$

Transfer On input $(\mathsf{Transfer}, b)$ where $b \in \{0,1\}$ from Alice and input $(\mathsf{Transfer}, x_0, x_1)$ from Bob, output x_b to Alice.

Functionality 3.1: The $\binom{2}{1}$-Oblivious Transfer functionality.

We will not cover in detail how to realize this functionality but simply mention that one generic way of doing so is to use a *public key encryption* scheme that supports *oblivious key generation*. Such a scheme should be able to generate an oblivious public key that is indistinguishable from a real public key, but where it is not possible to learn the secret key corresponding to the oblivious public key. For instance, the famous ElGamal cryptosystem [11] can be extended to support oblivious key generation and hence, it can be used to realize (passively secure) OT.

Given access to $\mathcal{F}_{\mathsf{OT}}$ and using a projective garbling scheme, a passively secure 2PC protocol can be implemented as follows:

Garbled Circuits

Setup Alice is the evaluator and Bob is the garbler. Alice and Bob have private inputs $x \in \{0,1\}^{n_A}$ resp. $y \in \{0,1\}^{n_B}$ where $n_A + n_B = n$ and want to compute $\mathsf{ev}(f, x \,\|\, y)$ where $f \in \{0,1\}^*$ is a description of a circuit computing a function from n bits to m bits. They also agree on a security parameter λ.

Garble Bob computes $(\langle f \rangle, (e_x \,\|\, e_y), d) \leftarrow \mathsf{Gb}(1^\lambda, f)$ and sends $\langle f \rangle$ to Alice. $(e_x \,\|\, e_y)$ means that e is divided into two parts for encoding x and y.

Encode Bob's input Bob computes $\langle y \rangle \leftarrow \mathsf{En}(y, e_y)$ and sends $\langle y \rangle$ to Alice.

Encode Alice's input Since we use a projective garbling scheme, we know that $e_x = \{K_0^i, K_1^i\}_{i\in[n_A]}$. Furthermore, Alice's input is a vector of bits $x \in \{0,1\}^{n_A}$. This means that Alice and Bob can use n_A calls to $\mathcal{F}_{\mathsf{OT}}$ in order for Alice to learn e_x without Bob learning x in the following way: For all $i \in [n_A]$, Alice inputs $(\mathsf{Transfer}, x_i)$ and Bob inputs $(\mathsf{Transfer}, K_0^i, K_1^i)$ to $\mathcal{F}_{\mathsf{OT}}$.

Finally, Alice can compute $\langle x\rangle \leftarrow \mathsf{En}(x, e_x)$.

Evaluate Alice computes $\langle z'\rangle \leftarrow \mathsf{Ev}(\langle f\rangle, \langle x\rangle \,\|\, \langle y\rangle)$.

Reveal output to Bob Alice sends $\langle z'\rangle$ to Bob who outputs $z \leftarrow \mathsf{De}(d, \langle z'\rangle)$.

Reveal output to Alice Bob sends d to Alice who outputs $z \leftarrow \mathsf{De}(d, \langle z'\rangle)$.

Protocol 3.1: The Garbled Circuits MPC protocol.

Security and Efficiency Protocol 3.1 is passively secure given circuit privacy of the underlying projective garbling scheme and access to $\mathcal{F}_{\mathsf{OT}}$. Actively secure variants exist, if one is willing to sacrifice some degree of efficiency, but the details are not in the scope of this project. One particularly nice property of this protocol is that it has a constant number of communication rounds (most rounds can actually be merged to achieve almost optimal round complexity) as opposed to the SPDZ protocol [7] which, as we shall see, has a round complexity proportional to the multiplicative depth of the computed circuit. On the other hand, an advantage of SPDZ is that it immediately generalizes to an arbitrary number of parties. This is not the case for GC which is inherently a 2PC protocol.

3.5 SPDZ

The SPDZ protocol was first introduced by Damgård et al. [7] in 2011 and subsequently improved in several papers [23, 24]. The following presentation is due to a recent survey from 2022 by Orsini [30]. SPDZ is an MPC protocol that is actively secure in the presence of a dishonest majority of corrupt parties. It relies on a technique called *secret sharing* to ensure privacy while the parties jointly evaluate a circuit, gate by gate in topological order, on their respective inputs, in order to obtain an output that is revealed to an agreed-upon subset of the parties. As opposed to GC, SPDZ works in the *preprocessing model*, meaning that it consists of a slow *preprocessing phase*, which is independent of both the function to be computed and the parties' private inputs, followed by a very fast *online phase*. Originally, SPDZ works on arithmetic circuits over sufficiently large finite fields, but it has since been adapted to various other settings, notably also Boolean circuits. We will focus on the arithmetic variant and assume that we work in some finite field $\mathbb{F}$. We now describe how authenticated secret sharing works in SPDZ, and then we proceed to describe the preprocessing and online phases of the protocol.

3.5.1 Authenticated Secret Sharing

We start by defining secret sharing in general and the important technique of *linear secret sharing* which is used in SPDZ.

Definition 3.11 (Secret Sharing)**.** Let $n \in \mathbb{N}_2$. We say that a value s from a set S is *secret shared* between n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ if it is the case that

1. for all $i \in [n]$, P_i holds a secret share $s^{(i)}$ from a set T,
2. no combination of $n - 1$ secret shares reveal anything about s, and
3. there exists a surjective function $\text{Combine}\colon T^n \to S$ where it holds that $\text{Combine}(s^{(1)}, \ldots, s^{(n)}) = s$.

Adhering to Notation 2.1, we write $\langle s \rangle$ for the collection of all secret shares of s, that is $\langle s \rangle = (s^{(1)}, \ldots, s^{(n)})$. △

Definition 3.12 (Linear Secret Sharing)**.** An important version of Definition 3.11 that we will be using is *linear secret sharing* (LSS), sometimes also called additive secret sharing. In linear secret sharing, an integer $x \in \mathbb{F}$ is divided into n uniformly random shares as $\langle x \rangle = (x^{(1)}, \ldots, x^{(n)})$ such that $x = \text{Combine}(\langle x \rangle) = \sum_{i=1}^{n} x^{(i)}$. Due to linearity, we can define secret addition as a local operation on the ith share of x and y for all $i \in [n]$:

$$\begin{aligned} \langle x \rangle + \langle y \rangle &:= \left(x^{(1)} + y^{(1)}, \ldots, x^{(n)} + y^{(n)} \right) \\ &= \langle x + y \rangle \,. \end{aligned}$$

In a similar, local way, we can define addition with a public constant and multiplication by a public constant. We will later see that multiplication of secret-shared values requires interaction between all parties and the use of correlated randomness. △

In order to achieve active security, it is not enough to secret-share values since actively corrupt parties may choose to deviate from the protocol by altering their secret shares during the circuit computation. In order to mitigate this, SPDZ relies on something called *authenticated secret sharing* using an information-theoretic MAC scheme (see Definition 2.17 on page 10). Authentication is performed using a global secret-shared MAC key as follows.

Representation A value $x \in \mathbb{F}$ is linearly secret shared along with a global MAC key $\Delta \in \mathbb{E}$, where $\mathbb{F} \subseteq \mathbb{E}$ (usually, $\mathbb{E} = \mathbb{F}$, but if $\mathbb{F}$ is small, $\mathbb{E}$ needs to be a superset in order for the MAC scheme to be secure) and the computed MAC $\mathsf{m}_\Delta(x)$ on x which is simply defined by $\mathsf{m}_\Delta(x) = x \cdot \Delta$ (where $\cdot$ denotes multiplication in $\mathbb{E}$).

We write $\langle x \rangle_S$ for the authenticated secret-shared value x, and let

$$\langle x \rangle_S = (\langle x \rangle, \langle \mathsf{m}_\Delta(x) \rangle, \langle \Delta \rangle) = \left[\left(x^{(i)} \right)_{i=1}^n, \left(\mathsf{m}_\Delta(x)^{(i)} \right)_{i=1}^n, \left(\Delta^{(i)} \right)_{i=1}^n \right]$$

where x, $\mathsf{m}_\Delta(x)$ and Δ are additively secret shared, i.e.

$$\begin{aligned} x &= \sum_{i=1}^n x^{(i)}, \\ \mathsf{m}_\Delta(x) &= \sum_{i=1}^n \mathsf{m}_\Delta(x)^{(i)}, \text{ and} \\ \Delta &= \sum_{i=1}^n \Delta^{(i)}, \end{aligned}$$

between parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$, such that P_i knows the share

$$x_S^{(i)} = (x^{(i)}, \mathsf{m}_\Delta(x)^{(i)}, \Delta^{(i)}).$$

Arithmetic An important property of the SPDZ authenticated secret sharing is that we can – as in regular linear secret sharing – perform addition locally on the parties' shares of any $x, y \in \mathbb{F}$ authenticated under the same key Δ:

$$\begin{aligned} \langle x \rangle_S + \langle y \rangle_S &\coloneqq (\langle x \rangle + \langle y \rangle, \langle \mathsf{m}_\Delta(x) \rangle + \langle \mathsf{m}_\Delta(y) \rangle, \langle \Delta \rangle) \\ &= (\langle x + y \rangle, \langle \mathsf{m}_\Delta(x) + \mathsf{m}_\Delta(y) \rangle, \langle \Delta \rangle) \\ &= (\langle x + y \rangle, \langle \mathsf{m}_\Delta(x + y) \rangle, \langle \Delta \rangle) \\ &= \langle x + y \rangle_S . \end{aligned}$$

The above equalities hold due to linearity of the underlying LSS and MAC schemes. In a similar, straightforward way, we can perform local constant addition and constant multiplication. To add a public constant c to a secret value $\langle x \rangle_S$, only P_1 needs to add c to their share $x^{(1)}$ of x. To multiply $\langle x \rangle_S$ by c, all parties need to multiply their share $x^{(i)}$ by c. The MAC value on c is in both public cases defined by each party setting $\mathsf{m}_\Delta^{(i)}(c) = c \cdot \Delta^{(i)}$ which they then add to their share $\mathsf{m}_\Delta(x)^{(i)}$. Multiplication of secret-shared values requires communication between the parties as we shall see in the description of the online phase.

Verification Using authenticated secret sharing, the honest parties can ensure that all parties have disclosed correct shares by verifying the MAC on the sum of the shares. The trivial way of performing this verification is to disclose all MAC and key shares. However, this is inefficient since then the MAC key is not secret afterwards, and a new key must therefore be generated for each revealed value. Of course, the parties could wait until the end of the computation to reveal their shares, allowing the MAC key to be used throughout the circuit evaluation. Still, this is a problem if we want to reuse the key for *reactive computation* where new circuits are generated and evaluated based on the output of previous circuit evaluations. There is a more

clever check that allows the parties to verify a MAC (or a batch of MACs efficiently) without disclosing the global key at all:

Single MAC check To verify a single secret-shared MAC $\langle \mathsf{m}_\Delta(x) \rangle$ against a revealed value x', each party P_i computes

$$\sigma^{(i)} = \mathsf{m}_\Delta(x)^{(i)} - x' \cdot \Delta^{(i)} = \mathsf{m}_\Delta(x)^{(i)} - \mathsf{m}_\Delta(x')^{(i)}.$$

Then, using a *commitment functionality*[2], P_i commits to $\sigma^{(i)}$, and when every party has made their commitment, all commitments are opened to all parties, allowing every party to check that

$$\sum_{i=1}^{n} \sigma^{(i)} = \mathsf{m}_\Delta(x) - \mathsf{m}_\Delta(x') = 0.$$

Batch MAC check Using a random shared vector $\mathbf{r} = (r_1, \ldots, r_t) \in_R \mathbb{F}^t$ (which the parties can jointly sample using a functionality that is realized with the help of commitments), the parties can batch check t MACs on $x'_1, \ldots, x'_t$ as follows: Each party computes

$$x' = \sum_{i=1}^{t} r_i \cdot x'_i$$

and

$$\mathsf{m}_\Delta(x)^{(i)} = \sum_{i=1}^{t} r_i \cdot \mathsf{m}_\Delta(x_i)^{(i)}$$

Then, the parties verify the MAC $\langle m_\Delta(x) \rangle$ against x' using the single MAC check described above.

3.5.2 Preprocessing Phase

We now describe an ideal preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$ that is capable of performing the minimum amount of preprocessing needed by the online phase of the SPDZ protocol. In the following, we omit "authenticated" when mentioning secret sharing since it is always the case that we perform authenticated secret sharing.

We say that SPDZ is in the *correlated randomness* preprocessing model. "Correlated randomness" refers to the fact that a major part of the preprocessing consists of generating so-called *Beaver triples*, invented by Beaver [4] in 1991, of correlated random values and secret-sharing them between the parties such that they can be consumed by a very fast online phase.

[2] A commitment functionality allows a party P to *commit* to any value and share the commitment with other parties. At any time after having committed, P can choose to *open* their commitment to other parties, revealing the value. Two guarantees are provided: *Hiding* guarantees that no other party can learn which value P committed to before P chooses to open the commitment. *Binding* guarantees that P can never open their commitment to a value that is different from the one they committed to in the first place.

Definition 3.13 (Beaver Triple)**.** A *Beaver triple* is a triple of uniformly random numbers $a, b, c \in_R \mathbb{F}$ subject to $c = a \cdot b$. △

In addition to Beaver triples, $\mathcal{F}_{\mathsf{Prep}}$ should also be able to generate and secret-share uniformly random values $r \in_R \mathbb{F}$ such that only one party P_i knows r while every party knows a secret share of r. We shall see in the description of the online phase that P_i can use their knowledge of r to secret-share one of their private inputs with the other parties. Importantly, all secret shares of this kind of random value must also be random – specifically, no corrupt party must be allowed to influence how the secret shares are generated. Finally, we assume that $\mathcal{F}_{\mathsf{Prep}}$ is able to generate a uniformly random MAC key $\Delta \in_R \mathbb{E}$ and distribute secret shares of it to all parties. The complete functionality $\mathcal{F}_{\mathsf{Prep}}$ is specified as follows:

Functionality $\mathcal{F}_{\mathsf{Prep}}$

Initialize On input $(\mathsf{Init}, \mathbb{F}, \mathbb{E})$, check that $\mathbb{F} \subseteq \mathbb{E}$. If not, output $\perp$ and abort. Otherwise, store $\mathbb{F}$ as the circuit domain and store $\mathbb{E}$ as the MAC domain.

Generate MAC key On input Key from all parties, generate a uniformly random MAC key $\Delta \in_R \mathbb{E}$ and distribute shares of $\langle \Delta \rangle_S$ to all parties.

Generate input mask On input $(\mathsf{InputMask}, \mathsf{P}_i)$ from all parties, generate a uniformly random value $r \in_R \mathbb{F}$, reveal r to P_i, and distribute shares of $\langle r \rangle_S$ to all parties.

Generate Beaver triple On input Triple from all parties, generate a Beaver triple (a, b, c) and distribute shares of $(\langle a \rangle_S, \langle b \rangle_S, \langle c \rangle_S)$ to all parties.

Functionality 3.2: The SPDZ preprocessing functionality.

Implementation and Efficiency $\mathcal{F}_{\mathsf{Prep}}$ is realized in practice using a *1-leveled threshold homomorphic encryption* scheme. "Threshold" means, at a high level, that the encryption scheme allows *distributed decryption* of ciphertexts by letting each party keep a secret share of the secret key and allowing each party to locally compute a *partial decryption* which is in fact a *secret share* of the plaintext. "Homomorphic" means that we can add and multiply ciphertexts *as if* they were plaintexts, i.e.

$$\mathsf{E}_K(m_1) + \mathsf{E}_K(m_2) = \mathsf{E}_K(m_1 + m_2) \quad \text{and} \quad \mathsf{E}_K(m_1) \cdot \mathsf{E}_K(m_2) = \mathsf{E}_K(m_1 \cdot m_2)$$

where $\mathsf{E}_K(m)$ denotes encryption of the message m under the key K. Finally, "1-leveled" means that the scheme is secure when homomorphically performing at most 1 multiplication of ciphertexts before decrypting the result. With such an encryption scheme available, $\mathcal{F}_{\mathsf{Prep}}$ can be realized with passive or active security as follows:

Passive security The parties can essentially create input masks and multiplication triples (as well as other kinds of correlated randomness such as random bits which are used in practice to optimize parts of the computation) by sampling random values locally, distributing shares of these locally sampled random values, encrypting the shares they have received from other parties, computing encrypted MACs on encrypted shares using an encrypted MAC key, and finally, performing distributed decryption on the encrypted MACs to obtain secret MAC shares. By following this outline, we can realize a passively secure preprocessing phase.

Active security Active security is much harder to enforce; In most implementations, active security is achieved using so-called *zero-knowledge proofs of knowledge* (ZKPs) which is a primitive that the parties can use to prove to other parties that they know the plaintext of a given ciphertext, but without revealing the plaintext (or anything else) to the other party. Such proofs are very costly to execute in practice, which is the main reason for the SPDZ preprocessing phase being much slower than the online phase. A substantial amount of work has been devoted to optimizing the ZKP part of the preprocessing phase. Keller, Orsini, and Scholl [23] show how to replace the use of ZKPs in the original SPDZ protocol [7] with the use of OT (see Functionality 3.1 on page 19). Subsequently, Keller, Pastro, and Rotaru [24] show that using the original ZKP is in fact more efficient in practice (unless working with Boolean circuits). They introduce two state-of-the-art, actively secure protocols, *LowGear* and *HighGear*, that are optimized for a small resp. large number of parties. Finally, an additional technique known as *sacrifice* is required in order to ensure correctness of correlated multiplication triples. As hinted by the name of the technique, it works by sacrificing (revealing) one triple in order to be able to use another (indpendent) triple, such that malicious parties have no way of knowing in advance which of the two triples is going to be sacrificed.

3.5.3 Online Phase

For the description of the online phase of SPDZ, we assume access to $\mathcal{F}_{\mathsf{Prep}}$ and show how to use it for input sharing and the evaluation of arithmetic circuits. The complete online protocol is specified as follows:

SPDZ

Initialize Parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ agree on circuit domain $\mathbb{F}$ and MAC domain $\mathbb{F}$ and input $(\mathsf{Init}, \mathbb{F}, \mathbb{E})$ to $\mathcal{F}_{\mathsf{Prep}}$. If $\mathcal{F}_{\mathsf{Prep}}$ aborts, the parties output $\perp$ and abort. Otherwise, they proceed to input Key to $\mathcal{F}_{\mathsf{Prep}}$ to obtain secret shares $\langle\Delta\rangle$ of the generated global MAC key $\Delta \in_R \mathbb{E}$.

Input When P_i wants to secret-share their private input $x \in \mathbb{F}$, all parties input $(\mathsf{InputMask}, \mathsf{P}_i)$ to $\mathcal{F}_{\mathsf{Prep}}$ such that P_i learns a random value $r \in_R \mathbb{F}$ along with $r_S^{(i)}$, and P_j learns $r_S^{(j)}$ for all $j \neq i$. Then, P_i broadcasts $x - r$ and the parties locally compute $\langle x \rangle_S = \langle r \rangle_S + (x - r)$.

Evaluate The gates of the circuit are evaluated in topological order, and the following actions are performed based on the type of gate:

Add constant To add a public constant $c \in \mathbb{F}$ to $\langle x \rangle_S$, the parties locally compute $\langle x \rangle_S + c = \langle x + c \rangle_S$.

Multiply by constant To multiply $\langle x \rangle_S$ by a public constant $c \in \mathbb{F}$, the parties locally compute $c \cdot \langle x \rangle_S = \langle c \cdot x \rangle_S$.

Add To add $\langle x \rangle_S$ and $\langle y \rangle_S$, the parties locally compute $\langle x \rangle_S + \langle y \rangle_S$.

Multiply To multiply $\langle x \rangle_S$ and $\langle y \rangle_S$, the parties input Triple to $\mathcal{F}_{\mathsf{Prep}}$ to obtain a fresh secret-shared Beaver triple $(\langle a \rangle_S, \langle b \rangle_S, \langle c \rangle_S)$. Then, the parties do the following:

1. They compute $\langle \epsilon \rangle_S = \langle x \rangle_S - \langle a \rangle_S$ and $\langle \rho \rangle_S = \langle y \rangle_S - \langle b \rangle_S$.
2. They open $\langle \epsilon \rangle_S$ and $\langle \rho \rangle_S$ and perform a MAC check on both opened values. If any check fails, the parties output $\perp$ and abort.
3. They locally compute

$$\langle x \rangle_S \cdot \langle y \rangle_S := \langle c \rangle_S + \epsilon \cdot \langle b \rangle_S + \rho \cdot \langle a \rangle_S + \epsilon \cdot \rho = \langle x \cdot y \rangle_S .$$

Output To output a value $\langle z \rangle_S$, the parties open the value and run a MAC check on the opened value. If the check fails, the parties output $\perp$ and abort. Otherwise they output the opened, authenticated value.

Protocol 3.2: The online SPDZ MPC protocol.

Correctness We have already argued correctness of constant addition, constant multiplication, and regular addition due to linearity of LSS and the MAC scheme. The multiplication procedure uses the so-called *Beaver's trick* and is correct since

$$\begin{aligned} c + \epsilon b + \rho a + \epsilon\rho &= c + (x - a)b + (y - b)a + (x - a)(y - b) \\ &= ab + xb - ab + ya - ba + xy - xb - ay + ab \\ &= xy. \end{aligned}$$

Security It is clear that the input sharing is secure since r is uniformly random in $\mathbb{F}$ so it acts as a one-time pad to perfectly hide the private input x. Multiplication is secure since no parties know a, b, or c so revealing ϵ and ρ does not reveal anything about x or y since they are masked by uniformly random a and b. Finally, the MAC checks ensure that we can always detect active corruption, so the dishonest parties cannot trick the honest parties into revealing information by disclosing wrong shares.

3.6 The Arithmetic Black Box Model

The arithmetic black box (ABB) is an ideal functionality that can be used as an abstract interface when working with circuit-based MPC protocols like SPDZ. It will be particularly useful when combining oblivious data structures and MPC in Chapter 5 because it allows so-called *reactive programming* where the parties can learn some intermediate result and provide new input that depends on this result. In essence, the parties executing a protocol in the ABB model are given pointers to elements in a black box and can instruct the box to perform certain operations on the elements. More formally, we present $\mathcal{F}_{\mathsf{ABB}}$ as follows, inspired by Keller, Orsini, and Scholl [23]:

Functionality $\mathcal{F}_{\mathsf{ABB}}$

Initialize On input $(\mathsf{Init}, \mathbb{F})$ from all parties, store $\mathbb{F}$.

Input On input $(\mathsf{Input}, \mathsf{P}_i, \mathsf{id}, x)$ from P_i and $(\mathsf{Input}, \mathsf{P}_i, \mathsf{id})$ from all other parties, store (id, x).

Add On input $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, retrieve (id_1, x), (id_2, y) and store $(\mathsf{id}_3, x + y)$.

Multiply On input $(\mathsf{Mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, retrieve (id_1, x), (id_2, y) and store $(\mathsf{id}_3, x \cdot y)$.

Add constant On input $(\mathsf{CAdd}, \mathsf{c}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties, retrieve (id_1, x) and store $(\mathsf{id}_2, c + x)$,

Multiply by constant On input $(\mathsf{CMult}, \mathsf{c}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties, retrieve (id_1, x) and store $(\mathsf{id}_2, c \cdot x)$,

Reveal On input $(\mathsf{Reveal}, \mathsf{id})$ from all honest parties, retrieve (id, z) and output it to the adversary. Wait for an input from the adversary. If this is $\mathsf{Deliver}$, then output z to all parties, otherwise output $\perp$ and abort.

Functionality 3.3: The Arithmetic Black Box functionaliy.

We say that a given MPC protocol is in the ABB model if it implements $\mathcal{F}_{\mathsf{ABB}}$.

Remark 3.14. The SPDZ online protocol with access to $\mathcal{F}_{\mathsf{Prep}}$ implements $\mathcal{F}_{\mathsf{ABB}}$, so it is in the ABB model. ▲

Chapter 4

Oblivious Data Structures

It is a well-known security problem that accessing data, even when it is encrypted, can reveal sensitive information to parties observing the *data access pattern*. For instance, Pinkas and Reinman [32] give the example that accessing one or more encrypted data locations can indicate a certain stock trading transaction – information that is considered highly sensitive. They also argue that adversaries observing the access pattern to an encrypted database might focus their resources on breaking the encryption of the most frequently accessed database entries since it is often a reasonable assumption that these entries are important. Solving this problem of hiding the data access pattern is of great interest in any scenario where data has to be stored in an insecure location, for instance in the case of *cloud storage* or when implementing a *secure processor* that relies on the use of *insecure memory*. Furthermore, efficient, oblivious data access is relevant in an MPC context, since, as we have seen, making input-dependent memory access oblivious in a circuit incurs a linear overhead if no additional measures are taken.

An *oblivious data structure* is a data structure in the RAM model of computation that solves this security problem by hiding the *logical* memory access pattern generated by a *sequence of operations* carried out on the data structure. This is achieved by *translating* the logical access pattern into a different *physical* access pattern that leaks no information about the sequence of data structure operations, except some predetermined allowed information such as the number, and perhaps the types, of operations. Not surprisingly, hiding the logical access pattern in this way comes at a cost; when analyzing the efficiency of oblivious data structures, they are compared to their non-oblivious baselines, and the operational *bandwidth overhead* becomes an indicator of performance; indeed, for many oblivious data structures, lower bounds have shown that an asymptotic overhead is unavoidable. Jacob, Larsen, and Nielsen [19] prove $\Omega(\log n)$ amortized lower bounds for several oblivious data structures, including the *oblivious priority queue* which we will study in this thesis. All of these lower bounds imply an inherent asymptotic overhead over the best known non-oblivious variants.

Remark 4.1 (Oblivious Data Structures vs. Encryption). As we have argued, the problem of hiding the data access pattern is different from the problem of hiding the contents of the data. When discussing oblivious data structures, we always implicitly assume that we have the means to securely encrypt our data when storing it in an untrusted location. Under this assumption, the stored, encrypted data by itself does not reveal anything about the unencrypted data – but when the data is accessed, we must still take care not to reveal the logical access pattern. ▲

Oblivious data structures come in many forms that complement each other and perform well in different settings. We will now formally define properties of a general oblivious data structure, and then we will present two approaches in detail: the important, general-purpose oblivious RAM structure and the more specialized oblivious priority queue.

4.1 Formal Definition

We define oblivious data structures along the lines of Wang et al. [38]. First, we define a general (not necessarily oblivious) data structure:

Definition 4.2 (Data Structure)**.** A *data structure* is a collection of data supporting a set of possibly probabilistic *operations* in the RAM model of computation such as Insert, Delete or Access. Every operation is parametrized by a set of operands, i.e. an operation is a tuple, $(\mathsf{op}, \vec{\mathsf{args}})$ where op is the *operation code*, and $\vec{\mathsf{args}}$ is a vector of *arguments*. A data structure must always support an Initialize operation, which possibly accepts arguments such as a size, a security parameter, and some initial data, and configures the data structure to be in an initial state based on those arguments. △

We always require that a data structure has the same input-output behavior (up to a negligible error probability) as an ideal functionality:

Definition 4.3 (Data Structure Correctness)**.** Let $\mathcal{D}$ be a data structure with security parametrized by λ. Let $\mathcal{F}_{\mathcal{D}}$ be an ideal functionality that specifies an ideal behavior of the operations supported by $\mathcal{D}$. We say that $\mathcal{D}$ implements $\mathcal{F}_{\mathcal{D}}$ correctly if, for any polynomial-length input $x \in \{0,1\}^*$, for any polynomial-length sequence of operations $\vec{\mathsf{ops}}$ supported by $\mathcal{D}$ (and thus also by $\mathcal{F}_{\mathcal{D}}$), and for any program Π, there exists a function μ such that

$$\Pr\left[\Pi_{(\mathcal{D},\vec{\mathsf{ops}})}(x) \neq \Pi_{(\mathcal{F}_{\mathcal{D}},\vec{\mathsf{ops}})}(x)\right] < \mu(\lambda)$$

where μ is negligible in λ and $\Pi_{(\mathcal{D},\vec{\mathsf{ops}})}(x)$ means the output of Π executed on input x that carries out the sequence $\vec{\mathsf{ops}}$ on the data structure $\mathcal{D}$. △

Remark 4.4 (Security Parameter). The *security parameter*, denoted by λ, controls the degree of security of the oblivious data structure while affecting the performance. It is passed as an additional parameter during initialization. Sometimes this is done implicitly if the parameter is not used in the the high-level specification. ▲

In order to define obliviousness, we need a notion of *generated addresses* and *leakage*.

Definition 4.5 (Operation Sequence Addresses)**.** Let $\mathcal{D}$ be a data structure, and let $\vec{\mathsf{ops}}$ be a sequence of operations supported by $\mathcal{D}$. We define $\mathsf{addresses}_{\mathcal{D}}(\vec{\mathsf{ops}})$ to be the *sequence of physical memory addresses* that are accessed when executing $\vec{\mathsf{ops}}$ on $\mathcal{D}$. △

Definition 4.6 (Operation Sequence Leakage)**.** Let $\mathcal{D}$ be a data structure, and let $\vec{\mathsf{ops}}$ be a sequence of operations supported by $\mathcal{D}$. We denote by $\mathcal{L}_{\mathcal{D}}(\vec{\mathsf{ops}})$ the *leakage function* and it represents the information leakage generated by executing $\vec{\mathsf{ops}}$ on $\mathcal{D}$. △

Remark 4.7. Typically,

$$\mathcal{L}_{\mathcal{D}}(\vec{\mathsf{ops}}) = |\vec{\mathsf{ops}}| \quad \text{or} \quad \mathcal{L}_{\mathcal{D}}(\vec{\mathsf{ops}}) = \{\mathsf{op}^i\}_{i=1}^{|\vec{\mathsf{ops}}|}$$

where the latter means that the sequence of operation types (and therefore also implicitly the length) of $\vec{\mathsf{ops}}$ is leaked. See Remark 4.10 for a further discussion of leakage. ▲

With all these definitions in place, we can now finally define what it means for a data structure to be *oblivious*.

Definition 4.8 (Oblivious Data Structure)**.** Let $\mathcal{D}$ be a data structure. $\mathcal{D}$ is said to be *oblivious* if there exists a polynomial-time simulator, $\mathcal{S}$, such that for any polynomial-length sequence of operations, $\vec{\mathsf{ops}}$, supported by $\mathcal{D}$,

$$\mathsf{addresses}_{\mathcal{D}}(\vec{\mathsf{ops}}) \cong \mathcal{S}(\mathcal{L}_{\mathcal{D}}(\vec{\mathsf{ops}})).$$

△

Remark 4.9 (Naïve Oblivious Data Structure with Linear Access Overhead). A naïve way of constructing an oblivious data structure from its non-oblivious baseline is to always emulate the *superset* of memory accesses for all operation types (see Remark 4.10), and to furthermore perform a linear scan of the underlying memory when accessing one address. This guarantees perfect obliviousness since the distribution of memory access patterns for any two operation sequences of the same length is clearly exactly the same. However, this approach yields a $\Omega(n)$ access bandwidth overhead which is unacceptable in most applications, so this is mainly a theoretically interesting possibility result. ▲

Remark 4.10 (Type Hiding Security). If we do not emulate the superset of memory accesses for all operation types, we risk leaking the operation types in addition to the length of a sequence of operations. In some cases, e.g. when doing *oblivious sorting* using a priority queue, a topic we treat in Section 4.3, it is fine to define the leakage function to allow this leakage, but in any case, it is important to consider whether revealing operation types compromises security. By "emulating the superset of

memory accesses for all operation types", we mean that for every type of operation, we always emulate the memory access pattern of all operation types *in a fixed order*, replacing every operation but the actual operation by a dummy operation. This makes the memory access pattern perfectly independent of the type of operation – in this case we say that a data structure is *type oblivious*. Emulating the superset of memory accesses for all operation types means that all operation types end up having the same algorithmic complexity, namely the sum of the complexities of all the individual operation types. In the following, we do not assume that we take measures to ensure type obliviousness, unless specified, but it should be understood that measures can always be taken in the way described in this remark. ▲

4.2 Oblivious RAM

Oblivious RAM (ORAM) was first introduced by Goldreich [13] in 1987 and the foundational theory further developed by Goldreich and Ostrovsky [14] in 1996 in the context of *software protection* where a trusted CPU with limited storage capacity needs to outsource storage to an untrusted memory location where adversaries can observe the memory access pattern. In line with Definition 4.8 on the preceding page, they present ORAM as a RAM construction that is allowed to be probabilistic, and that hides the logical memory access pattern of a program by *compiling* it to a program with a different physical memory access pattern while preserving the input/output behavior of the original program. Another more dynamic view is that an ORAM scheme is a layer between a program and external memory that exposes a RAM interface but takes care of translating sequence of logical access requests of the program in order to hide the requests.

ORAM is the most general-purpose oblivious data structure; given *any* program – particularly a program using any other data structure – ORAM can be used to compile it to an oblivious form. However, in many cases, more specialized implementations of other oblivious data structures perform better in practice, since they are not limited by the lower bounds of ORAM which we will discuss later in this section.

Applications In addition to the original setting of software protection, *cloud storage* is another setting in which the application of ORAM has turned out to be of great practical interest [34, 35]: In this setting, a *client* has a large dataset that they want to work on but limited local storage capacity. Therefore, they would like to outsource the task of storing the data to an untrusted *server* in the cloud while keeping the data and how they access it secret. In this setting, it is clear that an ORAM scheme with linear client-side storage complexity is unacceptable; in that case, the client might just as well store the original data themselves. Furthermore, linear per-entry access bandwidth is far too expensive for many practical applications – even more so when the communication between the client and the server is carried out over a slow network connection. Luckily, it turns out that ORAM schemes with sublinear client storage complexity and sublinear access bandwidth do exist. In

the following, we will adopt the client-server terminology when discussing ORAM and other oblivious data structures. Finally, a different line of research starting with Gordon et al. [16] focuses on applying ORAM in a reactive MPC setting to circumvent the otherwise inherent linear access time per entry. We will study this application in Chapter 5.

Specification An ORAM scheme must expose a regular RAM interface and abstract away the details of how the logical access pattern is hidden. It also makes sense to think of this data structure as an *oblivious array*. Formally, this means that it must expose an Initialize operation and an Access operation. We specify the ideal RAM functionality $\mathcal{F}_{\mathsf{RAM}}$ as follows:

Functionality $\mathcal{F}_{\mathsf{RAM}}$

Initialize On input $(\mathsf{Init}, \mathsf{entries})$ where entries is a sequence of n entries, indexed from 0 to $n-1$, store all entries by index.

Access On input $(\mathsf{Access}, i, \Phi)$ where $i \in [n]_0$ is a logical index, and Φ is an operation, replace the value at index i with the output of executing Φ on the value at index i.

Functionality 4.1: The ideal RAM functionaliy.

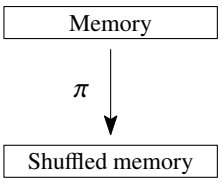
Remark 4.11. Given the availability of Access, the operations Read and Write can be implemented in a straightforward way using Φ, so we will only specify Access in the following. ▲

Lower Bounds Goldreich and Ostrovsky [14] proved an amortized $\Omega(\log n)$ lower bound on the access bandwidth complexity of a *statistically oblivious* ORAM of size n – even when the ORAM is in the so-called *offline model*, meaning that it receives the entire sequence of operations ahead of time. Larsen and Nielsen [27] recently strengthened this bound to also apply to *computationally oblivious* ORAM schemes in the *online model*, meaning that the sequence of operations are revealed dynamically during computation. The ORAM schemes with which we are concerned in this thesis are all in the online model.

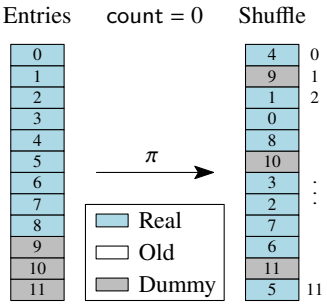
Since the introduction of ORAM, two major approaches to building efficient ORAM have crystallized in the literature which we will now describe in detail: *square-root ORAM* and *tree ORAM*.

4.2.1 Square-Root ORAM

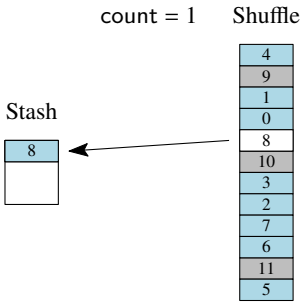
The *square-root ORAM* scheme was introduced by Goldreich and Ostrovsky [14] in 1996. The basic idea of the scheme is that the client samples a uniformly random permutation π, initializes the ORAM by obliviously permuting the memory according to π – in order to break the association between logical and physical indices being accessed – and sends the permuted memory entries to the server. The server-side memory is called the *shuffle*. Next, the client locally stores some representation of π and uses this to map logical indices that are to be accessed to physical indices in the server-side shuffle. If no entry in the shuffle is accessed repeatedly, this construction immediately satisfies Definition 4.8. In order to allow repeated accesses, a small client-side *stash* of size T is used to cache already accessed entries so the server does not learn if an entry is being accessed repeatedly. To completely hide that an entry from the stash is being accessed, T *dummy entries* are appended to the initial entries before they are all permuted and sent to the server, and a *dummy request* is being issued to the shuffle for each stash access, which means fetching a fresh dummy entry and storing it in the stash. After T accesses, the stash might be full of real entries from the server's point of view. At this point, a fresh permutation π' is sampled uniformly at random by the client, the shuffle is obliviously *reshuffled* according to $\pi' \circ \pi^{-1}$, and the stash is emptied at the same time, updating the real entries in the shuffle with data from the stash entries. This reshuffling step is repeated after every T accesses for the entire lifetime of the ORAM, resulting in a cycle with $T - 1$ cheap access operations followed by 1 expensive access operation. Thus, square-root ORAM has a superlinear *worst-case* access cost and a sublinear *amortized* access cost. In this thesis, we present a simple construction that, as we shall see, results in a $\Theta(\sqrt{n})$ amortized access cost. However, it should be noted that similar, albeit more involved, *hierarchical* approaches exist which yield polylogarithmic amortized access costs [14, 15], based on recursion and *cuckoo hashing*. Refer to Fig. 4.1 on the next page for an example of a simple square-root ORAM cycle and initialization.



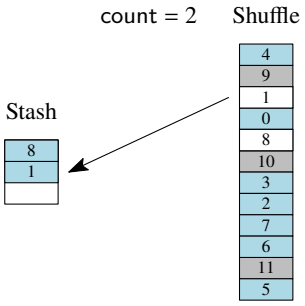
Sorting Networks The scheme uses a comparison-based *sorting network*, which is a circuit that consists only of comparison gates and is able to sort an array. It has the key property that the structure of the circuit is independent of the inputs. Because of this key property, a sorting network can be used for *oblivious sorting*. Practical sorting networks can be built with a circuit of size $O(n \log^2 n)$, such as *bitonic sort* (Batcher [3]). Optimal solutions matching the lower bound of $\Omega(n \log n)$ [10, 28] exist [1], but all of these have huge constants, preventing their use in practice.



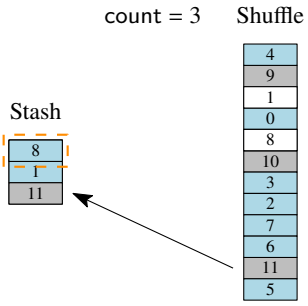
(a) Initial shuffling according to a randomly sampled permutation π. The label of each entry is its logical index.



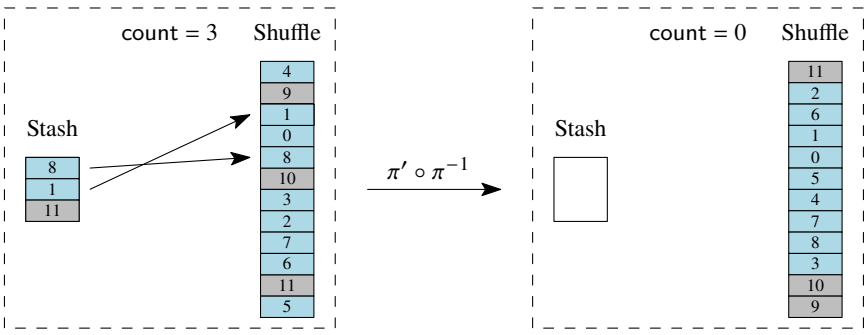
(b) Access logical index 8 at physical index $\pi(8) = 4$. The updated entry is stored in the stash.



(c) Access logical index 1 at physical index $\pi(1) = 2$. The updated entry is stored in the stash.



(d) Access logical index 8 which is already stored in the stash (marked with the orange dashed box). The dummy entry with logical index $n-1+\mathsf{count} = 11$ is fetched at physical index $\pi(11) = 10$ in the shuffle and stored in the stash.



(e) $\mathsf{count} = T$. Reshuffling according to $\pi' \circ \pi^{-1}$ is carried out for a freshly sampled, random permutation π', and updated entries from the stash are written to the shuffle during this step (indicated by the stash-to-shuffle arrows).

Figure 4.1: An example of a square-root ORAM cycle and initialization. The ORAM size is $n = 9$, and the number of accesses per cycle is $T = \sqrt{n} = 3$.

Specification The detailed square-root ORAM protocol is specified as follows:

Square-Root ORAM

Initialize Upon the input of n entries, indexed from 0 to $n-1$, and a security parameter λ, the client lets $T = \Theta(\sqrt{n})$ be the reshuffling period and carries out the following steps:

1. Append T *dummy entries*, indexed from n to $n+T-1$, to the input entries.
2. Sample and store a uniformly random permutation $\pi\colon [n+T]_0 \to [n+T]_0$ and permute all entries (including the dummy entries) according to π. To store π using sublinear client space, it can be represented as a *pseudorandom function* (PRF) parametrized by λ.
3. Initialize the stash as an empty array of size T and a counter count that initially holds the value 0.
4. Send the permuted entries to the server.

Access Upon the input of a logical index $i \in [n]_0$, and an operation Φ, the client carries out the following steps:

1. Let $\mathsf{count} = \mathsf{count} + 1$.
2. Perform a linear scan of the stash to find an entry with logical index i (we assume that each entry stores its own logical index).
   - If an entry is found, execute Φ on the found entry. Then, request the entry with physical index $\pi(n-1+\mathsf{count})$ from the server's shuffle and save it in the stash.
   - Otherwise, request the entry with physical index $\pi(i)$ from the server's shuffle, execute Φ on it, and save it in the stash.
3. If $\mathsf{count} = T$, do the following to reshuffle:
   (a) Sample a new uniformly random permutation π', and sort the server's shuffle according to $\pi' \circ \pi^{-1}$ by jointly evaluating a sorting network gate by gate with the server: for each gate, fetch the two gate input entries from the server, evaluate the gate, overwrite entries with stash data if a newer version of the entry is stored in the stash, and write the outputs back to the server. Let $\pi = \pi'$.
   (b) Let $\mathsf{count} = 0$ and reinitialize the stash as an empty array of size T.

Protocol 4.1: The Square-root ORAM protocol.

Correctness and Obliviousness Clearly, the square-root ORAM scheme satisfies Definition 4.3 with respect to $\mathcal{F}_{\mathsf{RAM}}$. Since the permutation is chosen uniformly at random and is always unknown to the server, and since the reshuffling is oblivious and is always carried out before the stash overflows and the shuffle runs out of unaccessed dummy entries, there is no association between the logical and physical indices being accessed at any time, so the square-root ORAM scheme also satisfies Definition 4.8 (in a computational sense if representing π as a PRF).

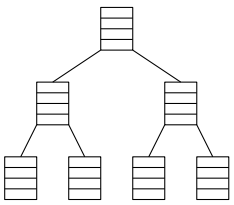
Efficiency Assuming that the PRF representation is sufficiently small, and that the oblivious sorting is performed by the client and server jointly evaluating a comparison-based sorting network gate by gate, accessing entries in a streaming fashion, the client-side storage is dominated by the size of the stash, i.e. $O(\sqrt{n})$. The server storage is $n + T + O(1)$ where the last term ensures that the server has available registers for executing a sorting network with the client. To compute the amortized access bandwidth per entry, we divide the total cost of T accesses by T, which gives us a per-entry bandwidth of

$$T^{-1}(\Theta(T) + \Theta(n \log^2 n)) = \Theta(1) + \Theta(\sqrt{n} \log^2 n)$$

entries sent between the client and server when using a $\Theta(n \log^2 n)$ sorting network (since the circuit size translates directly into asymptotic bandwidth), which gives an amortized cost per access of $\Theta(\sqrt{n} \log^2 n)$ for $T = \Theta(\sqrt{n})$. Note, however, that the *worst-case* access bandwidth is superlinear because of the reshuffling.

4.2.2 Tree ORAM

The *tree ORAM* scheme was first introduced by Shi et al. [34] in 2011. In this scheme, the data is stored in a complete binary tree where every node is called a *bucket* and is able to store a fixed number β of entries. A bucket is always full in the sense that it is padded with sufficiently many *dummy entries*. Only the client can distinguish dummy entries from non-dummy entries, and as we will see, the data access pattern does not reveal to the server which entries are dummy and which are not. One can think of a bucket as a trivial ORAM by itself, i.e. to access an entry in a bucket, the entire bucket is scanned linearly (in theory, a bucket can be implemented using any type of ORAM, but in practice, buckets are almost always small enough that trivial ORAM is the most efficient implementation). When we say that we "add" or "remove" an entry to or from a bucket, it means that we replace a dummy entry by a real entry or vice versa. The tree has $\Theta(n)$ leaves, and each leaf is assigned a unique label between 0 and $O(n)$. The main invariant that is maintained in tree ORAM is the following:



Invariant 4.12 (Tree ORAM)**.** At any point in time, every entry in the tree is associated with exactly one leaf label and resides in a bucket somewhere on the path from the root to that leaf. △

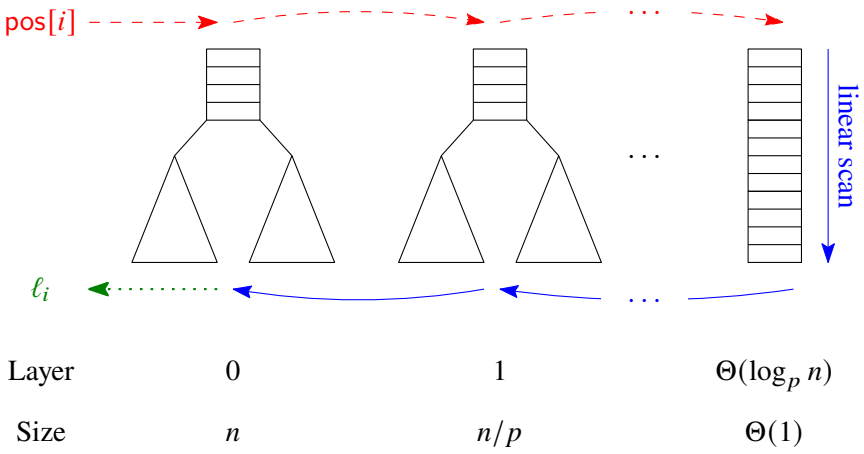


Figure 4.2: A diagram that shows the flow of looking up the leaf label ℓ_i of index i in the recursive position map pos of a tree ORAM. The recursive request flow (the red dashed arrows) starts with the statement $\mathsf{pos}[i]$ and traverses all recursive layers, starting with layer 0. When the base layer is reached, the actual workflow (the blue solid arrows) begins by linearly scanning the constant-size, trivial ORAM and continues by performing one tree ORAM lookup per recursive layer until we are back at layer 0 where ℓ_i can finally be returned (the green dotted arrow).

The tree ORAM furthermore maintains a *position map* that translates logical indices to leaf labels. The position map is stored recursively in ORAM layers of decreasing size on the server (except for the client-side, constant-size base layer) in order to minimize client-side storage. For each step in the recursion, indices are packed together by a factor of $p = \Theta(1)$ which means that the recursive structure ends up having $\Theta(\log_p n)$ layers. Refer to Fig. 4.2. In line with Shi et al. [34], we consider a slightly different, but equivalent, ORAM interface with two operations: ReadAndRemove and Add. Whenever an entry is to be read by the client, the client looks up the associated leaf label in the position map, and scans through the whole path ($\Theta(\log n)$ buckets) from the root to that leaf, reading and writing the entries stored by the server in a streaming fashion to keep the client-side storage minimal. When the entry in question is found during this scan, it is stored in client-side memory and overwritten with a dummy entry in the bucket. If the client wants to add an entry to the ORAM, the client must locate a dummy entry in the root bucket and overwrite it with the entry to be added, assigning it a uniformly random leaf label. Next, an *eviction* procedure is carried out, where some entries are *pushed down* towards the leaves in order to lower the risk of bucket overflow. The details of how this eviction is performed, the bucket size β, the number of leaves, and whether the structure is augmented with a client-side stash as in square-root ORAM are ways in which various tree ORAM schemes differ. We now formally specify and analyze the basic tree ORAM scheme, before we present some important variants that we will be working with in the thesis.

Specification For a tree ORAM of size n, we always assume that its buckets store atomic units called *blocks* of size $b = \Omega(\log n)$ bits. Specifically, we assume that a block is large enough that it can store an entire entry consisting of data, index, and leaf label. The detailed protocol is specified as follows:

Tree ORAM

Initialize On input n entries, indexed from 0 to $n - 1$, the client carries out the following steps:

1. Sample a random permutation π on $[n]_0$ and use it to assign a unique leaf label to every entry. Store π as pos in recursive ORAM where indices in each layer are packed together by factor $p = \Theta(1)$. The client only stores the constant-size base level, while all other levels are stored on the server.
2. Store one entry in each leaf bucket at the server, according to pos.

ReadAndRemove On input logical index $i \in [n]_0$, the client carries out the following steps:

1. Look up the leaf label $\ell_i = \mathsf{pos}[i]$.
2. Scan the tree path from the root to the leaf ℓ_i. When the entry with logical index i is found, store it in client memory and overwrite it with a dummy entry.

Add On input logical index i and data data, the client carries out the following steps:

1. Sample a uniformly random leaf label ℓ^* and let $\mathsf{pos}[i] = \ell^*$.
2. Write $(i, \mathsf{data}, \ell^*)$ to an empty block (i.e. a block containing a dummy entry) in the root bucket. If the root bucket is full, abort and output $\perp$.
3. Perform *eviction*:
   (a) Let v denote the *eviction rate*. On each level in the tree, except for the leaf level, choose v buckets uniformly at random.
   (b) For each chosen bucket $\mathcal{B}$, try to remove a random real entry from $\mathcal{B}$ and add it to the left or the right child bucket of $\mathcal{B}$, maintaining Invariant 4.12. Add a dummy entry to the other child bucket. If $\mathcal{B}$ was empty, dummy entries are instead added to both child buckets. The two add operations should always be carried out in a deterministic order to ensure obliviousness.

Protocol 4.2: The Tree ORAM protocol.

Correctness and Obliviousness Correctness of Protocol 4.2 follows from the following lemma about overflow probability which is proved by Shi et al. [34]:

Lemma 4.13 (Bucket Overflow)**.** *Suppose* $0 < \delta < 1$ *and* $n, m \geq 10$*. Then one can instantiate tree ORAM with buckets of capacity* $O(\log \frac{nm}{\delta})$ *such that with probability at least* $1 - \delta$*, it supports* m *operations without any bucket overflow.*

The lemma implies that we can obtain a negligible error probability $\delta = n^{-\Theta(1)}$ for a bucket capacity of $\beta = \Theta(\log n)$ if the number of operations is $m = n^{\Theta(1)}$. Obliviousness follows from the fact that entries are always assigned random leaf labels, so reading and removing an entry induces a tree path access that is independent of the index to be accessed. Furthermore, adding an entry always induces an access to the root node. Finally, eviction chooses v uniformly random buckets on each level and always writes to both children of every bucket. Assuming no overflow happens, the physical access pattern is therefore completely independent of any logical indices that are accessed. Hence, tree ORAM is statistically oblivious.

Efficiency By storing the position map server-side using recursive ORAM and having the client access tree paths in a streaming fashion, it is possible to achieve $O(1)$ client-side storage since all used client-side storage then becomes transient. Using a recursive *pack factor* of $p = \Theta(1)$, we end up with a position map that has $\lceil \log_p n \rceil$ recursive levels, where each level $j \in \{0, 1, \ldots, \lceil \log_p n \rceil - 1\}$ is a binary tree with $\Theta(\frac{n}{p^j})$ buckets, each bucket of size $\beta_j = \Theta(\log \frac{n}{p^j})$. In total, this results in

$$\Theta\left(\sum_{j=1}^{\lceil \log_p n \rceil} \frac{n \cdot \beta_j}{p^j}\right) = \Theta(n \log n)$$

position map entries stored on the server. This is also the asymptotic size of the tree storing the data; assuming that $n = 2^k$ for some $k \in \mathbb{N}_0$, the server stores exactly $2n - 1 = \Theta(n)$ buckets of capacity $\beta = \Theta(\log n)$, resulting in a total server storage of $\Theta(n \log n)$ blocks. The operational bandwidth is computed as follows: in the recursive position map, the bucket size of the tree on level j is β_j which also happens to be the number of buckets read in the tree on level j. Hence, the total number of entries read in all levels during a lookup is

$$\Theta\left(\sum_{j=1}^{\lceil \log_p n \rceil} \beta_j^2\right) = \Theta(\log^3 n).$$

Thus, the *worst-case and amortized* bit bandwidth for both ReadAndRemove and Add for a block size of $b = \Theta(\log n)$ bits becomes

$$\Theta(\underbrace{b \log^3 n}_{\substack{\text{recursive} \\ \text{lookup}}} + \underbrace{b\beta \log n}_{\substack{\text{data} \\ \text{lookup}}}) = \Theta(\log n \cdot (\log^3 n + \log^2 n)) = \Theta(\log^4 n).$$

Path ORAM

Following the introduction of tree ORAM, a line of research has aimed to improve the basic construction in various ways. Stefanov et al. [35] introduced Path ORAM, which is a variant that achieves a worst-case access bandwidth of $O(\log^3 n)$ bits, a server storage of $O(n)$ blocks, and a *superpolynomial* failure probability of $n^{-\omega(1)}$ for $m = n^{\Theta(1)}$ operations, by using $\hat{O}(\log^3 n)$[1] bits of client-side storage and assuming a larger block size of $b = \Omega(\log^2 n)$ bits, while preserving statistical obliviousness. Path ORAM differs from basic tree ORAM in the following two major ways:

Constant bucket size and client-side stash The bucket size is $\beta = \Theta(1)$. However, the client uses an $\hat{O}(\log n)$ stash to cache entries overflowing from the smaller buckets. By treating the stash as a special *depth-0 bucket* that is the imaginary parent of the root bucket, we can still require Invariant 4.12 to hold in the description of Path ORAM and other tree ORAM schemes that use a stash.

Eviction Path ORAM only specifies an Access operation. Whenever an entry residing on some path is accessed, *greedy* eviction is performed on that path, meaning that entries residing in a bucket on the path (including entries in the stash) are pushed down towards the leaf as far as possible, respecting Invariant 4.12.

Circuit ORAM

Circuit ORAM was introduced by Wang, Chan, and Shi [37]. They aim to improve the performance of Path ORAM when used in an MPC setting. As they point out, the problem with Path ORAM when used in MPC is its complex eviction, which results in a circuit of size $\Theta(b \log^2 n)$. The authors note that the majority of the computation in MPC is spent evaluating circuits, and thus the *circuit complexity* of ORAM in MPC becomes the most important metric. The ORAM circuit complexity translates into the circuit size of the *ORAM client computation* since this is the private computation that is evaluated as a circuit. Assuming a block size $b = \Omega(\log^2 n)$, Circuit ORAM achieves a circuit size of $\hat{O}(b \log n)$ gates for a *polynomial* failure probability negligible in n. This is almost optimal since, as the authors argue, the logarithmic ORAM lower bound applies directly to the circuit size. Like Path ORAM, Circuit ORAM is statistically oblivious. The bucket size is still $\beta = \Theta(1)$. The main ways in which Circuit ORAM differs from Path ORAM are the following:

Server-side stash Circuit ORAM continues to use the stash terminology introduced in Path ORAM, but the $\hat{O}(\log n)$-sized stash is stored server-side and is operated on obliviously by the client, resulting in $O(1)$ blocks of client-side storage.

[1] The $\hat{O}(f(n))$ notation means that for any function $h(n) = \omega(1)$, it holds that $g(n) = O(f(n)h(n))$

Eviction Operations are the same as in Path ORAM, except for eviction; first of all, eviction is performed along *two* paths that are non-overlapping, except for in the root of the tree. Choosing these paths follows either a *uniformly random* strategy, or a *deterministic round robin* strategy, which evaluation suggests yields a better practical performance. Secondly, the eviction is, like Path ORAM, the greediest possible, but subject to *limiting the number of scans of the path to two metadata scans, followed by one data scan*. The metadata scans act as a *lookahead mechanism* such that it is possible to decide which entries to "pick up" in order to move them to a deeper position on the path during the real data scan.

4.3 Oblivious Priority Queues

In addition to ORAM, it is also interesting to study other, more specialized oblivious data structures as these may be better suited for certain applications. Of course, any oblivious data structure in the RAM model can be built on top of ORAM. However, in many cases, a more sophisticated implementation can yield a significantly better performance. This turns out to be the case for the *oblivious priority queue* (OPQ) data structure. We now formally specify a version of the general priority queue data structure as an ideal functionality and present known lower bounds for the oblivious variant before for we discuss the important application of *oblivious sorting* which is an interesting algorithmic primitive in and of itself, but is also an important building block in various ORAM schemes. Then, we proceed to present *Path Oblivious Heap*: a practical and optimal OPQ introduced by Shi [33] that comes in two variants which build on the ideas of Path ORAM and Circuit ORAM respectively. We argue that POH is correct and oblivious, and finally, we observe that POH immediately implies *optimal oblivious sorting*.

Specification A priority queue, as defined by Shi [33], must support the operations specified in the following functionality:

Functionality $\mathcal{F}_{\mathsf{PQ}}$

Initialize On input (Init), let $Q = \emptyset$ and let $\mathsf{count} = 0$.

Insert On input (Insert, k, v) where k is a *key* and v is a *value*, let $\mathsf{ref} = \mathsf{count}$, let $Q = Q \cup \{(k, v, \mathsf{ref})\}$, let $\mathsf{count} = \mathsf{count} + 1$, and return ref.

FindMin On input $(\mathsf{FindMin})$, if $Q = \emptyset$, return $\perp$. Otherwise, find and return the entry $(k, v, \mathsf{ref}) \in Q$ where k is minimal.

ExtractMin On input $(\mathsf{ExtractMin})$, if $Q = \emptyset$, return $\perp$. Otherwise, find, remove, and return the entry $(k, v, \mathsf{ref}) \in Q$ where k is minimal.

Functionality 4.2: The ideal priority queue functionality.

Note that $\mathcal{F}_{\mathsf{PQ}}$ assigns references by *timestamping* to ensure uniqueness. A specific priority queue implementation may also support more operations such as Extract, IncreaseKey, and DecreaseKey:

Extract On input $(\mathsf{Extract}, \mathsf{ref})$, find the entry $(k, v, \mathsf{ref}) \in Q$ and let $Q = Q \setminus \{(k, v, \mathsf{ref})\}$. If there is no such entry, return $\perp$. Otherwise, return (k, v).

IncreaseKey On input $(\mathsf{IncreaseKey}, \mathsf{ref}, \Delta)$, for a positive amount Δ, find the entry $(k, v, \mathsf{ref}) \in Q$ and let $Q = (Q \setminus \{(k, v, \mathsf{ref})\}) \cup \{(k + \Delta, v, \mathsf{ref})\}$. If there is no such entry, return $\perp$.

DecreaseKey Same as IncreaseKey, but Δ is a negative amount instead.

Lower Bounds Jacob, Larsen, and Nielsen [20] recently proved that there exists a sequence of n operations yielding an amortized $\Omega(\log n)$ lower bound on the bandwidth per operation for an oblivious priority queue that stores up to n entries, even when allowing $O(n^\epsilon)$ client storage for an arbitrary $0 < \epsilon < 1$. This is only a constant factor less efficient than the widely used insecure *binary heap*. However, the authors note that there does exist asymptotically faster insecure priority queue implementations, the best ones achieving amortized $O(1)$ time per operation assuming a block size of $b = \Theta(\log n)$ bits and an entry size of $w = \Theta(b)$. For $b, w = \omega(\log n)$ and $w = \Theta(b)$, the best known implementation achieves expected, amortized $O(\sqrt{\log \log n})$ time per operation. As we shall soon see, the *Path Oblivious Heap* OPQ is optimal in terms of the $\Omega(\log n)$ oblivious lower bound, and hence it is a $\Theta(\log n / \sqrt{\log \log n})$ or $\Theta(\log n)$ factor slower than the best known insecure implementation, depending on the block size assumption.

Application: Oblivious Sorting It is well-known that sorting is an important algorithmic primitive. Similarly, *oblivious sorting* is an important algorithmic primitive that we have already seen in use in this thesis in the square-root ORAM scheme. Using sorting networks, one can achieve practical $O(n \log^2 n)$ oblivious sorting on n entries. $O(n \log n)$ solutions have also existed since the 1980s [1], but these solutions have notoriously suffered from large constants, rendering them impractical. However, an optimal, practical OPQ immediately implies practical and optimal $O(n \log n)$ oblivious sorting, as noted by Shi [33]. This can be achieved by first inserting the n entries into the OPQ and then calling ExtractMin n times, resulting in the entries being extracted in sorted order in time $O(n \log n)$, assuming an $O(\log n)$ bound on the OPQ operations.

4.3.1 Path Oblivious Heap

Path Oblivious Heap (POH), introduced by Shi [33], is an optimal and practical OPQ implementation. It is based on the same structure as tree ORAM – however, due to the nature of the operations that need to be supported, it is not necessary to maintain a position map of every entry and store it recursively in ORAM. Instead, each bucket

$\mathcal{B}$ in the tree is augmented with a field $\mathcal{B}.\mathsf{Min}$ that stores the entry (k, v, ref) with the current minimal key that resides somewhere in the subtree rooted at $\mathcal{B}$ (called the *subtree-min*). References are on the form $\mathsf{ref} = (\ell, \mathsf{id})$ where ℓ is a leaf label and id is a globally unique identifier. Ties when comparing keys are broken by the unique identifiers. As in tree ORAM, Invariant 4.12 is maintained throughout the lifetime of the data structure. More specifically, two variants of POH are introduced: the *Path variant* and the *Circuit variant*. The former is based on Path ORAM while that latter is based on Circuit ORAM. We will present the Path variant while noting that the Circuit variant is identical, except for the inner workings of the eviction procedure and the fact that it uses server-side stash storage.

Specification The detailed POH protocol is specified as follows:

Path Oblivious Heap

Initialize Upon the input of a max capacity n and a security parameter λ, the server initializes an empty Path ORAM structure of size n without a recursive position map. Initially, the client fills every bucket to its max capacity $\beta = \Theta(1)$ (or $\omega(\log \lambda)$ for the client-side stash) with dummy entries on the form $(\mathsf{ref} = \bot, k = \infty, v = \bot)$. To support an unbounded number of queries, an *unconsumed identifier stack* S is maintained, instead of ensuring reference uniqueness by timestamping, since the size of the timestamps would keep increasing infinitely. Initially, $\mathsf{S} = [n]_0$, and references are popped from S on insertion and pushed to S on extraction.

Insert To insert (k, v) into the queue, the client does the following:

1. Choose a random leaf label $\ell^* \in_R [n]_0$ and pop an unused identifier id from S.
2. Add $(k, v, (\ell^*, \mathsf{id}))$ to the root bucket.
3. Choose two random paths $\mathcal{P}$ and $\mathcal{P}'$ that are non-overlapping, except in the root, and evict along those paths.
4. Call $\mathsf{UpdateMin}(\mathcal{P})$ and $\mathsf{UpdateMin}(\mathcal{P}')$.
5. Return $\mathsf{ref} = (\ell^*, \mathsf{id})$.

Extract To extract the entry identified by $\mathsf{ref} = (\ell, \mathsf{id})$, the client does the following:

1. Retrieve the path $\mathcal{P}$ identified by ℓ.
2. Scan through the buckets in $\mathcal{P}$ and remove the unique entry (k, v, ref) from $\mathcal{P}$.

3. Evict along $\mathcal{P}$.
4. Call $\mathsf{UpdateMin}(\mathcal{P})$.
5. Push ref to S.
6. Return (k, v, ref).

FindMin The client fetches the root bucket $\mathcal{B}_{\text{root}}$ and returns $\mathcal{B}_{\text{root}}.\mathsf{Min}$.

ExtractMin The client calls $(k, v, \mathsf{ref}) \leftarrow \mathsf{FindMin}$ followed by $(k, v) \leftarrow \mathsf{Extract}(\mathsf{ref})$ and returns (k, v) (or $\perp$ if any operation fails).

Protocol 4.3: The Path Oblivious Heap OPQ protocol.

For details about eviction and bucket operations, we refer to the descriptions of the different tree ORAM schemes. Note, however, that regardless of the variant implemented, eviction is performed along two random, non-overlapping paths on insertion and along the read path on extraction. UpdateMin is implemented as follows:

$\mathsf{UpdateMin}(\mathcal{P})$ Upon the input of a tree path $\mathcal{P}$, which is a leaf-to-root sequence of buckets, do the following for each bucket $\mathcal{B}$ in $\mathcal{P}$:

1. Let $\mathcal{B}_L$ and $\mathcal{B}_R$ denote the two children of $\mathcal{B}$.
2. Let m be the current entry in $\mathcal{B}$ with the smallest key, found by linear scan.
3. Let $\mathcal{B}.\mathsf{Min} = \min(m, \mathcal{B}_L.\mathsf{Min}, \mathcal{B}_R.\mathsf{Min})$.

For the degenerate case where $\mathcal{B}$ is a leaf bucket, we just let $\mathcal{B}.\mathsf{Min} = m$ in step 3. It is easy to see that the bandwidth of this operation is proportional to the length of $\mathcal{P}$.

Correctness and Obliviousness Shi [33] proves that when instantiated with a security parameter λ, and for $n = \text{poly}(\lambda)$ capacity, if the stash has capacity $\omega(\log \lambda)$, then Protocol 4.3 satisfies correctness with respect to $\mathcal{F}_{\mathsf{PQ}}$ with error probability negligible in λ. The proof follows by reduction to Path ORAM correctness. Furthermore, Protocol 4.3 satisfies statistical obliviousness which is also proved in the original paper. The proof considers the generated access patterns of an imaginary priority queue $\mathcal{PQ}^{\infty}$, which is identical to POH except that it has infinite bucket capacity, and defines a perfectly oblivious simulation in a straightforward way by letting the simulator sample and output uniformly random leaf labels upon the inputs of Insert and Extract, and by letting it output $\emptyset$ upon the input of FindMin, such that the output of the simulator is distributed identically to the output of $\mathcal{PQ}^{\infty}$. They finally combine this simulation with the proven correctness, and note that there is a one-to-one correspondence between the generated access patterns of $\mathcal{PQ}^{\infty}$ and Protocol 4.3, in order to conclude that the protocol is statistically oblivious.

Efficiency Let the block size be $b = \Omega(\log n)$ bits and assume an entry fits in $\Theta(1)$ blocks. Furthermore, let the stash size be $s = \alpha(\lambda) \log \lambda$ for an arbitrarily small, super-constant function $\alpha(\lambda)$. Then the bandwidth of Insert, Extract, and ExtractMin is proportional to the height of the tree, i.e. $\Theta(b \log n)$ bits. FindMin has a bandwidth complexity of $\Theta(b)$ bits, or even zero bandwidth if the stash min is stored client-side. The client storage is $O(bs)$ bits. The Circuit variant achieves $O(b)$ bits of client storage at the cost of $\Theta(b \cdot (\log(n) + s))$ bits of bandwidth for Insert, Extract, and ExtractMin.

Path Oblivious Sort The $\Theta(n \log n)$ oblivious sorting algorithm immediately implied by POH is also briefly outlined by Shi [33] and is called *Path Oblivious Sort*.

Chapter 5

Oblivious Data Structures in MPC

In the quest towards enabling practical, generic MPC, it has proven useful to utilize oblivious data structures in order to reduce the MPC circuit complexity of functions with input-dependent memory access which we have seen yield large (at least linear) circuit sizes when implemented naïvely as circuits – even when their computational complexity in the RAM model is sublinear. The general idea is to combine the two concepts by implementing an oblivious data structure *inside an MPC protocol*. This has been done in several ways in the literature:

- Gentry et al. [12] and Gordon et al. [16] present an asymmetric *client-server* 2PC ORAM approach where the client state is secret shared between the two parties while one party stores the ORAM server state. ORAM client computation is carried out as an MPC circuit, and the requests to the ORAM server are revealed to the MPC party holding the server state, who inputs the requested data into the secure computation. By obliviousness of ORAM, revealing the physical addresses to be accessed does not compromise security. It should be clear that this approach does not depend on any specifics of ORAM, but can be used with other oblivious data structures as well.

- The approach that has been developed most in recent years [25, 33, 37, 42] works by secret-sharing the entire data structure state (client *and* server) between all parties taking part in the secure computation, and does thus not rely on a client-server setup. We shall focus on this approach in the following and call it *ODS-MPC*.

In ODS-MPC, the parties initially secret-share the data structure input. Then, they evaluate the initialization phase as an MPC circuit in order to end up with secret shares of the client and server state of the initialized data structure. In a more complex program, a shared state is initialized for each data structure that is being used. As for the initialization, all data structure operations are carried out in MPC on secret-shared inputs. For instance, when accessing an ORAM entry by index, the index is secret shared and used in the secure computation – and it remains secret due to the ORAM obliviousness property. Since the server state is secret shared,

one benefit of the ODS-MPC approach is that the secret-shared state need not be encrypted, because privacy is guaranteed by definition of secret sharing. Another advantage of ODS-MPC is that it immediately generalizes to an arbitrary number of parties, provided the underlying MPC protocol generalizes.

In the ODS-MPC setting, *circuit complexity* becomes the most important performance metric. The goal is to improve this complexity, both asymptotically and by shaving off constants, such that it outperforms a naïve MPC circuit with a linear input-dependent access complexity.

It is important to note that building a practically efficient ODS in MPC is not as simple as taking an efficient MPC protocol and combining it with an asymptotically efficient ODS. Since the data structure is to be implemented in a (Boolean or arithmetic) circuit, certain types of operations turn out to be prohibitively expensive in terms of circuit size due to high constants, e.g. evaluating pseudorandom functions or hash functions. Thus, non-trivial modifications often need to be made to existing oblivious data structures in order to tune them to perform well for practical parameter ranges when implemented in the ODS-MPC setting.

In this chapter, we develop a pseudocode framework for specifying ODS implementations in MPC, and then we show how to use the framework by presenting an implementation of square-root ORAM by Zahur et al. [42] which uses garbled circuits. In the following chapter, we finally build upon this framework to present our own implementation of a Path Oblivious Heap.

5.1 A Framework for Specifying ODS-MPC

When specifying an ODS-MPC implementation, we assume that we are given access to an arbitrary MPC protocol which implements the arithmetic black box functionality (Functionality 3.3 on page 27). For specifying oblivious data structures in MPC, we use a notation similar to the one used by Zahur et al. [42] which we will now describe in detail for completeness. Recall that the $\langle \cdot \rangle$ notation denotes a hidden (secret-shared) value or an oblivious statement (see Notation 2.1 on page 7).

Notation 5.1 (Entries)**.** We assume that a data block is able to hold a minimum of $b = \Omega(\log n)$ bits of data and metadata, allowing a block to store a complete data structure entry. For an entry, entry, we can access its data via $\mathsf{entry}.\langle \mathsf{data} \rangle$. We also assume that an entry always stores an index of size $\Theta(\log n)$ (this applies to all our use cases, although an index is not generally part of an entry), accessed via $\mathsf{entry}.\langle \mathsf{index} \rangle$. The index may be the logical RAM index of the entry, or a leaf index if the entries are stored in a tree structure. Finally, it will be convenient to store an *empty bit*, $\mathsf{entry}.\langle \mathsf{empty} \rangle$, to denote whether the entry is empty or not. When working with a sequence of entries, $\mathsf{entries}$, we can access the ith entry using the notation $\mathsf{entries}_i$. △

An oblivious data structure must expose an Initialize algorithm (Algorithm 5.1) that accepts relevant parameters such as size/capacity (n), and possibly a security parameter (λ) and some initial input. Initialize need not accept initial input however; if this is not the case, the data structure is initialized with empty entries.

```
1: interface Initialize(n, λ, params)
2:     // Initialize and return the oblivious data structure of size n with security
   parameter λ, using params if supplied.
```

Algorithm 5.1: The Initialize interface.

Example 5.2 (Trivial ORAM)**.** A trivial, but inefficient, solution to building an ORAM scheme is to implement it as an array of size n and always touch the full contents of the array for every access operation by carrying out a linear scan, executing the specified operation on the desired entry only. The scheme is specified by the following two algorithms:

```
1: function Initialize(n, entries)
2:     Oram ← (n, entries)
3:     return Oram
```

Algorithm 5.2: A trivial, linear Initialize implementation, accepting initial entries of size n.

```
1: function Access(Oram, ⟨i⟩, Φ)
2:     for j from 0 to Oram.n − 1 do
3:         ⟨if⟩ ⟨i⟩ = j then
4:             Φ(Oram.entries_j)
```

Algorithm 5.3: A trivial, linear Access implementation. Note that since ⟨**if**⟩ is an oblivious statement, every entry is touched but Φ is only executed on one entry.

It is clear that this solution satisfies Definitions 4.3 and 4.8 – it is easy to see that the scheme is even perfectly oblivious, provided the underlying secret sharing is perfectly secure. However, it is very inefficient ($\Omega(n)$ access cost!), Thus, more efficient solutions are needed in practice. ▲

Observation 5.3. (Implementing Read and Write given Access) Note that it is a straightforward task to implement Read and Write on top of Access from the above example:

```
1: function Read(Oram, ⟨i⟩)
2:     ⟨x⟩ ← ⊥
3:     function Φ(entry)
4:         ⟨x⟩ ← entry.⟨data⟩
5:     Access(Oram, ⟨i⟩, Φ)
6:     return ⟨x⟩
```

Algorithm 5.4: Implementation of the ORAM Read method using Access.

```
1: function Write(Oram, ⟨i⟩, ⟨x⟩)
2:     function Φ(entry)
3:         entry.⟨data⟩ ← ⟨x⟩
4:     Access(Oram, ⟨i⟩, Φ)
```

Algorithm 5.5: Implementation of the ORAM Write method using Access.

▲

Remark 5.4 (Low-level Implementation of Oblivious Statements). When implementing oblivious data structures in a modern MPC framework such as MP-SPDZ [22], which we will be using later in this thesis, high-level oblivious statements such as ⟨**if**⟩ are made directly available to the developer. However, translating these oblivious statements into circuits is non-trivial; assume we want to implement

$$\langle \textbf{if} \rangle \ \langle i \rangle = j \ \textbf{then} \ \Phi(\mathsf{Oram.entries}_j) \tag{5.1}$$

(lines 3-4 of Algorithm 5.3) as an arithmetic circuit. We only wish to execute Φ if $\langle i \rangle = j$. How to represent this as an arithmetic expression depends on how Φ is defined. First, consider the case of calling Access from Read (Algorithm 5.4). In this case, (5.1) can be rewritten as

$$\langle x \rangle \leftarrow (\langle i \rangle = j) \cdot \mathsf{Oram.entries}_j.\langle \mathsf{data} \rangle + (1 - (\langle i \rangle = j)) \cdot \langle x \rangle \, .$$

For the case of calling Access from Write (Algorithm 5.5), we can rewrite (5.1) as

$$\mathsf{Oram.entries}_j.\langle \mathsf{data} \rangle \leftarrow (\langle i \rangle = j) \cdot \langle x \rangle + (1 - (\langle i \rangle = j)) \cdot \mathsf{Oram.entries}_j.\langle \mathsf{data} \rangle \, .$$

Indeed, the $\langle i \rangle = j$ comparison is in itself high-level notation that requires rewriting as an arithmetic expression. We will assume that these widely used programmatic primitives are available to us in an oblivious sense and use high-level notation, unless the low-level details are specific and important to the data structure implementation with which we are concerned. ▲

5.2 Implementing Square-Root ORAM in MPC

Zahur et al. [42] present an adapted version of square-root ORAM (introduced in Section 4.2.1 on page 34) aimed at performing well in MPC – specifically, they focus on improving the practical performance for a range of values of n, providing benefits over a linear scan for as little as 8 blocks of data (depending on block size, underlying network, and processor), while noting that the solution is still asymptotically worse than the best known schemes, and thus, the performance gains compared to these schemes diminish as n increases. The modifications made to the original square-root ORAM scheme substantiate the claim made in the introduction of this chapter that it is non-trivial to tune an oblivious data structure to perform well in practice in an MPC setting.

5.2.1 Circuit Optimizations

Before presenting the ORAM scheme in pseudocode, we outline the modifications made to the original scheme:

No dummy entries Instead of appending T dummy entries to the shuffle and accessing one of them to perform a fake access, a real, untouched entry (in the current cycle) from the shuffle is accessed and added to the stash. Consequently, the shuffle is kept at size n instead of $n + T$.

Keeping track of used entries A public set Used, which keeps track of indices that have already been accessed, is maintained in each cycle to avoid oblivious sorting when moving entries from the stash to the shuffle during reshuffling. This is secure since the entries in Used have already been revealed to all parties.

Explicit position map Instead of using a PRF, the position map is represented explicitly using a specialized recursive ORAM structure, packing indices with a factor of $p = \Theta(1)$ per level, and terminating with a linear ORAM on the first level with less than T blocks. See Fig. 5.1 on page 53 for a visualization of the position map control flow. Using this map eliminates the need of evaluating a PRF for each access operation. In addition, using an explicit position map means that expensive $\Theta(n \log^2 n)$ oblivious sorting can be replaced by $\Theta(n \log n)$ oblivious permutation using a Waksman network [36] during initialization. Superficially, the recursive position map we have just described might appear to be similar to the ones used in tree ORAM schemes (see Fig. 4.2 on page 38). However, making this recursive structure efficient requires a careful implementation; as the authors argue, naïve usage of recursive ORAM would require $T + n$ position map lookups per T accesses (n lookups are needed to store the new permutation during reshuffling). If we let $n_i = n/p^i$ and $T_i = \sqrt{n_i \log n_i}$ denote respectively the size and reshuffling period of the recursive level i, when using a recursive pack factor of $p = \Theta(1)$,

the amortized cost of accessing level i of a naïve recursive position map would be $c_i = O(1)$ in the base case ($n_i = O(1)$), and in the recursive case, it would be

$$\begin{aligned} c_i &\geq \frac{n_i + T_i}{T_i} (c_{i+1} + p) \\ &> \frac{n_i}{\sqrt{n_i \log n_i}} c_{i+1} \\ &= \sqrt{\frac{n_i}{\log n_i}} c_{i+1} \\ &\geq n_i^{\frac{1}{4}} c_{i+1}. \end{aligned}$$

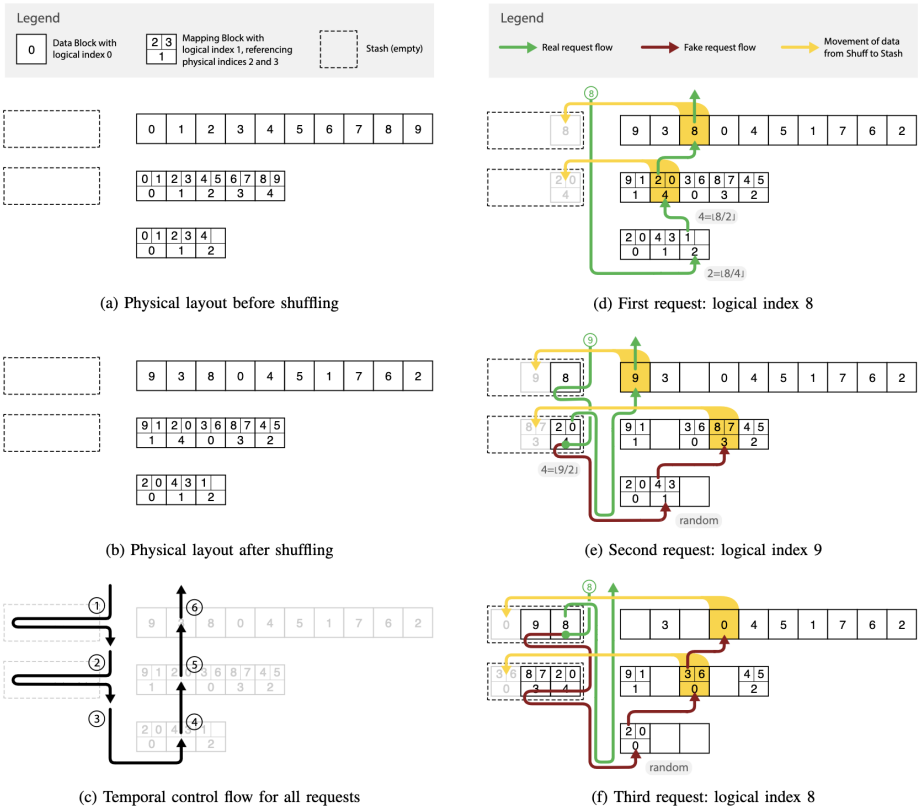
We see that c_0 expands to a product of $\log_p n$ factors, all of which are greater than 1. The first half of these factors are greater than $n^{\frac{1}{8}}$ so the product is

$$\begin{aligned} c_0 &> \left(n^{\frac{1}{8}}\right)^{\frac{1}{2}\log_p n} \\ &= n^{\frac{1}{16}\log_p n} \\ &= n^{\omega(1)} \end{aligned}$$

which is a superpolynomial function, and hence much too inefficient. In order to fix this issue, the recursive structure is re-initialized with the freshly sampled permutation on every reshuffling, which eliminates the need for n extra accesses. In addition, a global reshuffling period $T = \sqrt{n \log n}$ is fixed for every recursive level, enabling a much more efficient representation of Used in the recursive structure at the cost of suboptimal reshuffling for recursive levels: Used on the top-level is maintained in the clear, and a superset of Used is implicitly represented in the recursive structure by tracking which blocks on the deepest recursive level have been used during the current cycle. We maintain the following invariant:

Invariant 5.5 (Square-Root ORAM Recursive Used Representation)**.** If a block $\mathsf{block_d}$ on the deepest recursive level of the position map has been marked as used, it means that *at least one entry* on the top level, which is pointed towards by $\mathsf{block_d}$, must be in Used on the top level. Conversely, if $\mathsf{block_d}$ is not marked as used, we are certain that *no entry* on the top level, which is pointed towards by $\mathsf{block_d}$, can be in Used on the top level. △

This invariant makes it straightforward to sample an unused block during a dummy request to the deepest recursive level: since this level is linearly scanned, we can simply pick the first block that is not marked as used. Furthermore, we know that the smallest recursive level has size $\Theta(T)$, so if we fix T to be at most the number of blocks in this level, we are guaranteed that there will always be enough unused blocks available in this way, even in the extreme case where a cycle consists of T fake accesses.



(a) Physical layout before shuffling

(b) Physical layout after shuffling

(c) Temporal control flow for all requests

(d) First request: logical index 8

(e) Second request: logical index 9

(f) Third request: logical index 8

Figure 5.1: A visualization of the adapted square-root ORAM recursive position map control flow from the original paper [42, Fig. 7].

5.2.2 Specification

See Algorithms 5.6 and 5.7 below for a pseudocode specification of the adapted square-root ORAM scheme, as presented in the original paper [42]. We have omitted the detailed specification of InitializePosMap and GetPos since, in our opinion, the informal description of the recursive position map in Section 5.2.1 does a better job at providing intuition, and the Access algorithm is a sufficient example of using the oblivious pseudocode notation. Instead, we refer to the original paper for the pseudocode of these implementations.

```
1: function Initialize(entries)
2:     n ← |entries|
3:     ⟨π⟩ ← random permutation on n entries
4:     Shuffle ← ObliviousPermute(entries, ⟨π⟩)
5:     T ← ⌈√W(n)⌉
6:     Oram_1 ← InitializePosMap(⟨π⟩, 1, T)
7:     Oram_0 ← (n, t ← 0, T, Oram_1, Shuffle, Used ← ∅, Stash ← ∅)
8:     return Oram_0
```

Algorithm 5.6: The Initialize method of adapted square-root ORAM. $W(n) = \Theta(n \log n)$ is the cost of obliviously permuting n elements using a Waksman network.

```
1: function Access(Oram, ⟨i⟩, Φ)
2:     ⟨found⟩ ← false
3:     for j from 0 to Oram.t do
4:         ⟨if⟩ Oram_0.Stash_j.⟨index⟩ = ⟨i⟩ then
5:             ⟨found⟩ ← true
6:             Φ(Oram_0.Stash_j)
7:     p ← GetPos(Oram_0.Oram_1, ⟨i⟩, ⟨found⟩)
8:     ⟨if⟩ not ⟨found⟩ then
9:         Φ(Oram_0.Shuffle_p)
10:    append Oram_0.Shuffle_p to Oram_0.Stash
11:    Oram_0.Used ← Oram_0.Used ∪ {p}
12:    Oram_0.t ← Oram_0.t + 1
13:    if Oram_0.t = Oram_0.T then
14:        for j from 0 to Oram_0.T − 1 do
15:            p′ ← Oram_0.Used_j
16:            Oram_0.Shuffle_p′ ← Oram_0.Stash_j
17:        Oram_0 ← Initialize(Oram_0.Shuffle)
```

Algorithm 5.7: The Access method of adapted square-root ORAM

5.2.3 Analysis

We now present a detailed analysis of the efficiency of the Initialize and Access operations in the adapted square-root ORAM scheme:

Initialize The lines that dominate the asymptotic cost of Algorithm 5.6 are lines 4 and 6. In line 4, the cost of obliviously permuting n entries is $b \cdot W(n) = \Theta(bn \log n)$ bits. In line 6, we instantiate a recursive position map with $\Theta(\log n)$ ORAM levels, each level a factor of $p = \Theta(1)$ smaller than the previous level. Thus, we have the recurrence relation

$$c(n) = c\left(\frac{n}{p}\right) + bn \log n$$

which, just as in line 4, yields a total cost of $\Theta(bn \log n)$ bits. The complete cost of initialization is $\Theta(bn \log n)$ which also translates into the asymptotic circuit size.

Access To analyze the amortized cost of accessing an entry using Algorithm 5.7, we first compute the cost for a batch of T operations, and then we derive the amortized cost by dividing the batch cost by T. First, the stash is linearly scanned on lines 3-6 for each access, and the stash size t increases by 1 per access, which costs a total of

$$\sum_{t=0}^{T-1} bt = b \cdot \sum_{t=0}^{\sqrt{n \log n}} t = \Theta(bn \log n) \tag{5.2}$$

bits for T accesses. Next, on line 7, we call GetPos, which means that we perform a recursive position map lookup that amounts to scanning $\Theta(\log_p n)$ stashes of size t per access (corresponding to multiplying Eq. (5.2) by $\log_p n$), performing a linear scan of size $\Theta(T)$ of the deepest recursive level, and finally, performing $\Theta(\log_p n)$ lookups of indices. For T accesses, this costs

$$\underbrace{\Theta(bn \log^2 n)}_{\text{stash scans}} + bT \cdot (\underbrace{\Theta(T)}_{\substack{\text{linear} \\ \text{scan}}} + \underbrace{\Theta(\log n)}_{\substack{\text{recursive} \\ \text{index} \\ \text{lookup}}}) = \Theta(bn \log^2 n) \tag{5.3}$$

bits. Maintaining Used (line 11) is quite cheap; it only amounts to appending one index (in clear) per access, so we get $\Theta(bT)$ which is an insignificant contribution. Finally, reinitialization (lines 13-17) is dominated by the cost of calling Initialize on line 17, which we have already seen costs $\Theta(bn \log n)$ bits. Combining this with Eqs. (5.2) and (5.3), and dividing the total cost by T, we see that the amortized asymptotic access cost is dominated by the

$\Theta(\log n)$ stash scans, which have a cost of

$$\frac{\Theta(bn \log^2 n)}{T} = \frac{\Theta(bn \log^2 n)}{\Theta(\sqrt{n \log n})} = \Theta(b\sqrt{n \log^3 n})$$

bits in total. As for Initialize, this cost translates into the asymptotic circuit size.

We end the analysis by concluding that the adapted square-root ORAM scheme has taken non-trivial measures to avoid computations that yield large circuits in terms of constants, while keeping the asymptotic access cost a reasonable $\log n$ factor more expensive than the original square-root ORAM scheme.

5.2.4 Evaluation

Zahur et al. [42] implement their adapted square-root ORAM scheme as well as Circuit ORAM and Linear ORAM using the Obliv-C MPC framework (Zahur and Evans [41]), executing a garbled circuits protocol. They perform benchmarks showing that their scheme is indeed efficient (see Fig. 5.2): it has a faster access time than Linear ORAM and Circuit ORAM for block sizes ranging from 2^2 to 2^{12} bytes and n ranging from 2^4 to 2^{20} blocks (the exact crossover points depend on the block size). Linear ORAM is faster only for a small number of blocks, while Circuit ORAM outperforms the two other schemes asymptotically, starting from around 2^{13} or more blocks, depending on the block size.

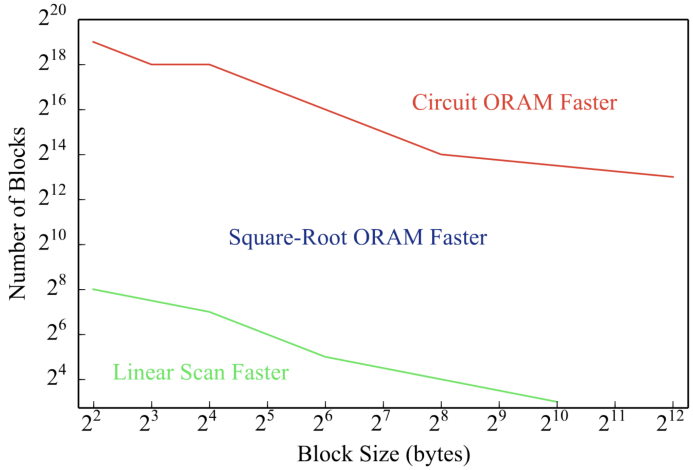


Figure 5.2: Per-access cost crossover between ORAM schemes (from the original paper [42, Fig. 8]).

Chapter 6

Path Oblivious Heap in SPDZ

In this chapter, we present a new implementation of the Path Oblivious Heap OPQ (described in Section 4.3.1 on page 43), as well as a variant which we call *Unique Path Oblivious Heap*, in the MP-SPDZ MPC framework by Keller [22]. First, we will give pseudocode and analyze the circuit complexity of our implementations, after which we present detailed benchmarks and compare the performance of our implementations with the performance of an OPQ by Keller and Scholl [25] which is a classic min-heap built on top of ORAM.

6.1 Implementation

Everything presented in this chapter has been implemented within MP-SPDZ and can be found in the official MP-SPDZ GitHub repository: https://github.com/data61/MP-SPDZ/pull/1014.

We have implemented the Path variant of POH based on an existing implementation of Path ORAM described by Keller and Scholl [25], removing all functionality related to maintaining the recursive position map which is not used in POH. We refer to this functionality as NonRecursivePathORAM and assume that it is available in our pseudocode specification. Next, we augmented the data structure with the capability to store and maintain subtree-mins of every node in the tree. We chose the Path variant over the Circuit variant since benchmarks by Keller and Yanai [26] indicate that the Path ORAM implementation in MP-SPDZ is more efficient than the Circuit ORAM implementation when used in conjunction with a SPDZ protocol. This is due to the fact that the Path ORAM implementation uses a more efficient eviction procedure than originally proposed, based on oblivious shuffling, as well as a constant-sized stash justified by an empirical bound for achieving a negligible failure probability [25].

We chose to implement a basic version of POH that only supports a simplified version of the operations specified in Functionality 4.2 on page 42. Specifically, our implementation does not return a reference with a unique ID upon inserting an entry, and hence does not support updating operations such as Extract, IncreaseKey,

and DecreaseKey. These operations are not needed for implementing oblivious sorting. However, in order to make a meaningful comparison with the min-heap OPQ implementation by Keller and Scholl [25], we also implemented a version, which we named Unique POH, that only allows inserting *unique* values between 0 and $n-1$. Essentially, Unique POH is an extension of our basic POH version that uses an internal ORAM of size n to maintain a map from values in the queue to their associated leaf indices. Thus, values in this version act as unique IDs, enabling the structure to support an efficient, generic Update operation like the min-heap OPQ, required for instance by Dijkstra's shortest path graph algorithm [8].

6.1.1 Specification

In this section, we give a pseudocode specification of POH and Unique POH.

Notation 6.1 (POH Entries)**.** For contextual clarity, we make a couple of modifications to Notation 5.1 on page 48 in the following: We assume that an entry entry stores entry.⟨key⟩ *and* entry.⟨value⟩ instead of entry.⟨data⟩. Furthermore, we assume that the index is accessed via entry.⟨leaf⟩ instead of entry.⟨index⟩. △

Path Oblivious Heap

The following algorithm shows how to initialize POH with different parameters:

```
1: function Initialize(n, λ, k, v, β, s)
2:     return NonRecursivePathORAM(n, λ, k, v, β, s)
```

Algorithm 6.1: The Initialize method of Path Oblivious Heap. k and v denote the bit lengths of keys resp. values, β denotes the bucket size, and s denotes the stash size.

This procedure showcases the conceptual simplicity of POH when having access to an underlying tree ORAM implementation, as initialization of the POH requires nothing more than initializing a non-recursive tree ORAM structure. Maintaining the minimum in each subtree is handled by UpdateMin, which updates the Min field of every bucket along a specified leaf-to-root path (recall that the stash is the imaginary parent of the tree root, and thus the actual root):

```
1: function UpdateMin(Queue, ℓ)
2:     for Bucket in PathFromLeafToRoot(Queue, ℓ) do
3:         minEntry ← FindBucketMin(Bucket)
4:         Bucket.Min ←
               Min(minEntry, Bucket.LeftChild.Min, Bucket.RightChild.Min)
```

Algorithm 6.2: The UpdateMin method of Path Oblivious Heap. FindBucketMin performs a linear scan of a bucket and returns the minimum entry. To handle degenerate loop cases, we define the tree root to be the left *and* right child of the stash, and we define every leaf to be its own left and right child.

UpdateMin relies on an implementation of oblivious entry comparison, which we implement in EnriesLt and EntriesEq:

```
1: function EntriesLt(entry, entry′)
2:     ⟨prioLt⟩ ← entry.⟨prio⟩ < entry′.⟨prio⟩
3:     ⟨prioEq⟩ ← entry.⟨prio⟩ = entry′.⟨prio⟩
4:     ⟨valueLt⟩ ← entry.⟨value⟩ < entry′.⟨value⟩
5:     return ¬entry.⟨empty⟩ ∧
                  (entry′.⟨empty⟩ ∨
                      (¬entry′.⟨empty⟩ ∧
                          (⟨prioLt⟩ ∨ (⟨prioEq⟩ ∧ ⟨valueLt⟩))))
```

Algorithm 6.3: The oblivious EntriesLt method of Path Oblivious Heap. Returns $\langle 1 \rangle$ if entry is strictly less than entry′ and $\langle 0 \rangle$ otherwise.

```
1: function EntriesEq(entry, entry′)
2:     ⟨bothEmpty⟩ ← entry.⟨empty⟩ ∧ entry′.⟨empty⟩
3:     ⟨emptyEq⟩ ← entry.⟨empty⟩ = entry′.⟨empty⟩
4:     ⟨prioEq⟩ ← entry.⟨prio⟩ = entry′.⟨prio⟩
5:     ⟨valueEq⟩ ← entry.⟨value⟩ = entry′.⟨value⟩
6:     return ⟨bothEmpty⟩ ∨ (⟨emptyEq⟩ ∧ ⟨prioEq⟩ ∧ ⟨valueEq⟩)
```

Algorithm 6.4: The oblivious EntriesEq method of Path Oblivious Heap. Returns $\langle 1 \rangle$ if the entries are equal and $\langle 0 \rangle$ otherwise.

Maintaining the subtree-mins in every node leads to the following straightforward implementation of FindMin which simply returns the current subtree-min of the stash:

```
1: function FindMin(Queue)
2:     return Queue.Stash.Min
```

Algorithm 6.5: The FindMin method of Path Oblivious Heap.

Finally, we conclude the specification by presenting Insert and ExtractMin:

```
1:  function Insert(Queue, ⟨k⟩, ⟨v⟩)
2:      entry ← (
            ⟨key⟩ ← ⟨k⟩ ,
            ⟨value⟩ ← ⟨v⟩ ,
            ⟨leaf⟩ ← RandomLeafLabel(Queue.n),
            ⟨empty⟩ ← ⟨0⟩
        )
3:      Queue.Stash.Add(entry)
4:      ℓ1 ← Reveal(RandomLeftSubtreeLeafLabel(Queue.n))
5:      ℓ2 ← Reveal(RandomRightSubtreeLeafLabel(Queue.n))
6:      EvictAlongPath(Queue, ℓ1)
7:      EvictAlongPath(Queue, ℓ2)
8:      UpdateMin(Queue, ℓ1)
9:      UpdateMin(Queue, ℓ2)
```

Algorithm 6.6: The Insert method of Path Oblivious Heap. EvictAlongPath is reused from Path ORAM.

```
1:  function ExtractMin(Queue)
2:      entry ← FindMin(Queue)
3:      ⟨ℓ⟩ ← entry.⟨leaf⟩
4:      ⟨if⟩ entry.⟨empty⟩ then
5:          ⟨ℓ⟩ ← RandomLeafLabel(Queue.n)
6:      ℓ ← Reveal(⟨ℓ⟩)
7:      ⟨done⟩ ← ⟨0⟩
8:      for Bucket in PathFromLeafToRoot(Queue, ℓ) do
9:          for i from 0 to Queue.β − 1 do
10:             entry′ ← Bucket.entries_i
11:             ⟨if⟩ ¬ ⟨done⟩ ∧ EntriesEq(entry, entry′) then
12:                 emptyEntry ← (
                        ⟨key⟩ ← ⟨0⟩ , ⟨value⟩ ← ⟨0⟩ ,
                        ⟨leaf⟩ ← ⟨0⟩ , ⟨empty⟩ ← ⟨1⟩
                    )
13:                 Bucket.entries_i ← emptyEntry
14:                 ⟨done⟩ ← ⟨1⟩
15:     EvictAlongPath(Queue, ℓ)
16:     UpdateMin(Queue, ℓ)
17:     return entry.⟨value⟩
```

Algorithm 6.7: The ExtractMin method of Path Oblivious Heap.

Unique Path Oblivious Heap

Unique POH is an adaptation of our basic POH implementation which enforces uniqueness of values in the queue. This is achieved by using any type of ORAM of size 2^v, where v is the value bit length, usually $\log n$, to maintain a map from values in the queue to their leaf indices. We first present the modified initialization method which initializes the queue *and* the value-to-leaf map:

```
1: function Initialize(n, λ, k, v, β, s, OramType)
2:     Queue ← NonRecursivePathORAM(n, λ, k, v, β, s)
3:     leafBits ← log n
4:     ValueLeafMap ← OramType(2^v, λ, leafBits)
5:     return UniqueQueue ← (Queue ← Queue, Map ← ValueLeafMap)
```

Algorithm 6.8: The Initialize method of Unique Path Oblivious Heap.

Since values are unique, we can implement an Update procedure that relies on comparison of entries by value specified in EntryValuesEq as follows:

```
1: function Update(UniqueQueue, ⟨k⟩, ⟨v⟩)
2:     ⟨ℓ⟩ , ⟨notFound⟩ ← UniqueQueue.Map_⟨v⟩
3:     ℓ ← ⊥
4:     ⟨if⟩ ⟨notFound⟩ then
5:         ⟨ℓ⟩ ← Insert(UniqueQueue, ⟨k⟩ , ⟨v⟩)
6:         UniqueQueue.Map_⟨v⟩ ← ⟨ℓ⟩
7:         ℓ ← Reveal(⟨ℓ⟩)
8:     ⟨else⟩ then
9:         entry ← (
               ⟨key⟩ ← ⟨k⟩ , ⟨value⟩ ← ⟨v⟩ ,
               ⟨leaf⟩ ← ⟨ℓ⟩ , ⟨empty⟩ ← ⟨0⟩
           )
10:        ℓ ← Reveal(⟨ℓ⟩)
11:        for Bucket in PathFromLeafToRoot(UniqueQueue, ℓ) do
12:            for i from 0 to UniqueQueue.β − 1 do
13:                entry′ ← Bucket.entries_i
14:                ⟨if⟩ EntryValuesEq(entry, entry′) then
15:                    Bucket.entries_i ← entry
16:    EvictAlongPath(UniqueQueue, ℓ)
17:    UpdateMin(UniqueQueue, ℓ)
```

Algorithm 6.9: The Update method of Unique Path Oblivious Heap. We assume that Insert (on line 5) returns the oblivious label of the inserted value, so we can update the value-to-leaf map.

```
1: function EntryValuesEq(entry, entry′)
2:     return ¬entry.⟨empty⟩ ∧ ¬entry′.⟨empty⟩ ∧
                  (entry.⟨value⟩ = entry′.⟨value⟩)
```

Algorithm 6.10: The oblivious EntryValuesEq method of Unique Path Oblivious Heap. Returns $\langle 1 \rangle$ if the entries are equal and $\langle 0 \rangle$ otherwise.

Update either updates an existing value or inserts a new value. This is an example of *type hiding security* (see Remark 4.10 on page 31), since it is oblivious whether the value is updated or inserted. This also means that the algorithmic complexity will be the sum of the complexities of inserting and updating, i.e. still $\Theta(\log n + e(n) + a(2^{\nu}))$ where $e(\cdot)$ is the eviction complexity and $a(\cdot)$ is the ORAM access complexity, assuming $s = \beta = \Theta(1)$. However, the constants hidden in the asymptotic complexity are higher than had we performed either an update or an insertion. In total, 3 calls to Evict and 3 calls to UpdateMin are issued during a call to Update, and a single tree path of length $\Theta(\log n)$ is traversed in order to update a value obliviously.

6.2 Analysis

In this section, we present an analysis of the security and efficiency of POH. We conclude by comparing it to Unique POH and the ORAM min-heap [25].

6.2.1 Security

We have already discussed correctness and obliviousness of POH (Protocol 4.3) in Section 4.3.1 on page 43, so in this chapter we will simply argue that the implementation we have presented follows Protocol 4.3, except for one difference, which does not affect security: The primary difference between our implementation and Protocol 4.3 is that our implementation does not maintain a stack of unconsumed unique identifiers. This just means that we consider possible access patterns that are a subset of possible access patterns in Protocol 4.3, and thus they are at least as easy to simulate in this implementation. We now consider the access pattern generated by UpdateMin: the leaf ℓ is revealed, and this defines a leaf-to-root path of bucket accesses. Since we obliviously compare every entry that we access (EntriesLt and EntriesEq are clearly translated into arithmetic circuits evaluated on secret values in MPC), and since ℓ is uniformly random, this does not reveal anything about the actual minimum entry. The same argumentation holds for Insert, where the traversed paths are uniformly random and independent of the leaf label assigned to the inserted entry, and for Extract where the oblivious $\langle$**if**$\rangle$ statement on line 10 ensures that the access pattern generated by its body is independent of whether the condition evaluates to true or false. We conclude that the implementation follows Protocol 4.3 and hence, it is correct and statistically oblivious.

6.2.2 Efficiency

We now analyze the circuit complexity of each individual operation of POH. Recall that β denotes the bucket size and s denotes the stash size. We will primarily consider the number of secret multiplications performed, since this is the standard measure of circuit complexity in SPDZ where other arithmetic operations can be computed locally by all parties. For convenience, we let $r = \beta \log n + s$ denote the length of a path including the stash. Note that the complexities are stated in terms of block complexity. The bit complexity can be derived by multiplying with the block size $b = \Omega(\log n)$.

Initialize The running time of Initialize is proportional to the size of the data structure, and the total local computation cost is $\Theta(n)$. However, no online communication is required since initialization is merely a question of secret sharing dummy values using preprocessed random data.

UpdateMin This operation scans a tree path and performs a constant number of multiplications per entry, in order to compare the entries, resulting in a complexity of $\Theta(r)$

EvictAlongPath We use the eviction from the Path ORAM implementation described in [25]. Without the need for a recursive position map, this operation has a complexity of $\Theta(r \log r)$.

FindMin Finally, FindMin is a simple lookup, so the complexity is $\Theta(1)$, and it even has zero communication cost.

Insert In order to insert an entry, we first insert it into the stash, and then we evict and call UpdateMin along two paths. This has a total complexity of $s + 2\Theta(r \log r) + 2\Theta(r) = \Theta(s + r \log r)$.

ExtractMin To extract the minimum entry, we look it up in constant time with a call to FindMin, after which we perform a linear scan of order $\Theta(r)$. Finally, we call EvictAlongPath and UpdateMin, resulting in a total complexity of $\Theta(r \log r)$.

If $s = \beta = \Theta(1)$, then $r = \Theta(\log n)$, which means that Insert and ExtractMin both have a block complexity of $\Theta(\log n \log \log n)$.

Comparison of POH, Unique POH, and the ORAM min-heap [25] Since Unique POH uses ORAM to maintain a map of size 2^v, its total complexities are determined by the ORAM access complexity as well as the POH complexities. The min-heap implemented on top of Path ORAM by Keller and Scholl [25] (which we will call "Path ORAM Heap" in the following) has the operational complexity of a classic min heap with Path ORAM access overhead. In Table 6.1 on the next page, we compare the complexities of all three approaches.

Table 6.1: A comparison of the operational block complexities of POH and Unique POH (using Path ORAM and assuming $v = \log n$) with $s = \beta = \Theta(1)$ and of Path ORAM Heap [25]. The $\tilde{\Theta}$ notation hides a $\log \log n$ factor.

Operation	Queue variant		
	POH	Unique POH	Path ORAM Heap [25]
Initialize	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Insert/Extract	$\tilde{\Theta}(\log n)$	$\tilde{\Theta}(\log^2 n)$	$\tilde{\Theta}(\log^3 n)$
FindMin	$\Theta(1)$	$\Theta(1)$	$\tilde{\Theta}(\log^2 n)$

6.3 Evaluation

6.3.1 Experimental Setup

Hardware and network We have run all our experiments on an Apple M1 ARM-based CPU with 8 GB of RAM. MPC parties communicate over localhost, so we are simulating a high-speed LAN setting with a bandwidth measured to approximately 7.5 GBps.

MPC Protocol We run our experiments for two parties in MP-SPDZ using the LowGear SPDZ protocol by Keller, Pastro, and Rotaru [24] which is recommended as the most efficient implementation for a small number of parties. The protocol is actively secure, tolerates a dishonest majority, and is instantiated with a cheating probability of 2^{-40} and a field size of 2^{64}. The entire data structure state is shared between all parties taking part in the protocol.

Metrics When measuring efficiency, we distinguish between offline, input-independent preprocessing and online, input-dependent computation. The offline cost is measured in terms of the number of random bits and triples needed. The online cost is measured as the time it takes to perform an operation, including communication between the parties. For the online-only benchmarking, we rely on insecure preprocessed data that is being reused, as this is what is supported for online benchmarking using the LowGear protocol in MP-SPDZ.

Concrete Instantiation and Parameters For all our experiments, we use a bucket capacity of $\beta = 2$ based on empirical estimates [33, Remark 4]. Furthermore, estimates from [25] also show that a stash size of 48 is sufficient to obtain a negligible error probability, so we only briefly compare this stash size to a superlogarithmic stash size of $\log^2 n$ in Fig. 6.1, which unsurprisingly shows that using a constant stash size results in a significant performance gain when $\log^2 n > 48$, that is from about $n > 2^{\sqrt{48}} \approx 2^7$, before we adopt a stash size of 48 for the rest of our experiments.

For keys and values, we adopt a standard setting of 32-bit keys and $\log n$-bit values, i.e. $k = 32$ and $v = \log n$.

MP-SPDZ makes it a simple task for developers to take advantage of parallelism and multithreading. Furthermore, constant optimization is made at a low level when compiling a program, in order to minimize communication and local computation cost. This is great for speeding up practical performance. However, for our benchmarking purposes, we have – to the best of our ability – made an attempt to disable these kinds of optimizations since they often result in sudden increases in performance when the optimization kicks in that are hard to explain, e.g. when a data structure reaches a certain capacity. We have mostly succeeded this task, but we suspect that there are still some low-level optimizations that are responsible for a number of sudden jumps in performance.

6.3.2 Results

First, we have compared the online time of executing Insert in POH and Unique POH in Fig. 6.2, and as expected, Unique POH has a performance overhead due to its use of ORAM which is what our analysis predicted. However, from our experimental results, it is not clear that this overhead is asymptotic; in fact, it rather appears to be a constant overhead. We hypothesize that there is indeed an asymptotic overhead as predicted, but that it will only start to show for larger values of n. Furthermore, we see that a series of performance jumps happen, for example at $n = 2^{16}$ for Unique POH. We hypothesize that these jumps stem from optimizations in Path ORAM or at a lower level in MP-SPDZ since they consistently appear in all implementations. The random bits and triples required, as shown in Fig. 6.3 give a cleaner picture that is not affected by optimization to the same extent.

Next, we have compared Unique POH to the Path ORAM Heap baseline. We show the online time required to execute Insert in Fig. 6.4 and the associated offline cost in Fig. 6.5. Here, it is also clear that Unique POH outperforms Path ORAM Heap, starting from around $n = 2^6$. For completeness, we also compare the online time required to execute ExtractMin in Fig. 6.6 which shows the same picture, but with Unique POH starting to outperform Path ORAM Heap already from around $n = 2^3$. For $n = 2^{20}$, Unique POH is orders of magnitude faster than Path ORAM Heap.

Finally, we have timed the online initialization time for all three queues, showed in Fig. 6.7. As expected, the plot shows superlinear trends (the bit cost of initialization is $\Theta(n \log n)$) for all three queues. However, Unique POH and Path ORAM Heap have a significant constant overhead compared to initializing POH because of the recursive position map.

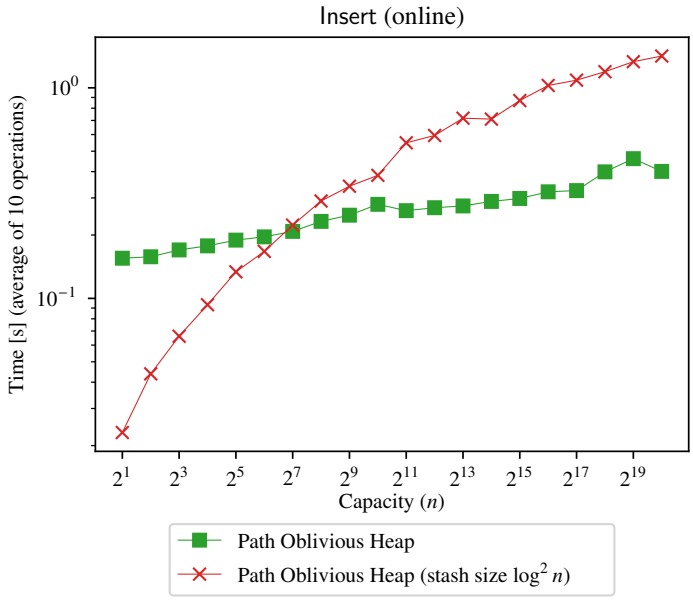


Figure 6.1: A comparison of the online time it takes to execute Insert for POH with a constant and superlogarithmic stash size.

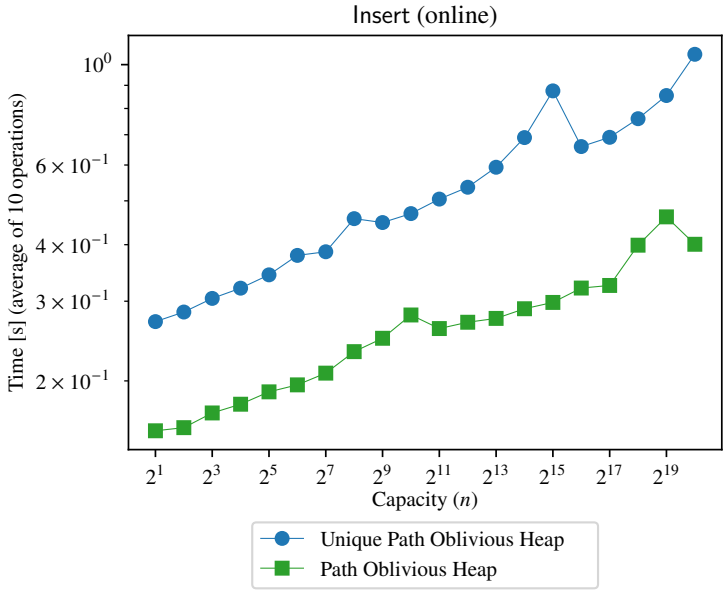


Figure 6.2: A comparison of the online time it takes to execute Insert for POH and Unique POH.

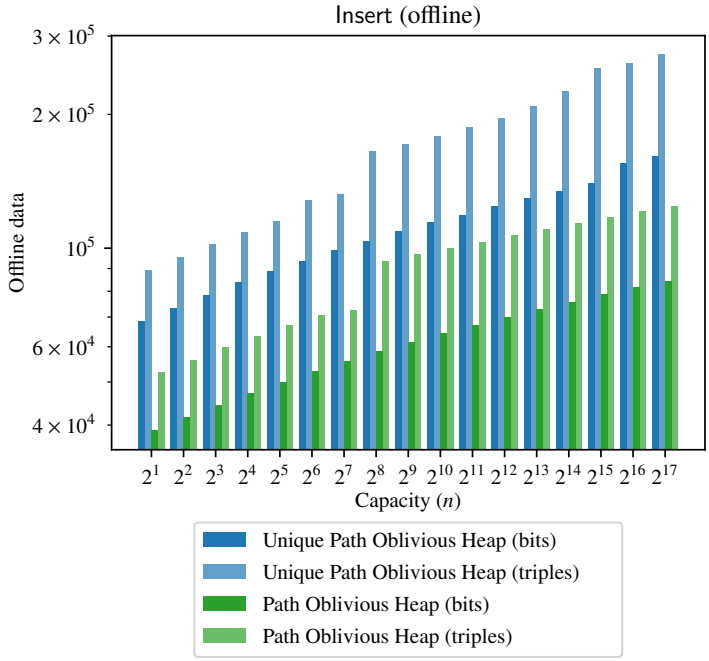


Figure 6.3: A comparison of the offline cost of executing Insert for POH and Unique POH.

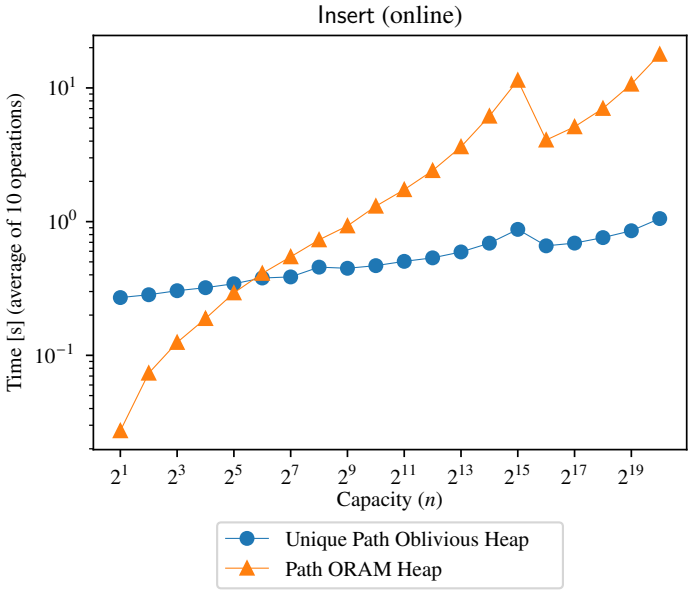


Figure 6.4: A comparison of the online time it takes to execute Insert for Unique POH and Path ORAM Heap.

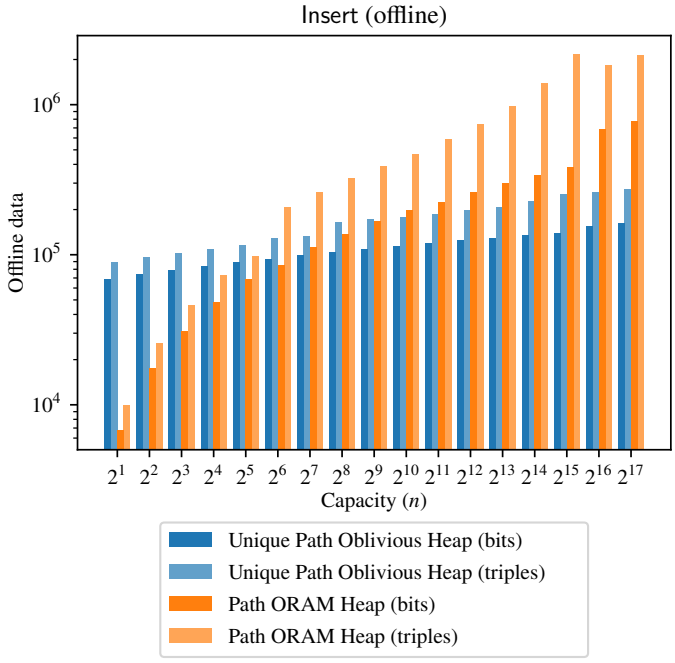


Figure 6.5: A comparison of the offline cost of executing Insert for Unique POH and Path ORAM Heap.

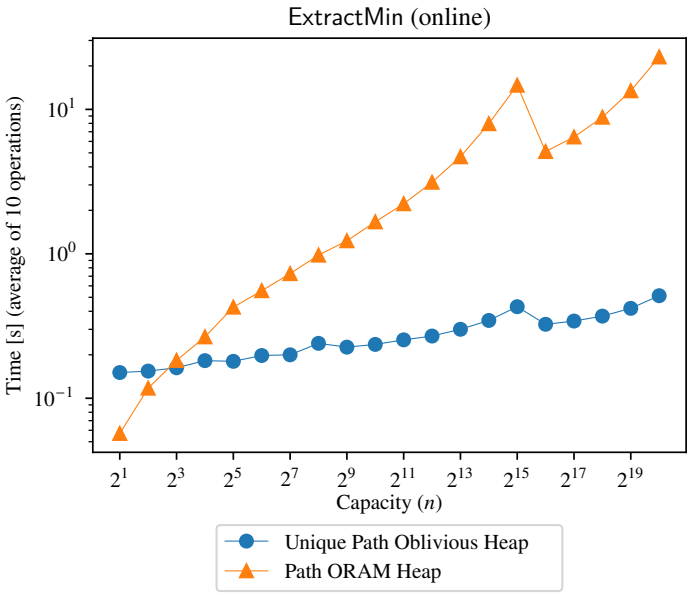


Figure 6.6: A comparison of the online time it takes to execute ExtractMin for Unique POH and Path ORAM Heap.

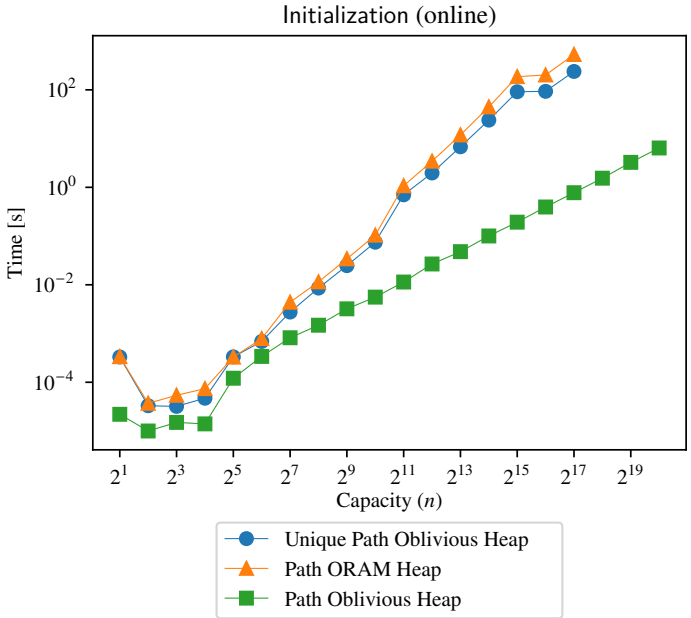


Figure 6.7: A comparison of the online time it takes to initialize POH, Unique POH, and Path ORAM Heap.

Chapter 7

Conclusion and Future Perspectives

In this thesis, we perform a survey and present new experimental results, both of which indicate that generic MPC is reaching a degree of maturity that makes it practical in the setting of evaluating programs that perform input-dependent data access, thanks to

- the development of publicly available generic MPC frameworks such as MP-SPDZ, that support high-level oblivious programming which we have described in the framework presented in Chapter 5, as well as
- the increasing availability of practical, specialized oblivious data structures such as Path Oblivious Heap.

Our experimental evaluation of POH and Unique POH in the SPDZ MPC protocol shows results that match the theoretical predictions while at the same time guaranteeing dishonest-majority, active security. For a queue of size 2^{20} and a cheating probability of 2^{-40} in a 64-bit arithmetic field, the online access time in SPDZ is about a second for Unique POH (and 400 ms for POH). The evaluation shows compelling and promising practical performance of POH and Unique POH which improves upon the state of the art and opens up the opportunity of many interesting applications which we hope to witness in a not-too-distant future.

The experimental work in this thesis is primarily concerned with evaluating the performance of POH. Since an oblivious priority queue is a basic building block in many algorithms, such as oblivious sorting or oblivious computation of Dijkstra's shortest path algorithm [8], evaluating the performance of POH in such applications – or even using oblivious algorithms built on top of POH to solve real-world problems – makes for an interesting line of research. In MP-SPDZ, the currently implemented oblivious sorting algorithm is *Oblivious Radix Sort*, proposed by Hamada et al. [17], which is very efficient for a small number of MPC parties. However, the asymptotic complexity of Oblivious Radix Sort is exponential in the number of MPC parties.

Thus, we predict that Path Oblivious Sort would make for an interesting alternative in some scenarios.

Finally, a priority queue is only one of many data structures that are relevant in an MPC context. Inventing new, specialized oblivious data structures is also important in order to further improve the state of practical generic MPC. Another way of approaching this task would be to first identify an application of MPC where the current state-of-the-art solutions are too inefficient for practical use, and then try to invent a new oblivious data structure that might lead to more efficient and practical solutions.

Bibliography

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. "An O(n log n) Sorting Network." In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. Ed. by David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas. ACM, 1983, pp. 1–9. DOI: 10.1145/800061.808726. URL: https://doi.org/10.1145/800061.808726.

[2] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. "Securely Solving Simple Combinatorial Graph Problems." In: *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*. Ed. by Ahmad-Reza Sadeghi. Vol. 7859. Lecture Notes in Computer Science. Springer, 2013, pp. 239–257. DOI: 10.1007/978-3-642-39884-1_21. URL: https://doi.org/10.1007/978-3-642-39884-1%5C_21.

[3] Kenneth E. Batcher. "Sorting Networks and Their Applications." In: *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*. Vol. 32. AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., 1968, pp. 307–314. DOI: 10.1145/1468075.1468121. URL: https://doi.org/10.1145/1468075.1468121.

[4] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization." In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. Ed. by Joan Feigenbaum. Vol. 576. Lecture Notes in Computer Science. Springer, 1991, pp. 420–432. DOI: 10.1007/3-540-46766-1_34. URL: https://doi.org/10.1007/3-540-46766-1%5C_34.

[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. "Garbling Schemes." In: *IACR Cryptol. ePrint Arch.* (2012), p. 265. URL: http://eprint.iacr.org/2012/265.

[6] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols." In: *IACR Cryptol. ePrint Arch.* (2000), p. 67. URL: http://eprint.iacr.org/2000/067.

[7] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* (2011), p. 535. URL: http://eprint.iacr.org/2011/535.

[8] Edsger W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[9] Jack Doerner and Abhi Shelat. "Scaling ORAM for Secure Computation." In: *IACR Cryptol. ePrint Arch.* (2017), p. 827. URL: http://eprint.iacr.org/2017/827.

[10] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. "Lower Bounds for External Memory Integer Sorting via Network Coding." In: *CoRR* abs/1811.01313 (2018). arXiv: 1811.01313. URL: http://arxiv.org/abs/1811.01313.

[11] Taher El Gamal. "A public key cryptosystem and a signature scheme based on discrete logarithms." In: *IEEE Trans. Inf. Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074. URL: https://doi.org/10.1109/TIT.1985.1057074.

[12] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. "Optimizing ORAM and Using it Efficiently for Secure Computation." In: *IACR Cryptol. ePrint Arch.* (2013), p. 239. URL: http://eprint.iacr.org/2013/239.

[13] Oded Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs." In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. Ed. by Alfred V. Aho. ACM, 1987, pp. 182–194. DOI: 10.1145/28395.28416. URL: https://doi.org/10.1145/28395.28416.

[14] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *J. ACM* 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553. URL: https://doi.org/10.1145/233551.233553.

[15] Michael T. Goodrich and Michael Mitzenmacher. "MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations." In: *CoRR* abs/1007.1259 (2010). arXiv: 1007.1259. URL: http://arxiv.org/abs/1007.1259.

[16] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. "Secure two-party computation in sublinear (amortized) time." In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, Octo-*

ber 16-18, 2012. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 513–524. DOI: 10.1145/2382196.2382251. URL: https://doi.org/10.1145/2382196.2382251.

[17] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. "Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation." In: *IACR Cryptol. ePrint Arch.* (2014), p. 121. URL: http://eprint.iacr.org/2014/121.

[18] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010. ISBN: 978-3-642-14302-1. DOI: 10.1007/978-3-642-14303-8. URL: https://doi.org/10.1007/978-3-642-14303-8.

[19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. "Lower Bounds for Oblivious Data Structures." In: *CoRR* abs/1810.10635 (2018). arXiv: 1810.10635. URL: http://arxiv.org/abs/1810.10635.

[20] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. "Lower Bounds for Oblivious Data Structures." In: *CoRR* abs/1810.10635 (2018). arXiv: 1810.10635. URL: http://arxiv.org/abs/1810.10635.

[21] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. "Optimal Oblivious Priority Queues and Offline Oblivious RAM." In: *IACR Cryptol. ePrint Arch.* (2019), p. 237. URL: https://eprint.iacr.org/2019/237.

[22] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation." In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417872. URL: https://doi.org/10.1145/3372297.3417872.

[23] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer." In: *IACR Cryptol. ePrint Arch.* (2016), p. 505. URL: http://eprint.iacr.org/2016/505.

[24] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: Making SPDZ Great Again." In: *IACR Cryptol. ePrint Arch.* (2017), p. 1230. URL: http://eprint.iacr.org/2017/1230.

[25] Marcel Keller and Peter Scholl. "Efficient, Oblivious Data Structures for MPC." In: *IACR Cryptol. ePrint Arch.* (2014), p. 137. URL: http://eprint.iacr.org/2014/137.

[26] Marcel Keller and Avishay Yanai. "Efficient Maliciously Secure Multiparty Computation for RAM." In: *IACR Cryptol. ePrint Arch.* (2017), p. 981. URL: http://eprint.iacr.org/2017/981.

[27] Kasper Green Larsen and Jesper Buus Nielsen. “Yes, There is an Oblivious RAM Lower Bound!” In: *IACR Cryptol. ePrint Arch.* (2018), p. 423. URL: https://eprint.iacr.org/2018/423.

[28] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. “Can We Overcome the n log n Barrier for Oblivious Sorting?” In: *IACR Cryptol. ePrint Arch.* (2018), p. 227. URL: http://eprint.iacr.org/2018/227.

[29] Claudio Orlandi. “Is multiparty computation any good in practice?” In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*. IEEE, 2011, pp. 5848–5851. DOI: 10.1109/ICASSP.2011.5947691. URL: https://doi.org/10.1109/ICASSP.2011.5947691.

[30] Emmanuela Orsini. “Efficient, Actively Secure MPC with a Dishonest Majority: a Survey.” In: *IACR Cryptol. ePrint Arch.* (2022), p. 417. URL: https://eprint.iacr.org/2022/417.

[31] Rafail Ostrovsky and Victor Shoup. “Private Information Storage.” In: *IACR Cryptol. ePrint Arch.* (1996), p. 5. URL: http://eprint.iacr.org/1996/005.

[32] Benny Pinkas and Tzachy Reinman. “Oblivious RAM Revisited.” In: *IACR Cryptol. ePrint Arch.* (2010), p. 366. URL: http://eprint.iacr.org/2010/366.

[33] Elaine Shi. “Path Oblivious Heap.” In: *IACR Cryptol. ePrint Arch.* (2019), p. 274. URL: https://eprint.iacr.org/2019/274.

[34] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. “Oblivious RAM with O((log N)^3) Worst-Case Cost.” In: *IACR Cryptol. ePrint Arch.* (2011), p. 407. URL: http://eprint.iacr.org/2011/407.

[35] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. “Path ORAM: An Extremely Simple Oblivious RAM Protocol.” In: *IACR Cryptol. ePrint Arch.* (2013), p. 280. URL: http://eprint.iacr.org/2013/280.

[36] Abraham Waksman. “A Permutation Network.” In: *J. ACM* 15.1 (1968), pp. 159–163. DOI: 10.1145/321439.321449. URL: https://doi.org/10.1145/321439.321449.

[37] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. “Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound.” In: *IACR Cryptol. ePrint Arch.* (2014), p. 672. URL: http://eprint.iacr.org/2014/672.

[38] Xiao Wang, Kartik Nayak, Chang Liu, Elaine Shi, Emil Stefanov, and Yan Huang. “Oblivious Data Structures.” In: *IACR Cryptol. ePrint Arch.* (2014), p. 185. URL: http://eprint.iacr.org/2014/185.

[39] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. “SCORAM: Oblivious RAM for Secure Computation.” In: *IACR Cryptol. ePrint Arch.* (2014), p. 671. URL: http://eprint.iacr.org/2014/671.

[40] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract).” In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25. URL: https://doi.org/10.1109/SFCS.1986.25.

[41] Samee Zahur and David Evans. “Obliv-C: A Language for Extensible Data-Oblivious Computation.” In: *IACR Cryptol. ePrint Arch.* (2015), p. 1153. URL: http://eprint.iacr.org/2015/1153.

[42] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. “Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation.” In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 218–234. DOI: 10.1109/SP.2016.21. URL: https://doi.org/10.1109/SP.2016.21.