© 2023 Zexiang Chen

3PC HONEST-MAJORITY PRAM COMPUTATION WITH PERFECT SECURITY AND LOW OVERHEAD

BY

ZEXIANG CHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Professor David Heath

ABSTRACT

In this thesis, we present new techniques for three-party secure computation in the parallel random access machine (PRAM) model. Our protocol is perfectly secure and concretely efficient. Considering a PRAM machine storing n w-bit words and having a large number ($p = O(n)$) of processors, and assuming at most one passively corrupt party, our construction exhibits the following properties:

- **Minimal cryptographic assumptions:** By carrying out all computations using secret shares, our protocol achieves perfect security without any cryptographic assumptions.

- **Low communication complexity:** To serve p queries to our PRAM in parallel, our construction requires only

$$O\left(\log^2(p)\log(n)\right) + \log\binom{n}{p} O(w\log(n) + \log^2(n))$$

bits of transmission per query, amortized over the total number of queries. In our setting of $p = O(n)$, this becomes

$$O(w + \log^3(n)),$$

matching the known lower bounds on Oblivious RAM if $w = \Omega(\log^2(n))$. The low constant factors in our construction also ensure that our protocol is concretely efficient. Specifically, with $n = 2^{25}, w = 625, p = 2^{16}$, n queries to our PRAM requires a transmission of 123130 bits per query.

- **Low round complexity:** By carefully leveraging the inherent parallelism available in the PRAM model, we were able to reduce the round complexity of each query. To serve p queries to our PRAM in parallel, our construction requires only

$$O(\log^2(p)\log\log(n)) + \log\binom{n}{p} O(\log(p) + (\log\log(n))^2)$$

ronuds of communications. When setting $p = O(n)$, this becomes

$$O(\log^2(n)\log\log(n)),$$

which states that our rounds scales only **logarithmically** in n. The low constant factors we have contribute to our protocol's concrete efficiency, allowing it to serve each set of parallel queries in 4352 rounds in the same setting as above.

Our protocol's concrete efficiency, coupled with its ability to serve p queries in parallel, makes it appealing for real-world applications such as allowing p end users to simultaneously access a shared database and receive their results back in real time.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Professor David Heath, for his unwavering guidance, support, and encouragement throughout the process of writing this thesis. His invaluable knowledge and expertise have been instrumental in shaping my research and enabling me to complete my thesis. This research project would certainly have been impossible without David's dedication, as he frequently went out of his way to guide me in the right direction.

I am sincerely grateful to my family for their boundless love, understanding, and steadfast support throughout my academic journey. Although they may not be familiar with the specifics of my work, they have always been there to help whenever I needed it. Their faith in me has provided the strength and determination to persevere and succeed. Their constant words of wisdom have greatly enriched my life and contributed to my personal and academic growth.

I would also like to extend heartfelt thanks to my friends for their unwavering support, camaraderie, and the cherished moments of joy we shared. I am particularly grateful to Zijing Di for engaging in numerous discussions on cryptography topics and for being a constant source of motivation and inspiration throughout my journey. I also wish to express my appreciation for Yongmei Lin, Shitao Shi, Boyang Sun, Haifeng Xia, and Haodan Yang, who constantly encouraged and inspired me. Some of them even generously offered to help me illustrate a complicated circuit for this thesis, even though the circuit was later replaced by a better alternative.

I'd like to thank my cat, Xiao Mi (literally, little cat), and the music bands Deca Joins and No Party For Cao Dong, for getting me through the times when I felt lost and down.

Finally, I want to express my gratitude to all the Professors I met at University of Illinois Urbana-Champaign, especially Professor Dakshita Khurana, for sparking my interest in cryptography and offering a wonderful course in quantum cryptography.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Our Contribution 1
1.2 Organization of Contents 2

CHAPTER 2 BACKGROUND 3
2.1 Definitions 3
2.2 Related Work 5

CHAPTER 3 PROTOCOL OVERVIEW 8

CHAPTER 4 FORMAL APPROACH 17
4.1 Permutation of an Array 17
4.2 The Timetable 20
4.3 The Position Maps 24
4.4 Upgrading RAM to PRAM 25

CHAPTER 5 SECURITY ANALYSIS 40
5.1 Automatically Secure Parts of our PRAM 40
5.2 Π_{perm} is Secure 42
5.3 $\Pi_{\mathsf{binsearch}}$ is Secure 44
5.4 The Rest of the PRAM Protocol is Secure 47
5.5 The Entire Protocol is Secure 49

CHAPTER 6 EFFICIENCY ANALYSIS 51

CHAPTER 7 EXPERIMENTS 55
7.1 Experiment Setups 55
7.2 Empirical Results 56

CHAPTER 8 CONCLUSIONS 62

REFERENCES 63

CHAPTER 1: INTRODUCTION

Cloud computing has gained immense popularity in recent years, providing businesses and individuals with access to powerful computing resources with unprecedented convenience. In a typical cloud computing setting, a client with limited computational capabilities outsources its data and tasks to a more powerful server. The client instructs the server to perform computationally demanding tasks, which the server then executes using its resources before delivering the results back to the client. However, since both the client's data and instructions can contain potentially sensitive information, security has emerged as a pressing issue in the cloud computing world.

To illustrate this, consider a cloud map application where the servers maintain a database of nearby places indexed by approximate location (such as a zip code), and an end user searches this database using their approximate location. For instance, a tourist exploring a new town might want to retrieve a list of nearby attractions from the database, but to use the cloud map they will need to provide their current approximate location. Understandably, the tourist, seeking to protect their privacy, would prefer not to disclose their whereabouts while making this query. Consequently, it is necessary for the servers to process this query in an oblivious manner, ensuring that neither the target database entry nor the user's access patterns are revealed. Preventing access pattern disclosure is crucial, as it would otherwise allow the servers to accumulate temporal information on users' recent whereabouts, thus compromising their privacy.

1.1 OUR CONTRIBUTION

In this work, we address the security concern in the above scenario by presenting a protocol for 3PC semi-honest secure PRAM computation, assuming only one party is passively corrupted. Our PRAM is both theoretically and practically efficient, exhibiting exceptional scalability when a high degree of parallelism is available. When the number of processors $p = O(n)$, each query incurs only $O(\log(n))$ amortized communication overhead, and each set of p parallel queries can be served in only $O(\log^2(n)\log\log(n))$ rounds. Moreover, our protocol achieves perfect security without any cryptographic assumption by working with secret shares of data. Alternatively, if we only aim for computational security, we can rely on computationally secure one-way functions to further optimize our permutation protocol, significantly improving the constants.

As a result, our protocol is well-suited for the aforementioned cloud map application, as

it allows efficient and simultaneous servicing of a large number of end users while protecting each user's privacy by operating solely on shares of their current physical location.

1.2 ORGANIZATION OF CONTENTS

In Chapter 2, we provide the background for this work and point out related works in the literature. Following this, we offer a high-level technical overview of our protocol's various components in Chapter 3, while reserving a detailed description of each component for Chapter 4, where we also discuss how to securely and efficiently handle parallel queries. In Chapter 5, we formally prove that our protocol is semi-honest secure in the honest-majority setting. To showcase the practical efficiency of our protocol, we both give an asymptotic calculation in Chapter 6 and present our empirical results in Chapter 7. Finally, we briefly summarize this work and suggest potential directions for future research in Chapter 8.

CHAPTER 2: BACKGROUND

In this chapter, we will briefly list the relevant definitions for our work and then proceed to review some existing works in the literature that are closely related to our own.

2.1 DEFINITIONS

Throughout this work, the model of computation that we use will be that of a PRAM machine commonly found in the literature [1]. Roughly speaking, a PRAM machine is a machine augmented with p processors, all of which can access a shared storage structure using special read/write instructions. The following definition is adapted from [1]:

Definition 2.1 (PRAM machine)**.** A p-processor PRAM with a memory size of n consists of processors numbered from 0 to $p-1$, which operate synchronously and in parallel. These processors can access a shared memory of size n with an address space of $[0, n-1]$. Each processor is equipped with an instruction set that contains two special instructions: Read and Write.

Read takes a physical address $i \in [0, n-1]$ as input. After completion of Read, the memory element at location i is returned to the processor that issued the instruction. Similarly, Write also takes a physical address $i \in [0, n-1]$ as input, along with an w-bit string x, where w is the word size of the shared memory. Upon completion of Write, the old memory element at location i is returned to the processor that issued the instruction, and x takes its place in the shared memory.

If more than one processor issues a Write instruction to the same location, the old memory element is returned to all processors involved. However, only the processor with the smallest ID can succeed in replacing the element at that location with its x.

The security definition our PRAM protocol achieves is the standard simulation-based semi-honest security definition found in [2]. In the semi-honest setting, each party participating in the protocol will follow the protocol specification, although they will try to learn the other parties' private inputs from its transcript, which is defined as the union of its internal states and messages it receives. Informally, we say that a MPC protocol is semi-honest secure if no parties can learn other parties' private inputs, except what's already implied by their outputs. The definition in [2] captures notion by requiring that each party's view, which consists of its input, its random coin tosses, and the messages it receives, be reproducible in polynomial time from the party's input and output only. The following definition is adapted from [2]:

Definition 2.2 (3PC semi-honest secure protocol with honest majority)**.** Let $f : \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$ be a functionality, let the parties be denoted by P_0, P_1, P_2, and let $f_i(x, y, z)$ denote the ith element of $f(x, y, z)$. Let Π be a 3PC protocol for computing f. The **view** of P_0 during an execution of Π on (x, y, z), denoted $\mathsf{VIEW}_0^{\Pi}(x, y, z)$, is $(x, r_0, m_1, \ldots)$, where r_0 is P_0's private coin tosses and m_i is the ith message it has received. The views for the other two parties are analogously defined. The *output* of the ith party after an execution of Π on (x, y, z), denoted $\mathsf{OUTPUT}_i^{\Pi}(x, y, z)$, is implicit in the party's own view of the execution, and $\mathsf{OUTPUT}^{\Pi}(x, y, z) = (\mathsf{OUTPUT}_0^{\Pi}(x, y, z), \ldots, \mathsf{OUTPUT}_2^{\Pi}(x, y, z))$.

We say that Π is a semi-honest secure protocol for f with honest majority if there exists probabilistic polynomial-time simulators, denoted S_0, S_1, S_2, such that

$$\{(S_0(x, f_0(x, y, z)), f(x, y, z))\}_{x,y,z} \stackrel{c}{\equiv} \{(\mathsf{VIEW}_0^{\Pi}(x, y, z), \mathsf{OUTPUT}^{\Pi}(x, y, z)\}_{x,y,z} \tag{2.1}$$

$$\{(S_1(y, f_1(x, y, z)), f(x, y, z))\}_{x,y,z} \stackrel{c}{\equiv} \{(\mathsf{VIEW}_1^{\Pi}(x, y, z), \mathsf{OUTPUT}^{\Pi}(x, y, z)\}_{x,y,z} \tag{2.2}$$

$$\{(S_2(z, f_2(x, y, z)), f(x, y, z))\}_{x,y,z} \stackrel{c}{\equiv} \{(\mathsf{VIEW}_2^{\Pi}(x, y, z), \mathsf{OUTPUT}^{\Pi}(x, y, z)\}_{x,y,z} \tag{2.3}$$

where $\stackrel{c}{\equiv}$ denotes computational indistinguishability. The above definition is for computational security, if we want perfect security, then we require that the ensembles in (2.1)-(2.3) be identically distributed.

In our setting, we will call the three parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$, where $\mathcal{S}_0$ and $\mathcal{S}_1$ are the two *servers* that shares the RAM contents and $\mathcal{D}$ is a dealer who only exists to help $\mathcal{S}_0$ and $\mathcal{S}_1$ in executing the queries. As it turns out, the physical access patterns are almost always included in the views of the two servers, and therefore, our PRAM satisfies the traditional ORAM definition of obliviousness almost "for free". Informally, this definition requires that the physical access patterns resulting from any two sequences of accesses are indistinguishable, as long as they access the same number of elements. Since we are working with a PRAM, we will slightly modify the sequences of accesses in the original definition to accommodate parallel accesses, but the rest is directly adopted from [3].

Definition 2.3 (Obliviousness)**.** Let $\mathcal{Q}_t = (q_0, \ldots, q_{p-1})$ be a vector of p parallel queries issued at time t, and let $Q = (\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_{m-1})$ be a sequence of parallel queries from time 0 to $m-1$. We define the access patterns of Q to be the sequence of physical accesses made to the storage, and denote it by $A(Q)$. An ORAM construction is said to be secure if for any two such sequence of parallel queries Q, Q' of the same length, their access patterns $A(Q)$ and $A(Q')$ are computationally indistinguishable by anyone but the client who issued them. If we want perfect security, we require that $A(Q)$ and $A(Q')$ be identically distributed.

With the necessary definitions now in place, we are ready to state our main theorem.

Theorem 2.1 (Main)**.** Our PRAM is secure in the sense of Definition 2.2, with the inputs from $\mathcal{S}_0$ and $\mathcal{S}_1$ being shares of the memory contents and a sequence of parallel queries as in Definition 2.3 and the input from $\mathcal{D}$ being λ (i.e., it has no inputs). Let $\mathcal{R}_t$ be the parallel query results corresponding to the parallel queries $\mathcal{Q}_t$, and let $R = (\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{p-1})$ be the sequence of parallel query results corresponding to R. The outputs of $\mathcal{S}_0$ and $\mathcal{S}_1$ is a share of R and the output of $\mathcal{D}$ is again λ.

2.2 RELATED WORK

In this section, we summarize the similarities and differences between our work and existing works. The line of research that aligns most closely with our work focuses on Oblivious RAMs (ORAMs) or Oblivious Parallel RAMs (OPRAMs). However, as we will see, there are crucial distinctions between our work and these constructions in each case. In fact, our PRAM protocol is provably more powerful than both. Specifically, we will demonstrate that our PRAM protocol satisfies the classical ORAM or OPRAM security definition of access pattern hiding [3], while having additional properties that makes it desirable for secure RAM computation. Remarkably, this proof almost immediately follows from the simulation-based security proof that we will give.

Oblivious RAMs (ORAMs). A closely related and extensively studied line of research is Oblivious RAMs (ORAMs), initially investigated by Goldreich and Ostrovsky in [4, 5]. The main object of study in ORAMs is a compiler that can transforms a RAM program into an ORAM program, where all the memory accesses (which logical address is read/written at what time) are hidden. An ORAM compiler thus acts as an interface between a client and its memory (or possibly a remote server). A significant amount of effort has been devoted to understanding the theoretical limitations of ORAMs [5, 6, 7] and constructing practically efficient ORAMs [8, 9]. With reasonable parameter settings, the seminal Path-ORAM construction in [8] achieves the theoretical lower bounds established in [5], while maintaining practical efficiency.

Our PRAM protocol surpasses traditional ORAMs in three fundamental ways. While each of these shortcomings of ORAMs has been addressed separately, to the best of our knowledge, no work has yet tackled all three problems at the same time. These shortcomings are as follows:

- Our PRAM can handle p queries concurrently, whereas a traditional ORAM must

process them sequentially. The line of research focusing on enhancing this aspect of ORAMs is known as *oblivious parallel RAMs* (OPRAMs).

- In a client-server setting, our PRAM protocol leverages the computational power of servers to perform the majority of the computation. Meanwhile, the less powerful client simply shares its query and awaits the query result. The line of research aiming to reduce client computation by utilizing server capabilities is sometimes referred to as *oblivious storage*.
- The basic non-parallel version of our RAM can be easily embedded into a 3PC semi-honest secure MPC protocol using the RAM model (RAM-MPC protocol), tolerating up to 1 passively corrupt party. Any RAM-MPC protocol can treat our secure RAM as a black-box that accepts shares of logical addresses and returns shares of elements. The line of research seeking to develop MPC-friendly ORAMs is called secure-computation ORAM (SCORAM) or MPC-ORAM.

Oblivious Parallel RAMs (OPRAMs). The first and most obvious distinction between our PRAM and a traditional ORAM is our ability to handle parallel queries to servers, whereas traditional ORAMs can only manage sequential queries. Some works [1, 10, 11, 12, 13] have extended the ideas behind ORAMs to the PRAM setting, constructing OPRAM protocols that enable parallel memory access similar to our protocol. Much like our own work, [11] even developed a perfectly secure OPRAM. However, unlike our concretely efficient PRAM, their construction remains largely theoretical due to the use of expensive primitives such as expander graphs.

Oblivious storage. The second difference between our protocol and a traditional ORAM is the amount of clients' computation. In traditional ORAMs, the servers are assumed to be memory devices supporting only reads and writes, whereas in applications such as cloud computing the servers are also powerful machines. Consequently, traditional ORAM clients must access a logarithmic number of physical addresses and periodically shuffle memory elements to hide access patterns, and for this they need to invest some computational resources. A lot of works [3, 14, 15, 16, 17, 18, 19] tried to leverage the resourceful servers to accelerate an ORAM access. These constructions are sometimes called *oblivious storages*. However, these constructions differ from our PRAM because they either rely on expensive cryptographic primitives such as fully homomorphic encryption or still require clients to expend some effort to execute a query.

Secure-computation RAM (SCORAM) or MPC-ORAM. Lastly, integrating a traditional ORAM into an MPC protocol requires expressing the client's and server's computations as circuits and utilizing secure circuit evaluation techniques like Yao's garbled circuit [20] or the GMW protocol [21]. In contrast, our PRAM protocol can be easily incorporated into any MPC-RAM computation, as it provides a secure RAM interface that allows RAM programs to retrieve memory elements using secret shares of logical addresses. Programs already utilizing secret-share-based techniques for security can seamlessly integrate our PRAM into their computations by treating our PRAM as a black box. This line of research [22, 23, 24, 25, 26, 27, 28], known as secure-computation ORAM (SCORAM) or MPCORAM, mostly aligns with our PRAM, except that we can handle parallel queries.

3PC secure computation. Not surprisingly, there have already been works [29, 30] on achieving semi-honest secure computation with an honest-majority, and even works [31] on achieving full security. The 3PC setting is often considered because many expensive cryptographic assumptions can be waived when a third party is involved. This third party can distribute correlated randomness to the other two, greatly assisting them in computing basic primitives such as 1-out-of-2 oblivious transfers. Both previous [29] and recent [30] works have developed concretely efficient 3PC semi-honest secure protocols with an honest-majority, with [30] even achieving $O(1)$ rounds of communication. However, none of these constructions consider the PRAM model, making them less powerful than our PRAM protocol when a high degree of parallelism is available.

CHAPTER 3: PROTOCOL OVERVIEW

In this section, we give a comprehensive overview of our 3PC PRAM protocol. To set the stage, we first describe a RAM protocol (i.e., without parallelism). The key property of this RAM protocol is (1) its high efficiency and (2) its natural compatibility with parallelism. We then demonstrate how to upgrade the base protocol with parallelism.

At a high level, our PRAM protocol will assign different roles to the different parties. Two parties will be assigned the roles of servers, and the third will be assigned the dealer. By carefully leveraging the dealer, we ensure that basic RAM operation can be cheaply implemented. In short, the two servers will store the actual RAM elements and perform most of the actual work; the dealer will prepare large amounts of metadata useful for searching for memory elements, while also aiding the servers in otherwise expensive tasks such as permuting an array.

The challenge in designing a secure PRAM is to make the following two cases look indistinguishable to the parties:

1. When all the queries are issued to distinct logical addresses of the RAM. In this case, we say the queries are non-conflicting.
2. When there are at least two queries that are issued to the same logical address in the RAM. In this case, we say the queries are conflicting.

The former case is easier to handle as we can assign each query to its own processor and ask the processors to execute their own queries in parallel. To deal with the conflicting queries, our idea is to resolve their conflicts and then fall back to the non-conflicting case. We start with replacing duplicate queries by dummy queries, so that each logical address is requested by at most one query. We then proceed in the same way we handle collision-free queries, followed by additional postprocessing of the query results to make sure that the each processor receives the content of the logical address it requested for.

Our setting. We have 3 parties that together securely implement the RAM functionality. We have two servers we denote by $\mathcal{S}_0$ and $\mathcal{S}_1$, and a dealer we denote by $\mathcal{D}$. The servers hold secret shares of the array, which the dealer does not have access to. The job of the dealer is to assist the two servers in computing functionalities that would otherwise require expensive cryptographic operations. For example, as we will see in section 4.1, permuting an array of elements can be done efficiently with communication complexity $O(wn)$, where w is the size of elements stored in the array. We will also assume that $\mathcal{S}_0$ and $\mathcal{S}_1$ are powerful

machines both have p processors numbered from p_1 to p_p, where each processor is capable of executing its own instructions independently of the other processors, including the ability to simultaneously initiate communication to the other server.

Terminology. Throughout this work, we will use the following terminology to describe the "physical" and "logical" layers of our protocol:

- **Storage** refers to the physical memory that underlies our RAM, and we simply view it as a 0-indexed randomly access large contiguous array.
- **Physical address** or **position** refers to the index into the storage, i.e. the index into the underlying physical array.
- **RAM** refers to the logical memory infrastructure that our protocol builds on top of the storage, an user of our protocol simply views the RAM as an 0-indexed random access array capable of supporting read/write accesses. The additional operations required to protect the access patterns are completely transparent to the user. We additionally reserve this term for version of our protocol that isn't yet capable of serving parallel queries.
- **PRAM** refers to the parallel version of the basic RAM. An user of our protocol simply views the PRAM as an 0-indexed random access array capable of supporting p many read/write accesses at the same time.
- **Logical address** refers to the index into the RAM, i.e. the index the user uses to access the RAM. Note that the logical address needs **not** equal the physical address, and in fact they almost always differ for security concerns.
- An **access** or a **query** into our RAM (or PRAM) is a request to read or write an element the user stored in the RAM (or PRAM). The users issue an access or query by secret sharing the logical address it wants to access between $\mathcal{S}_0$ and $\mathcal{S}_1$. Since the servers both have p independent processors, our PRAM can fulfill up to p many accesses or queries at the same time.

Storage structure. We design our RAM so that each instance of the RAM can handle m accesses, where m is a parameter the user can customize. We only require that $m = 2^i n$ for some constant $i \geq 0$, with n, also a power of 2, being the number of elements stored in our RAM. Considering only powers of 2 is a simplifying assumption, and it is easy to generalize our RAM to any setting of parameters. To support m accesses, we require the

underlying storage to be an array of size $2m$. We organize this storage hierarchically into $\log(m) = \log(n) + i$ many levels, where the jth level holds 2^{j+1} elements. Physically, these layers can be laid out sequentially, with the ith level occupying indices $[2^i - 1, 2^{i+1} - 1)$. On each level, half of the elements are real elements that the user stored in the RAM, whereas the other half of the elements are what we call **dummy elements**. Dummy elements exist to help hide access patterns, in particular, they will help to hide on which level the accessed element resides in the storage, as we will see later. Each dummy is an all-zero string. When both servers access an element, it is important that the element be moved around in the storage, for otherwise the servers would learn access patterns [5]. Indeed, if elements do not move around in the storage, then accesses to the same physical address at time t and t' tell the server that the user wanted to access the same logical address at these time. To this end, we additionally allow the servers each to have a 2-element memory called the **stash**. When an element is accessed, they write the element back to the stash instead of wherever it was, alongside a dummy element.

Eviction policy. We observe that the stash becomes immediately full when an element is accessed. Thus, we must clear the stash and write it into the storage to make room for the next access. When we clear the stash, we always write the content of the stash to the top level, i.e., level 0, of our storage. This then also requires level 0, also only capable of holding 2 elements, to be cleared on every other access. We clear level 0 by writing its content onto level 1, which then requires level 1 to be empty on every fourth access by writing its content onto level 2. We call this process of clearing one level of storage and writing it to the next level **eviction**.

Notice how this process corresponds exactly to incrementing binary numbers: we treat the ith level as the ith bit of a binary number, with the 0th bit being the least significant bit. When we write an element to the storage, we always write it to level 0, and this corresponds to always adding 1 to the least significant bit, i.e., incrementing this binary number by 1. Under this interpretation, 0 indicates that the level still has vacancies, 1 indicates that the level is full, and a carry indicates an eviction has happened on the level that generated that carry. Thus, on the tth access, the binary expansion of t conveniently indicates the occupancy of all levels of the storage before any eviction. To implement the evictions for this access, we increment t and note down the bit positions that generated carries, then gather all the corresponding levels and write their contents onto the first non-carry-generating level. More explicitly, on the tth access, we look at the binary expansion of t and locate the lowest level that corresponds to a 0, gathering all contents from levels above, then writing them onto this level.

Example 3.1. Consider the 39th access, which in binary is $(100111)_2$. Since the 0th level of our storage corresponds to the least significant bit, the lowest level that corresponds to a 0 is the 3rd level, and thus we gather all the elements from levels $0, 1, 2$, and place them onto the 3rd level.

Permuting the storage levels. With stash write-back and the eviction process in place, we can ensure that an element keeps moving around in the storage, and accesses to the same physical address at different times do not necessarily mean that the user wanted the same element. However, at this point the servers can still infer access patterns if we are not careful about how we place elements on each level of the storage. Indeed, if we naïvely place the elements sequentially onto the levels, then just by looking at the position of an accessed element and calculating its offset from the beginning position of the storage, the servers will be able to tell exactly how long this element has been in the storage and exactly when was the last time this element was accessed. To avoid this problem, after each eviction, we ask all three parties to together randomly permute the level of storage just populated. As we will see, The permutation used in this step will be randomly and uniformly sampled by $\mathcal{D}$ and kept hidden from $\mathcal{S}_0$ and $\mathcal{S}_1$, and different random permutations will be used for each eviction. After the permutation, the positioning of elements onto the storage levels is no longer sequential, with the additional benefit that the servers now no longer knows which elements are dummies and which elements are real. With $\mathcal{D}$'s help, permuting the ith level can be done with communication cost $O(w2^i)$, and since the larger levels are permuted less frequently, the overall communication costs resulting from these permutations amortizes well over the number of accesses.

The timetable T. One of the new insights and most crucial components of our RAM is the timetable, which enables the servers to look up the physical address corresponding to a logical address without communicating with $\mathcal{D}$. With a permutation following each eviction, when $\mathcal{S}_0$ and $\mathcal{S}_1$ want to access an element, they have no idea where this element resides physically. While $\mathcal{D}$ also does not know which element resides where, our key insight is that $\mathcal{D}$ *does* know how elements move through levels of storage. Thus, $\mathcal{D}$ can help the servers search for the target physical address.

One obvious but insecure way is for $\mathcal{S}_0$ and $\mathcal{S}_1$ to send to $\mathcal{D}$ the time when this requested element was last accessed, and then ask $\mathcal{D}$ to calculate how this element would move through the storage and tell them where this element current resides. If we put aside security for a moment, then we can easily see that $\mathcal{D}$ has the knowledge to calculate where this element current resides, again because it knows at what time this element is evicted to which level

and where this element resides after permuting that level.

Unfortunately, sending $\mathcal{D}$ the time when an element was last accessed completely reveals access patterns to $\mathcal{D}$, and thus we must find other ways to ask for its help. The crucial observation is that how an element moves through the storage is completely independent of the identity of that element. Indeed, it depends only on the time when that element was lastly accessed. Thus, $\mathcal{D}$ can build a data structure that records how the tth accessed element would move through the storage, for each $t \in [0, m]$. It then carefully secret shares this data structure and sends the shares to $\mathcal{S}_0$ and $\mathcal{S}_1$, who can now refer to this data structure for the physical address of an element. We call this data structure the **timetable**, and denote it by $\mathbf{T}$. $\mathbf{T}$ is computed at the initialization stage of our RAM by $\mathcal{D}$, who then secret shares it between $\mathcal{S}_0$ and $\mathcal{S}_1$. $\mathbf{T}$ can be thought of as a table that implements the following mapping:

$$\mathbf{T} : [\![t']\!] \mapsto [\![((t_1, p_1), \ldots, (t_l, p_l))]\!], \tag{3.1}$$

where t' is the last time an element was accessed, and (t_i, p_i) means that from time t_i (inclusive) to t_{i+1} (exclusive), this element lived at physical address p_i. Here we used the notation $[\![a]\!]$ to denote a XOR secret share of a. We quite literally implement the timetable as a look-up table, where each **row** is a key-value pair of the above form. The timetable is stored "locally" in secret shared form by $\mathcal{S}_0$ and $\mathcal{S}_1$, in some random access memory structure that's not part of of our RAM. Protecting access patterns into these working memories is not a major concern, as we will see these access patterns are easily simulatable. We point out that the rows of the timetable must be permuted as well, for otherwise when $\mathcal{S}_0$ and $\mathcal{S}_1$ access a row t', they now learn that the currently requested element was lastly accessed at time t'. This permutation, π, is again uniformly and independently sampled by $\mathcal{D}$ and again kept hidden from $\mathcal{S}_0$ and $\mathcal{S}_1$.

The position map P. Since each row of $\mathbf{T}$ records how an element would move through the storage **after** the last time it was accessed, if $\mathcal{S}_0$ and $\mathcal{S}_1$ wants to retrieve the row associated with the currently accessed element, they must know when was its last access time. In other words, we need yet another data structure that maps secret shares of an logical address to secret shares of its last access time $[\![t']\!]$. In fact, since the timetable rows are permuted according to π, they need to know $\pi(t')$ for indexing the permuted timetable. We call this data structure the **position map**, and denote it by $\mathbf{P}$. $\mathbf{P}$ can be thought of as a table that implements the following mapping:

$$\mathbf{P} : [\![\alpha]\!] \mapsto [\![\pi(t')]\!], \tag{3.2}$$

where α is a logical address, t' is the last time α was accessed, and $\pi(\cdot)$ is the permutation on the rows of the timetable. We notice that the access patterns into $\mathbf{P}$ also **must** be protected or otherwise it exposes α, and also notice that $\mathbf{P}$ takes secret shares of an logical address and maps it to some secret shared output. This data structure is thus required to achieve exactly the same functionality as our RAM, and has exactly the same semantics as our RAM. Therefore, perhaps not surprisingly, we will implement $\mathbf{P}$ recursively as another instance of our secure RAM, where we explicitly force each recursively instantiated RAM to have half the size of the last, so that the recursion will terminate in $\log(n)$ steps.

Tying it all together. We are now finally ready to explain how to execute a logical access in the basic non-parallel RAM. To access a secret shared logical index $[\![\alpha]\!]$ at time t, $\mathcal{S}_0$ and $\mathcal{S}_1$ first query the position map $\mathbf{P}$ with the exact same $[\![\alpha]\!]$, from which they get out $[\![\pi(t')]\!]$, with t' being the last access time of α. They then reveal this $\pi(t')$ to themselves and locally use this to read out their share of the associated row from the timetable. They obliviously scan through this row, looking for the entry $[\![(t_i, p_i)]\!]$ so that t_i is the largest time $\leq$ the current time. p_i is thus the physical address of α. However, it is not yet safe to for them to reveal p_i as it tells them whether α was recently accessed. Indeed, if α was recently accessed, p_i would be in the top levels of the storage, but otherwise it would be in the lower levels. The dummies we stored on each level can help us avoid this problem. The servers will ask $\mathcal{D}$ to send over a secret shared list of dummy physical addresses, sampled one per level and without replacement.

They then obliviously use p_i to replace the dummy address on the same level. After the replacement, they would finally reveal to each other this list of addresses, and the dummy addresses help to hide which level p_i was on. Notice that this means that the servers see exactly one address per level, regardless of which level the target memory element is actually on. Furthermore, each revealed address is uniformly random (without replacement) within its level, a fact that is ensured by permutations on RAM levels.

The servers proceed to read the storage using this list of addresses, what they get back are secret shares of a list of dummies, except for one being the real element at logical address α. Since the dummies are all-zero strings, $\mathcal{S}_0$ and $\mathcal{S}_1$ can locally XOR together all the shares in their list. at the end of which they each hold shares of the element at logical address α. (Note, this revealing of one element per level is similar to the Garbled RAM technique of [32].)

Now that the servers have accessed element α, they must prepare for the next time α is accessed. To do so, they ask $\mathcal{D}$ to send over $[\![\pi(t)]\!]$ where t is the current time, and overwrites $\mathbf{P}[\alpha]$ with this new value. This way, when α is accessed again in the future, the position

map stores the current time, which is indeed the last access time of α. Finally, $\mathcal{S}_0$ and $\mathcal{S}_1$ checks the binary representation of t and determine the levels that should be evicted. They gather the elements from these levels, place them onto the next level, and permutes the next level with the help of $\mathcal{D}$. This concludes this access of the RAM.

Initialization. Before our RAM starts to work, we want to make sure that we have stored all the elements the user passed to our RAM and that we have properly initialized all components of our RAM. This includes the storage, the timetable, but more importantly, the position map.

Recall that in the position map, $\mathbf{P}[\alpha]$ is the last access time of logical index α, but at the beginning of our protocol, α was never accessed before. Therefore, we must manually access each element at the beginning of the protocol, so that we have something to store in the position map. We do this by requiring that, after the user has passed us all the elements to store and before we allow the user to use our RAM, $\mathcal{S}_0$ and $\mathcal{S}_1$ spend n time steps writing these n elements to our storage. $\mathcal{S}_0$ and $\mathcal{S}_1$ will write all these elements to level $\log(n)$ of the storage, which is just enough to hold these elements along with n dummies. All parties then permute this level together.

$\mathcal{D}$ would add n more rows to the timetable, each associated with a time step $t \in [1, n]$. These rows, as usual, record how an element will move through the storage after it's been written at time t, though this time $\mathcal{D}$ keeps in mind that these elements started at level $\log(n)$ instead of level 0. $\mathcal{S}_0$ and $\mathcal{S}_1$ then ask for $\pi(t)$ from $\mathcal{D}$ and pass these to the top-level position map $\mathbf{P}$ in the same way that the user passed our RAM the elements they want to store. The recursive position maps are then initialized in the same way. From $\mathcal{S}_0$ and $\mathcal{S}_1$'s perspective, the access pattern of the initialization step is simulatable by just sequentially writing random elements to the first n positions on level $\log(n)$ of our storage.

From $\mathcal{D}$'s perspective, it receives inquiries about $\pi(t)$ for $t \in [1, n]$, and this is trivially simulatable. We point out that although $\mathcal{D}$ knows the physical address corresponding to each logical address immediately after the initialization, this information soon becomes stale since $\mathcal{D}$ does not know which element will be accessed when.

Refreshing. To ensure the storage size remains bounded, we limit each instance of RAM to only support m queries. However, we anticipate the need of an user to issue more than m queries into our RAM. We use k to denote the number of queries the user issue, and explain how to handle the case where $k > m$. In this case, a single instance of our RAM is not sufficient for these queries, as it simply does not have enough storage for eviction. Therefore, we must instantiate another fresh RAM and copy the contents of this RAM

over, the new RAM can support another m queries. We call this process **refreshing**, and by constantly refreshing our RAM, we will be able to serve all k queries the user issued. Refreshing consists of two stages, where stage one is reading the elements out of the current RAM and stage two is copying these elements to a new RAM.

The second stage can be easily implemented by handing the elements we read from stage one to a newly instantiated RAM, in much the same way the user handed these elements to the first instance of our RAM, and the newly instantiated RAM proceeds as described in the initialization section above.

To implement the first stage, $\mathcal{S}_0$ and $\mathcal{S}_1$ will access logical addresses $[1, n]$ sequentially. They access these logical addresses by following the same process as a normal access, although there is no need to enforce the eviction policy anymore as nothing is written back to the storage. Also, since m is a power of 2, at the end of m queries, all elements will be stored at the lowest level of our storage, and as such there is no need to hide the level information of a physical address. That is, during this stage, the servers will not ask from lists of dummy addresses from $\mathcal{D}$, saving us some communications.

Notations. The notations we adopt throughout this work is listed in Table 3.1.

$\mathcal{S}_0, \mathcal{S}_1$	the servers
$\mathcal{D}$	the dealer
n	the capacity of our RAM, i.e. number of elements stored
m	the maximum number of accesses that an instance of our RAM supports
k	the number of accesses that our RAM needs to serve
w	the word size of our RAM
p	the number of processors each server has
threshold	the threshold for terminating position map recursion, usually $= p$
$[\![x]\!]$	XOR secret share of x
$(x_0, x_1) = [\![x]\!]$	x is secret shared into x_0 and x_1 with $\mathcal{S}_i$ holding x_i
Π	a general permutation
π	the permutation on the rows of the timetable
$[n]$	the set of integers $\{1, \ldots, n\}$
$[i, j]$	the set of integers $\{i, \ldots, j\}$
$[i, j)$	the set of integers $\{i, \ldots, j-1\}$
$\mathbf{T}$	the timetable

Table 3.1: List of notations

l	number of entries in a row of the timetable
$\mathbf{P}_i$ for $i \geq 0$	the ith level position map, where the top level one is also denoted $\mathbf{P}$
$A[x]$	the xth entry of an indexable structure A (table, array, RAM, tuple, etc.)
A_x	an alternative way of writing the above
$\mathcal{Q}$	a vector of queries $(q_0, \ldots, q_p)$
Q	a sequence of parallel queries
$\mathcal{R}$	a vector of query results $(r_0, \ldots, r_p)$
λ	the empty string, indicating a party has no input or output
$\mathcal{F}_{\mathsf{func}}$	a functionality named func
Π_{func}	a secure protocol for computing $\mathcal{F}_{\mathsf{func}}$
χ_q	the one-hot encoding of $q \in \mathcal{Q}$
$\chi_{\mathcal{Q}}$	the characteristic string associated with $\mathcal{Q}$

Table 3.1: Continued from previous page

When we append a subscript i to a parameter of our RAM, we mean that it's the same parameter for the ith level position map. For example, n_i is the capacity of the ith level position map.

CHAPTER 4: FORMAL APPROACH

In this chapter, we explain in detail the various building blocks of our protocol. Starting from the permutation functionality that underlies much of our security guarantees in section 4.1, we gradually build towards a secure PRAM that is capable of handling parallel queries in 4.4.

To improve the organization of our content, we will treat non-conflicting and conflicting queries as distinct cases. However, we want to emphasize that in practice, these cases are not truly separate. Ensuring that the views in these two cases are indistinguishable is critical for security reasons. If we were to handle $\mathcal{Q}$ differently depending on whether it has a conflict, the resulting views would be easily distinguishable. Therefore, we want to draw the reader's attention to the fact that, when handling parallel queries, $\mathcal{Q}$ always goes through the same workflow as described in section 4.4.4, regardless of collision. In the case that $\mathcal{Q}$ is non-flicting, the additional steps we perform on it has no effect.

4.1 PERMUTATION OF AN ARRAY

Permutation is a fundamental building block of our protocol that underlies much of our security guarantees. Indeed, we permute the storage levels to hide which elements are real and which are dummies, we permute the timetable rows so that the servers don't learn the last access time of a particular element, and as we will see later, when handling parallel queries, we follow the shuffling-before-sort paradigm in [33] and permute the query vector $\mathcal{Q}$ before sorting it so that the ensuing sorting does not reveal ordinal relations among queries in the original $\mathcal{Q}$.

In this section, we will look at 2 variants of the same permutation protocol. The first variant is less efficient but perfectly secure, whereas the second one is more efficient but only computationally secure. Concretely, to permute an array of n w-bit elements, the first variant has communication complexity $6wn + 2n\log(n)$ and rounds complexity 8, whereas the second variant has communication complexity $3wn$ and rounds complexity 3, but relies on computationally secure one-way functions.

4.1.1 The perfectly secure but less efficient permutation protocol

As we already pointed out, permutation is a frequently computed functionality in our protocol, and thus it is crucial for efficiency that we employ an efficient and secure 3PC

Functionality 4.1: $\mathcal{F}_{\mathsf{perm}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a n w-bit element array $(A_0, A_1) = [\![A]\!]$. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs A from A_0, A_1.
2. Samples a random permutation Π uniformly and randomly from the set of permutations on $[n]$.
3. Secret shares $(\Pi(A)_0, \Pi(A)_1) = [\![A]\!]$ and sends them to $\mathcal{S}_0, \mathcal{S}_1$, respectively. Sends Π to $\mathcal{D}$.
4. $\mathcal{S}_0, \mathcal{S}_1$ outputs $\Pi(A)_0, \Pi(A)_1$, respectively, and $\mathcal{D}$ outputs Π.

Figure 4.1: The ideal functionality $\mathcal{F}_{\mathsf{perm}}$.

permutation protocol. We adopt the resharing-based permutation protocol from [34] and [35] and slightly tailor it to our setting. Formally, we want to compute the ideal functionality $\mathcal{F}_{\mathsf{perm}}$ listed in Figure 4.1.

The protocol Π_{perm} that we use to compute $\mathcal{F}_{\mathsf{perm}}$ is listed in Algorithm 4.1. In this permutation protocol, we will utilize the reshare functionality $\mathcal{F}_{\mathsf{reshare}}$ which operates on secret shares and "rerandomizes" them. Specifically, Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ holding secret shares $(x_0, x_1) = [\![x]\!]$ participate in $\mathcal{F}_{\mathsf{reshare}}$ with their own share, and $\mathcal{F}_{\mathsf{reshare}}$ delivers new shares $(x'_0, x'_1) = [\![x]\!]$ to $\mathcal{P}_0$ and $\mathcal{P}_1$. Usually, x'_0 and x'_1 is required to look uniformly random to $\mathcal{P}_0$ and $\mathcal{P}_1$. However, for our application, it suffices to use a resharing scheme where $\mathcal{P}_0$ and $\mathcal{P}_1$ just agree on a random string and both XOR it to their existing shares, and this is indeed the reshare we use in Π_{perm}. In the perfectly secure setting, we make the arbitrary choice that the first party always sends this random string to the second party, except the resharing in line 1 where this is reversed (so that line 2 is simulatable for $\mathcal{D}$). To avoid complicated notation, we denote both the original shares and the new shares by $(x_0, x_1) = [x]$, but it should be understood that after an execution of $\mathcal{F}_{\mathsf{reshare}}$, the shares of the parties necessarily change. Furthermore, whenever $\mathcal{D}$ and $\mathcal{S}_i$ together sample a random permutation in Π_{perm} (such as on line 4), it suffices for $\mathcal{D}$ to just select a random permutation locally and send it to $\mathcal{S}_i$.

Intuitively, this protocol is secure because $\mathcal{D}$ never receives anything related to $\mathcal{S}_1$'s input A_1 and as such it does not learn A_1, nor does it learn A_0 because the first resharing on line 1 effectively hides it. The second resharing on line 3 prevents $\mathcal{S}_0$ from learning anything about A_1, and the third resharing on line 6 prevents $\mathcal{S}_1$ from learning anything about Π_0, from which it can obtain $\Pi = \Pi_0 \circ \Pi_1$.

Algorithm 4.1: $\Pi_{\mathsf{perm}}(A_0, A_1, \lambda)$

Input: A_0, A_1, λ from $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$, respectively

Output: $\Pi(A)_0, \Pi(A)_1, \Pi$ for $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$, respectively

1. $\mathcal{S}_0$ and $\mathcal{S}_1$ reshare $[\![A]\!]$
2. S_0 sends A_0 to $\mathcal{D}$
3. $\mathcal{S}_1$ and $\mathcal{D}$ reshare $[\![A]\!]$
4. $\mathcal{D}$ and $\mathcal{S}_1$ together samples Π_1
5. $\mathcal{S}_1$ sends $\Pi_1(A_1)$ to $\mathcal{S}_0$
6. $\mathcal{S}_0$ and $\mathcal{D}$ reshare $[\![\Pi_1(A_1)]\!]$
7. $\mathcal{D}$ and $\mathcal{S}_0$ together samples Π_0
8. $\mathcal{D}$ sends $\Pi_0(\Pi_1(A_0))$ to $\mathcal{S}_1$
9. Let $\Pi = \Pi_1 \circ \Pi_0$
10. $\mathcal{S}_0, \mathcal{S}_1$ outputs $[\![\Pi(A)]\!]$
11. $\mathcal{D}$ outputs Π

4.1.2 The computationally secure but more efficient permutation protocol

The current implementation of Π_{perm} does not achieve optimal communication and rounds complexity. The pseudocode in Algorithm 4.1 requires eight rounds of communications between the parties, including 3 rounds for the resharings on lines 1, 3, and 6, and 2 rounds for sampling the permutations on lines 4 and 7. These communications along contribute $3wn + 2n\log(n)$ to the overall communication complexity and 5 to the overall rounds complexity. However, if we assume the existence of one-way functions and aim for computational security only, we can optimize Π_{perm} and eliminate these communications. This reduces the overall communication complexity to exactly $3wn$ and the rounds complexity to exactly 3.

The key insights behind saving these communications is that the goals they want to achieve can be easily achieved if the communicating parties can agree on a common random string. Indeed, if this is the case, resharing can be achieved by asking both parties to XOR their common random string to their shares, and sampling a permutation can be achieved by interpreting their common random string as a permutation. To this end, we require that at the beginning of the protocol, $\mathcal{D}$ agrees with $\mathcal{S}_0$ on a **stateful pseudorandom generator** (SPRG) [36] G_0 and a seed s_0, and agrees with $\mathcal{S}_1$ on a different SPRG G_1 and a seed s_1. Both SPRGs should be capable of handling $5m$ invocations. Whenever $\mathcal{D}$ and $\mathcal{S}_i$ need to agree on a common random string, they can locally query the SPRGs they previously agreed on, which return the same result to both parties. We can select the SPRGs G_0 and G_1 so that their outputs are $\max(wn, n\log(n))$ bits, and discard the redundant bits in the output for use in the shorter scenario. These modifications to Π_{perm} do not obstruct $\mathcal{D}$'s ability to construct the timetable, as $\mathcal{D}$ can perform lookahead into the SPRGs during construction.

Later, when the protocol starts running, $\mathcal{D}$ will act consistently with the lookahead values it already obtained. Although the security of the modified Π_{perm} is less obvious, formal simulation proofs for both the original and modified Π_{perm} will appear in Chapter 5.

4.2 THE TIMETABLE

The timetable $\mathbf{T}$ is a data structure that maps a time step t to the list of positions that the element accessed at time t would move to. It's stored directly in secret shared form on both servers' working memory, which is independent of our RAM. To prevent the servers from learning when the current element was last accessed, the rows of the timetable are permuted such that the servers cannot determine which timestep each row corresponds to.

In this section, we will examine the workflow involved in using the timetable. When accessing the relevant row for their current target element α, $\mathcal{S}_0$ and $\mathcal{S}_1$ must extract the current physical address of α through a two-step process. Firstly, they need to scan through the row to find the entry that records the current physical address. Secondly, to hide on which level this physical address is, they need to request a list of dummy addresses from $\mathcal{D}$ and replace the dummy address at the same level as α with its physical address.

We will start with a review of timetable's format and size, and then move on to the two-step process in section 4.2.2.

4.2.1 Format and size of the timetable

According to our storage structure and eviction policy, when an element is accessed at time t and written back to the stash, it gradually moves towards the lower levels of storage. In principle, an element can move through all levels of the storage and reach the lowest level, even though with at most m allowed accesses to one instance of our RAM, some elements written later may never reach certain levels. To make these two cases look indistinguishable in the timetable, we require that all rows contain the same number of entries as the levels in the storage, which is $\log(m) = \log(n) + i$. For the rows where the element can't reach a certain level, $\mathcal{D}$ appends random entries to the end to fill them up.

Each entry of a row is of the form (t_i, p_i), indicating that at time t_i, the element moved to position p_i in the RAM. If an element is accessed again, it is written back to the stash and starts from level 0 of the storage, rendering the rest of the row inaccurate. Therefore, the tth row of the timetable describes how the element accessed at time t will move through the storage **up to the point it is accessed again**.

The timetable needs to have $m + n$ rows since one instance of RAM supports up to m accesses and we need the additional n rows for initialization. Each entry (t_i, p_i) takes $\log(m) + \log(2m) = 2\log(m) + 1$ bits to represent, and each row contains $\log(m)$ entries. Therefore, the entire timetable takes $(m + n)(2\log(m) + 1)\log(m) = O(n\log^2(n))$ bits to represent. This incurs a communication cost of $O(n\log^2(n))$ when the dealer initially sends the timetable to the servers. However, when amortized over $k = O(n)$ many accesses, this cost becomes $O(\log^2(n))$.

4.2.2 Extracting current physical address from a row

When $\mathcal{S}_0$ and $\mathcal{S}_1$ want to determine the physical address of the element α they are currently accessing, they first consult the position map to find $\pi(t')$, where t' is the last access time of α. $\pi(t')$ serves as an index into the timetable, and each server retrieves shares of the corresponding row locally. They then read this row and locate the correct entry, which is the entry with the largest $t_i \leq$ the current time t. Throughout this step, it is critical that the servers do not reveal the time step or physical address of any other entry, as such information could lead to partial discovery of access patterns, compromising security.

Example 4.1. Suppose the current time step is 39, and the relevant timetable row is $[\![(0,4),(2,5),(4,9),(8,13),(16,27),(32,51),(64,99)]\!]$, then the entry with the largest t_i still ≤ 39 is $(32,51)$, which indicates that currently the element is at physical address 51. Therefore, after the first step, the servers output $[\![51]\!]$.

In the next step, the servers request that $\mathcal{D}$ provide a list of physical addresses for dummies. These dummy addresses are uniformly and randomly sampled without replacement, with one address selected from each level. $\mathcal{S}_0$ and $\mathcal{S}_1$ then scan through this list of addresses, replacing the dummy address on the same level as α with α's physical address. It is important to note that this sampling without replacement is crucial for security. If the same position is accessed across two closely-timed queries, $\mathcal{S}_0$ and $\mathcal{S}_1$ could infer that a dummy is being accessed rather than a real element. This is because real elements are written back to the stash and start from level 0, while dummies remain in place until the next eviction. This in turn reveals partial information about on which level the accessed element resided, and thus compromising security.

Example 4.2. Suppose the list of physical addresses $\mathcal{D}$ shared is $[\![(1,4,13,25,57,106)]\!]$ and the physical address for α is 51, then 57 is the dummy on the same level as α. Therefore, after the second step, the servers output $[\![(1,4,13,25,51,106)]\!]$.

Functionality 4.2: $\mathcal{F}_{\mathsf{binsearch}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share an ordered l element array $(A_0, A_1) = [\![A]\!]$ and a search key $(k_0, k_1) = [\![k]\!]$. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs A from A_0, A_1 and k from k_0, k_1.
2. Runs binary search on A with search key k. Let the resulting element be $a = \max_{i \in [0, l-1]}\{a_i \leq k\}$.
3. Secret shares $(a_0, a_1) = [\![a]\!]$ and sends them to $\mathcal{S}_0, \mathcal{S}_1$, respectively.
4. $\mathcal{S}_0, \mathcal{S}_1$ outputs a_0, a_1, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.2: The ideal functionality $\mathcal{F}_{\mathsf{binsearch}}$.

We notice that both steps are essentially searching through a list of ordered elements with a search key, where in the first step the search key is the current time step and in the second step the search key is α's physical address. As such, we will use an efficient binary search protocol to compute both steps. Formally, we want to compute the ideal functionality $\mathcal{F}_{\mathsf{binsearch}}$ listed in Figure 4.2.

The specific protocol we use to compute $\mathcal{F}_{\mathsf{binsearch}}$ is adapted from the rotation-based binary search (RotBS) protocol in [37]. For convenience, we will make the simplifying assumption that we are searching through an array A whose length l is a power of 2, however, we can easily generalize this to any array by padding it with random elements with large keys. For arrays of such length, there is an unique perfect binary search tree, where the root is $A[\lfloor l/2 \rfloor + 1]$, the left child of the root is $A[\lfloor l/4 \rfloor + 1]$, and the right child of the root is $A[\lfloor 3l/4 \rfloor + 1]$, and so on. Performing binary search on A is then equivalent to traversing this binary tree from the root to the last level. The problem with a naïve binary search is that it will completely expose the search path through this tree, and RotBS addresses this problem by randomly permuting each level of the tree. This way, whenever the servers access a node, they no longer know which node it originally was, and thus the search path is hidden to them.

Taking this idea one step further, we observe that for applications where we search through A only once, such as the two applications we consider, we can randomly swap the left and right child of a node to achieve the same effect as permuting the entire level. In both schemes, the search path exposed to the servers is randomly and uniformly distributed among all possible paths. Our slight modification significantly improves the efficiency of the binary search protocol, since a conditional swap of two children is much cheaper to implement than adding a rotation offset and taking a modulus, as is required in [37]. In fact, since $\mathcal{S}_0$ and $\mathcal{S}_1$ only search through A once and as such only expose one search path, we can even use

the same swap decision for all nodes on the same level. These swap bits are sampled by $\mathcal{D}$ randomly and sent to the servers, in addition, $\mathcal{D}$ will also order A as a flat binary search tree to facilitate the search procedure.

Our protocol for computing the binary search, $\Pi_{\mathsf{binsearch}}$, is listed in Algorithm 4.2. All the computations in $\Pi_{\mathsf{binsearch}}$ are implemented as circuits, except for revealing d on line 6 and accessing a_d on line 7. In this pseudocode, $\mathsf{compare}([\![x]\!], [\![y]\!])$ realizes the binary operator $<$, which returns 1 if $x < y$ and 0 otherwise.

Algorithm 4.2: $\Pi_{\mathsf{binsearch}}((A_0, B_0, k_0), (A_1, B_1, k_1), \lambda)$

Input: $\mathcal{S}_0$ and $\mathcal{S}_1$ hold secret shares of $(A_0, A_1) = [\![a_0, \dots, a_{l-1}]\!]$, already ordered as a binary search tree. They also holds secret shares of the swap bits $(B_0, B_1) = [\![b_0, \dots, b_{\log(l)-1}]\!]$ and the search key $(k_0, k_1) = [\![k]\!]$. $\mathcal{D}$ has no input.

Output: $[\![a_i]\!]$ where $i = \max_{j \in [0, l-1]}\{a_j \leq k\}$.

1. $[\![p]\!] = [\![a_0]\!]$ `// the element to return`
2. $[\![d]\!] = 0$ `// the current index in` A
3. $[\![c]\!] = \mathsf{compare}([\![k]\!], [\![a_0]\!])$ `// the current comparison result`
4. **for** $i = 1, \dots, \log(l) - 1$ **do**
5. $\quad [\![d]\!] = ([\![2d + 1 + c]\!]) \cdot b_i \oplus ([\![2d + c]\!]) \cdot (\neg b_i)$ `// calculating next index`
6. $\quad \mathcal{S}_0$ and $\mathcal{S}_1$ reveal d
7. $\quad [\![a]\!] = [\![a_d]\!]$ `// the current element`
8. $\quad [\![c]\!] = \mathsf{compare}([\![k]\!], [\![a]\!])$
9. $\quad [\![p]\!] = [\![c]\!] \cdot ([\![a]\!] \oplus [\![p]\!]) \oplus [\![p]\!]$ `// replace` $[\![p]\!]$ `by` $[\![a]\!]$ `if` $a < k$
10. **return** $[p]$

Remark 4.1. As a side note on implementation, we remark that the calculation of the next index to visit on line 5 can be hard-coded into the circuit. Indeed, it is a function determined by two bits c and b_i, and its "truth table" is given in Table 4.1. By hard-coding this table as a constant size and constant depth circuit, we avoid the expensive additions otherwise required, and this is a major efficiency improvement over the basic RotBS in [37].

b_i	c	next index
0	0	$2d + 2$
0	1	$2d + 1$
1	0	$2d + 1$
1	1	$2d + 2$

Table 4.1: The "truth table" for calculating next index in $\Pi_{\mathsf{binsearch}}$.

With our optimization, the majority of computation required is the $\log(l)$ sequential comparisons. We will soon show how to implement comparison on w-bit strings with $O(w)$ communication complexity and $O(\log(w))$ rounds complexity. But taking them as granted for now, we conclude that computing the binary search on an array of l w-bit elements incurs $O(w\log(l))$ communication cost and requires $O(\log(w)\log(l))$ rounds. In both of our applications, $l = O(\log(n)) = w$, and so computing the binary search incurs $O(\log(n)\log\log(n))$ communication cost and requires $O((\log\log(n))^2)$ rounds.

4.3 THE POSITION MAPS

As a reminder, when reading the timetable to determine the current position of an element, the servers must have access to information about when the element was last accessed. However, storing this information in plain text would reveal access patterns, so the servers must instead store secret shares of this information in a data structure. We require this data structure to be randomly accessible when provided with secret shares of a logical address. Our RAM is ideal for this purpose: its contents are secret shared between the servers and it supports random access when provided with secret shares of a logical address. As a result, we can recursively implement the position map as another instance of our RAM.

4.3.1 The recursive case

To ensure that the recursion process stops, we need to reduce the number of elements in each position map as we go down the recursion. We accomplish this by "packing" two entries of a position map into one entry when creating the next level position map. Let $\mathbf{P} = \mathbf{P}_0$ be the top-level position map that serves the RAM directly. The number of elements in this position map is the same as that of our RAM, which is n. However, as we create the next position map $\mathbf{P}_1$ to serve $\mathbf{P}_0$, we want to ensure that it only stores $n/2$ elements. To achieve this, for any two consecutive logical addresses $(\alpha_{2i}, \alpha_{2i+1})$, where $i \in [0, \lfloor n/2 \rfloor]$, we store the last access times of each of them as **a single entry** in $\mathbf{P}_1$. In other words, $P_1[i] = (t'_{2i}, t'_{2i+1})$, where t'_{2i} is the last time α_{2i} was accessed, and t'_{2i+1} is the last time α_{2i+1} was accessed. This policy applies to every $\mathbf{P}_i$ as long as $i \geq 1$. We also require that m is halved each time we recurse, so that the relation $n = 2^i m$ is maintained. It may seem that packing two entries into one entry in the position map would exponentially increase the word size w of the recursive position maps. However, this is not the case. In reality, the size of the entries in the position maps only depend on the maximum number of accesses m that an instance of the RAM can support. As we halve m from P_i to P_{i+1}, we have

$w_{i+1} = 2\log_2(m_{i+1}) = 2(\log_2(m_i) - 1) = 2\log_2(m_i) - 2 = w_{i-1} - 2$. Thus, the words become two bits shorter from P_i to P_{i+1}.

It is important to keep in mind that accessing the main RAM requires accessing all of the position maps $\mathbf{P}_i$ as well. Therefore, while n is halved and w is reduced by 2 at each recursion, the lower level position maps still need to handle the same number of accesses as the main RAM. As m is also halved, the number of position map instances increases exponentially with each level of recursion. This means that at some point, the workload required to refresh these position maps can become overwhelming, severely impacting the efficiency of our RAM. To avoid this issue, we need to set a threshold value threshold, such that once the number of elements stored in a position map drops below this threshold, we terminate the recursion and use a different technique to manually serve the queries.

4.3.2 The base case

As previously observed, lower-level position map work is dominated by refreshing. When the size of the position map drops below a certain threshold that can be fine-tuned, we switch to explicit query handling. At this point, the base case position map no longer has the same storage structure as the recursive ones, nor does it have the same components. We explicitly eliminate the hierarchical storage structure, the eviction policy, the timetable, and obviously the position map.

Instead, to hide access patterns, we read every element stored in this position map when a query reaches it, and linearly scan through the results for the queried element. This ensures that the access pattern is easily simulated and security is guaranteed. This simple scheme works for non-parallel RAM since there is only one query and one linear scan through the contents suffices. However, when we upgrade our RAM to PRAM, the base case position map must now serve p queries, and performance will be severely impacted if we naïvely scan through the contents of the base case position map p times. As the result, a different approach is necessary, and we will elaborate on it in the following section.

4.4 UPGRADING RAM TO PRAM

We have successfully developed a basic RAM that allows index-oblivious querying by the user. Compared to traditional ORAMs that require the query index to be given in clear text to the compiler, our RAM is already more powerful. Indeed, a traditional ORAM can be obtained from our RAM by asking $\mathcal{S}_0$ and $\mathcal{S}_1$ to exchange shares of the queried logical address to reconstruct it in clear text. Where our PRAM truly stands out is its ability

to handle queries in parallel. In the current setting, we imagine there are p users of our PRAM, and they each issue a query into our PRAM. We collect the queries into a query vector $\mathcal{Q} = [(q_0, \ldots, q_{p-1})]$ where the ith user issued the ith query and each q_i is a logical address $\in [0, n-p)$. $\mathcal{Q}$ could contain duplicate queries, that is, for $i \neq j \in [0, p-1]$, we might have $q_i = q_j$. We say that in this case, we have a **conflict**. Our challenge is to ensure that the physical access patterns and parties' views look indistinguishable regardless of conflicts. To handle queries in parallel, we need to leverage the parallelism we have and make modifications to some of our components. Fortunately, we will only need to change the storage structure, the base case position map, and find a new way to interpret the timetable.

4.4.1 New storage structure

The first and most straightforward modification to our existing RAM is to increase the size of storage levels to accommodate the simultaneous access of p elements. We allow each processor to have its own stash of size 2, which means we need to store $2p$ elements in the 0th level of our storage when we clear the stashes. This change requires a proportional increase in the size of all levels in our storage by a factor of p. However, this does not increase the total size of our storage, as we now need fewer levels. The eviction policy also changes, as the parties must now permute levels of increased size. Nonetheless, our binary number increment interpretation of the eviction policy still applies, since we can now treat p elements as a single unit and always write a unit of elements at each epoch. We generalize the notion of a time step to an **epoch**, calling the servicing of p parallel queries as one epoch. When $p = 1$, an epoch reduces to a time step. The epoch counter conveniently indicates the occupancy of our storage levels, with the ith bit of this counter corresponding to the ith level of our storage and the least significant bit representing the 0th level.

4.4.2 New semantics of the timetable

The timetable used to store how elements accessed at a particular time step would move around in the storage. However, with the generalization of a time step to an epoch, the semantics of the timetable must change. To this end, we arbitrarily stipulate that during the ith epoch, the element requested by q_j is accessed at "time step" $ip + j$, for $j \in [1, p]$. This does not mean that we are executing the queries sequentially; rather, it means that when $\mathcal{D}$ generates the timetable, it has a consistent way to assign different rows for the elements accessed in the same epoch. Additionally, when $\mathcal{S}_0$ and $\mathcal{S}_1$ later ask for $\pi(\alpha)$ from $\mathcal{D}$, which is essentially the "last access time" of an element, the notion of "last access time"

is now well-defined. The rest of the timetable remains the same. $\mathcal{D}$ still adds n rows to it during initialization, knows how to generate the contents of each row since it samples all the permutations to be used in evictions a priori, and can still order the rows in a way that facilitates binary search.

4.4.3 Handling non-conflicting queries

When $\mathcal{Q}$ consists of p distinct queries, our RAM with modified storage structure already suffices. Indeed, since the p queries are all distinct, $\mathcal{S}_0$ and $\mathcal{S}_1$ can assign the ith query to the ith processor. Each of these processors will fetch distinct rows of the timetable, $\mathcal{D}$ will sends distinct lists of dummy addresses, sampled without replacement on each level so that no two queries will access the same elements on the same level. This sampling without replacement is again crucial for security for the following reason: the queries themselves are distinct, so this implies that the physical addresses corresponding to these queries are also distinct. Therefore, if $\mathcal{S}_0$ and $\mathcal{S}_1$ observed a position accessed across different processors, they can infer that this position stores a dummy instead of a real element, and now they have gained partial information about which level of storage the elements are on.

Following the physical accesses, $\mathcal{S}_0$ and $\mathcal{S}_1$ will ask for $\pi(t)$ from the $\mathcal{D}$, interpreting t for each element in the way we previously mentioned, and store these entries into the top-level position map. Even though the entries passed to the position map are still distinct at this level, but as a result of the way we enforced recursion termination by manually packing two entries into one in the recursive position maps, there will be higher chance of collision as we move from $\mathbf{P}_i$ to $\mathbf{P}_{i+1}$. In fact, when we move from $\mathbf{P}_i$ to $\mathbf{P}_{j+1}$ with $\mathcal{Q}_i$ becoming $\mathcal{Q}_{i+1}$, the pairwise distance of logical addresses queried in $\mathcal{Q}_{i+1}$ is 1/2 of its counterpart in $\mathcal{Q}_i$. For example, suppose $\mathcal{Q} = \mathcal{Q}_0 = [(2, 6, 5, 3, 7, 0, 1, 4)]$, then in $\mathbf{P}_1$ we have $\mathcal{Q}_1 = [(1, 3, 2, 1, 3, 0, 0, 2)]$, in $\mathbf{P}_2$ we have $\mathcal{Q}_2 = [(0, 1, 1, 0, 1, 0, 0, 1)]$, and finally in $\mathcal{Q}_3$ all the queries are to the same logical address 0. In general, when we move from $\mathcal{Q}_i$ to Q_{i+j}, we see that $|Q_{i+j}[x] - Q_{i+j}[y]| = 2^{-j}|Q_i[x] - Q_i[y]|$, and as such the collision chance increases exponentially as we move down the recursion.

4.4.4 Handling conflicting queries

In this case, $\mathcal{Q}$ contains at least one pair of queries to the same logical address, either as a result of two users wanting to access the same elements, or as a result of moving down the recursion. We want to ensure that the access patterns and parties' views looks indistinguishable from the non-conflicting case. Our idea is to resolve the conflicts in $\mathcal{Q}$,

Functionality 4.3: $\mathcal{F}_{\mathsf{dedupe}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a query vector $(Q_0, Q_1) = [\![Q]\!]$. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs Q from Q_0, Q_1.
2. Initialize a p-bit string $b = 0^p$.
3. Scan through Q, for each pair of indices such that $q_i = q_{i+1}$, set $b_{i+1} = 1$ and $q_{i+1} = n - p + i + 1$.
4. Secret shares $\mathcal{Q}$ and b with $\mathcal{S}_0$ and $\mathcal{S}_1$, $(\mathcal{Q}_0, \mathcal{Q}_1) = [\![\mathcal{Q}]\!], (b_0, b_1) = [\![b]\!]$.
5. $\mathcal{S}_0, \mathcal{S}_1$ output $(\mathcal{Q}_0, b_0), (\mathcal{Q}_1, b_1)$, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.3: The ideal functionality $\mathcal{F}_{\mathsf{dedupe}}$.

query for the elements as in the non-conflicting case, and finally post-process the results so that each processor gets back the element it wanted.

The conflict resolution begins with sorting $\mathcal{Q}$, so that future processing on $\mathcal{Q}$ is more efficient. Our sorting method follows the general **shuffling-before-sorting** paradigm proposed in [33]. The basic idea behind this paradigm starts with the observation that normal comparison sorting algorithms is not secure because they reveal the ordinal information of compared elements. However, if we randomly permute the elements prior to sorting, the ordinal information revealed during the comparisons no longer correlates to where the elements truly are.

Therefore, we first ask all three parties to permute $\mathcal{Q}$ together, and then let $\mathcal{S}_0$ and $\mathcal{S}_1$ run a revealing Batcher sorting network [38], in which the choices for swapping pairs of elements are made public. This way, both servers can later undo the sorting locally by running the Batcher network backward and reversing these choices. We emphasize that this shuffling-before-sorting step is only performed in the top-level RAM, as the queries induced in the recursive position maps will always be sorted moving forward.

After sorting $\mathcal{Q}$, $\mathcal{S}_0$ and $\mathcal{S}_1$ will deduplicate it to obtain a new query vector $\mathcal{Q}'$ where the duplicate queries are replaced by fake ones, and then serve these queries as in the non-conflicting case. If q_{i+1} was a duplicate of q_i, then $\mathcal{S}_0$ and $\mathcal{S}_1$ will replace q_{i+1} by logical address $n - p + i + 1$, which is the special purpose logical address we reserved for processor p_{i+1}. We call this replaced query a **fake query** and the target element a **fake element**. Because we sorted $\mathcal{Q}$ in the previous step, deduplication can be realized with high degree of parallelism. Formally, we want to compute the ideal functionality $\mathcal{F}_{\mathsf{dedupe}}$ listed in Figure 4.3.

When the query results are returned, $\mathcal{S}_0$ and $\mathcal{S}_1$ need to rearrange them so the ith processor

Functionality 4.4: $\mathcal{F}_{\mathsf{copy}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a query result vector $(\mathcal{R}_0, \mathcal{R}_1) = [\![\mathcal{R}]\!]$ and the duplication indicator string $(b_0, b_1) = [\![b]\!]$ from Figure 4.3. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs $\mathcal{R}$ from $\mathcal{R}_0, \mathcal{R}_1$ and b from b_0, b_1.
2. Scan through $\mathcal{R}$ and b simultaneously from left to right. If $b_{i+1} = 1$, set $\mathcal{R}_{i+1} = \mathcal{R}_i$.
3. Secret shares $\mathcal{R}$ with $\mathcal{S}_0$ and $\mathcal{S}_1$, $(\mathcal{R}_0, \mathcal{R}_1) = [\![\mathcal{R}]\!]$.
4. $\mathcal{S}_0, \mathcal{S}_1$ output $\mathcal{R}_0, \mathcal{R}_1$, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.4: The ideal functionality $\mathcal{F}_{\mathsf{copy}}$.

correctly gets back the element RAM$[q_i]$. We tackle this rearrangement problem in two steps, or one step if we are in one of the recursive position maps.

Firstly, $\mathcal{S}_0$ and $\mathcal{S}_1$ need to overwrite the fake elements with the real ones. Since the fake queries were duplicates in the first place, this process is essentially copying the real elements to the corresponding fake ones, and we call this process **copying**. To aid copying, we let $\mathcal{F}_{\mathsf{dedupe}}$ outputs shares of a bit-string b which we call the duplication indicator string. b_i indicates whether the ith query was fake (i.e., duplicate) or real. Formally, we want to compute the ideal functionality $\mathcal{F}_{\mathsf{copy}}$ listed in Figure 4.4.

The second step is only necessary if we are in the top-level RAM, for otherwise we have finished processing $\mathcal{R}$ and can simply return it up the recursion. If we are in the top-level RAM, however, $\mathcal{S}_0$ and $\mathcal{S}_1$ must reverse the sorting and permutation applied earlier. They accomplish this by first locally running the Batcher network backwards, using the choices bits they memorized before. At the end of this sorting reversal, the only remaining step is to reverse the permutation. Fortunately, $\mathcal{D}$ can assist $\mathcal{S}_0$ and $\mathcal{S}_1$ in this task as it remembers the permutation used earlier. Now, each processor has acquired the element it wanted, and so it returns the element to the user.

We proceed to describe our circuit implementation of $\mathcal{F}_{\mathsf{dedupe}}$. Because $\mathcal{Q}$ is already sorted, it suffices to compare each consecutive queries (q_i, q_{i+1}) in parallel. If they are equal, we will replace q_{i+1} by $n - p + i + 1$ and set $b_{i+1} = 1$, otherwise, we leave q_{i+1} unchanged and set b_{i+1}. Note that $b_{i+1} = \mathsf{eq}(q_i, q_{i+1})$ and $q_{i+1} = \mathsf{MUX}(b_{i+1}, q_{i+1}, n - p + i + 1)$. Here, $\mathsf{eq}(x, y)$ is the string equality comparison in Definition 4.1 and $\mathsf{MUX}(c, x_0, x_1)$ is the 2-to-1 string multiplexer in Definition 4.2.

Definition 4.1 ($\mathsf{eq}(x, y)$)**.** The string equality comparator, denoted $\mathsf{eq}(x, y)$, returns 1 if $x = y$ and 0 otherwise.

Definition 4.2 ($\mathsf{MUX}(c, x_0, x_1)$)**.** The 2-to-1 string multiplexer, denoted $\mathsf{MUX}(c, x_0, x_1)$, takes in three inputs: control bit c, candidate string x_0 and candidate string x_1, and returns x_c.

Since all the pairs of consecutive queries are examined in parallel, the depth of this circuit will be the sum of the depths of an eq and MUX, which is $O(\log(p))$ for p-bit strings. Equality between bits can be concisely implemented as a XNOR gate, which is the negation of XOR and can be expressed as $\mathsf{eq}(b_0, b_1) = \neg(b_0 \oplus b_1)$. Thus, the n-bit string equality comparator $\mathsf{eq}(x, y)$ can be written as

$$\mathsf{eq}(x_0 \dots x_{p-1}, y_0 \dots y_{p-1}) = \bigwedge_{i=0}^{p-1} \mathsf{eq}(x_i, y_i), \tag{4.1}$$

where the $\bigwedge$ denotes an AND gate of fan-in p. However, remember that we are only allowed AND gates of fan-in 2, consequently, we need $p-1$ fan-in 2 AND gates to implement a fan-in p AND gate. To minimize the depth, we leverage on the associativity of AND, which enables us to evaluate (4.1) in any order we wish. We select to evaluate it hierarchically like a binary tree on p-elements, where we combine two fan-in 2^i AND gates to form a fan-in 2^{i+1} AND gate, for $i \in [\lceil \log(p) \rceil - 1]$. The pseudocode describing this construction is given in Algorithm 4.3.

Algorithm 4.3: $\mathsf{eq}(x, y)$

Input: a pair of p-bit strings $x = x_0 \dots x_{p-1}, y = y_0 \dots y_{p-1}$
Output: 1 if $x = y$ and 0 otherwise
1. **if** $|x| = |y| = 1$ **then**
2.   **return** $\neg(x \oplus y)$
3. **return** $\mathsf{eq}(x_1 \dots x_{\lfloor p/2 \rfloor}, y_1 \dots y_{\lfloor p/2 \rfloor}) \wedge \mathsf{eq}(x_{\lfloor p/2 \rfloor + 1} \dots x_p, y_{\lfloor p/2 \rfloor + 1} \dots y_p)$

With the $\mathsf{eq}(x, y)$ gates at our disposal, we can produce the duplication indicator string b by running $\mathsf{eq}(x, y)$ on all pairs of consecutive queries (q_i, q_{i+1}) in parallel. In other words, we have

$$b = 0 \;||\; \mathsf{eq}(q_0, q_1) \;||\; \cdots \;||\; \mathsf{eq}(q_{p-2}, q_{p-1}). \tag{4.2}$$

Each b_{i+1} is then fed as control bits to MUX gates, also laid out in parallel for each q_{i+1}. To implement string MUX gates, we start with bit MUX gates. A bit MUX with inputs c, b_0, b_1 can be expressed as $\mathsf{MUX}(c, b_0, b_1) = (c \cdot b_0) \oplus (\neg c \cdot b_1)$, requiring only 2 parallel fan-in 2 AND gates to implement. To extend bit MUXs to string MUXs on p bits, we simply assign

a bit MUX to each bit in parallel and use the same c to control all of them. That is,

$$\mathsf{MUX}(c, x_0 \dots x_{p-1}, y_0 \dots y_{p-1}) = \mathsf{MUX}(c, x_0, y_0) \parallel \cdots \parallel \mathsf{MUX}(c, x_{p-1}, y_{p-1}). \tag{4.3}$$

To summarize, we give the pseudocode for the $\mathcal{F}_{\mathsf{dedupe}}$ circuit in Algorithm 4.4. Although the circuit as-is operates on clear text input, when we invoke the GMW protocol [21] on it, the participating parties will be holding secret shares of the input, as is formally required in $\mathcal{F}_{\mathsf{dedupe}}$ (Figure 4.3).

Algorithm 4.4: $\mathcal{F}_{\mathsf{dedupe}}(\mathcal{Q})$

Input: A query vector $\mathcal{Q} = (q_1, \dots, q_p)$
Output: $(\mathcal{Q}, b)$ as in Figure 4.3
1. **for** $i = 0, \dots, p-2$ **in parallel do**
2. $\quad b_{i+1} = \mathsf{eq}(q_i, q_{i+1})$
3. $\quad q_{i+1} = \mathsf{MUX}(b_{i+1}, q_{i+1}, n - p + i + 1)$
4. **return** $((q_0, q_1, \dots, q_{p-1}), 0b_1 \dots b_{p-1})$

To securely and efficiently implement the copying functionality $\mathcal{F}_{\mathsf{copy}}$, we employ the scan network in [39]. The scan operation generalizes prefix sum by allowing any associative binary operator to be used. In a prefix sum, we are given an array $A = (a_0, \dots, a_{p-1})$ and we would like to obtain a new array $\mathsf{PrefixSum}(A) = (\sum_{j=0}^{i} a_j)_{i=0}^{p-1}$. In other words, the ith entry of $\mathsf{PrefixSum}(A)$ is the sum of all the a_i's from 0 to i.

While prefix sum only uses addition, any associative binary operators is allowed in scan. We denote the scan operation using binary operator $\circ$ by $\mathsf{scan}_\circ$, e.g., prefix sum is now denoted scan_+. [39] describes an efficient network which enables us to compute scan with a circuit using exactly $2n - 2$ gates and $2d(\log(n) - 2)$ depth, where the gates implement the binary operator and d is the depth of such gate. We will use the following operator in our scan:

Definition 4.3 ($\rhd$)**.** Let $\rhd$ be the binary operator on two n-bit strings that satisfies

$$x \rhd y = \begin{cases} x, & \text{if } y \text{ is NIL} \\ y, & \text{otherwise} \end{cases} \tag{4.4}$$

Intuitively, when we scan A with $\rhd$, we will overwrite a_{i+1} by a_i if a_{i+1} is NIL, otherwise, it would leave a_{i+1} as-is. In our application, we will treat the fake elements as NIL. We claim that $\rhd$ is associative.

Lemma 4.1. $\forall a, b. [(a \rhd b) \rhd c = a \rhd (b \rhd c)]$.

Proof. Consider Table 4.2 which lists all the possible cases. Since $(a \rhd b) \rhd c$ agrees with $a \rhd (b \rhd c)$ in all possible cases, we conclude that $\rhd$ is indeed an associative binary operator. QED.

a	b	c	$(a \rhd b) \rhd c$	$a \rhd (b \rhd c)$
NIL	NIL	NIL	NIL	NIL
NIL	NIL	c	c	c
NIL	b	NIL	b	b
NIL	b	c	c	c
a	NIL	NIL	a	a
a	NIL	c	c	c
a	b	NIL	b	b
a	b	c	c	c

Table 4.2: “Truth table” for the $\rhd$ operator

We will now argue that, when using $\rhd$ to scan an array $A = (a_0, \ldots, a_{p-1})$ with possible NIL elements except the first, the resulting array is one where the non-NIL elements stay the same and the NIL elements are replaced by the closest non-NIL element preceding them.

Example 4.3. Suppose $A = (a_0, \mathrm{NIL}, a_1, \mathrm{NIL}, \mathrm{NIL}, a_2, a_3, \mathrm{NIL})$, then $\mathsf{scan}_{\rhd}(A) = (a_0, a_0, a_1, a_1, a_1, a_2, a_3, a_3)$.

Lemma 4.2. Let $A = (a_0, \ldots, a_{p-1})$ be an array such that a_0 is non-NIL. Then $\mathsf{scan}_{\rhd}(A)$ is an array where, for any i, $\mathsf{scan}_{\rhd}(A)[i] = a_j$, where $j = \max_{m \leq i}\{a_m \text{ is non-NIL}\}$. In other words, $j \leq i$ is the largest index such that a_j is non-NIL.

Proof. Another way of reading definition 4.3 is that $x \rhd y = y$ if y is non-NIL, regardless what x is. Consider $\mathsf{scan}_{\rhd}(A)[i]$ which holds the value $(a_1 \rhd \cdots \rhd a_j \rhd \cdots \rhd a_i)$, with a_j being the closest non-NIL element up to and including a_i. Such a_j always exist since we assumed a_0 is non-NIL. Because $\rhd$ is an associative binary operator, we may evaluate this expression in any order we wish, and we choose to evaluate it as $(a_1 \rhd \cdots \rhd a_{j-1}) \rhd (a_j \rhd \cdots \rhd a_i)$. If a_i is non-NIL, then the latter sequence is just a_i itself, and it is not hard to see that it evaluates to a_j. By our previous observation, regardless of the value of $(a_1 \rhd \cdots \rhd a_{j-1})$, it can't overwrite $(a_j \rhd \cdots \rhd a_i)$. Thus, we conclude that when the scan completes, a_i stays the same if it's non-NIL, or gets overwritten by the closest non-NIL element to its left. Since i was arbitrary, the proof is complete. QED.

$\mathsf{scan}_{\triangleright}(\cdot)$ is ideal for our purpose. In the current setting, we have a vector of query results $\mathcal{R}$, where some elements are fake due to duplication. When using $\mathsf{scan}_{\triangleright}(\cdot)$ on $\mathcal{R}$, we will treat the fake elements as NIL elements. Thus, by Lemma 4.2 and realizing that q_0 is always a real query, we conclude that when $\mathsf{scan}_{\triangleright}$ completes, each fake element will be overwritten by the closest real one preceding it. Since the query vector $\mathcal{Q}$ was sorted before deduplication, both the fake query and the closest real one preceding it were requesting for the same element, we can therefore deduce that this overwrite has the same effect as $\mathcal{F}_{\mathsf{copy}}$.

We finally remark on the implementation of $\triangleright$. As an optimization, we note that the duplication indicator string b already tells us exactly which queries are fake. Therefore, we can bypass some expensive comparison logic when implementing $\triangleright$ and let it directly read b, in which case the single-bit $a_i \triangleright_{b_{i+1}} a_{i+1}$ gate is just a single-bit $\mathsf{MUX}(b_{i+1}, a_i, a_{i+1})$ in disguise. Indeed,

$$a_i \triangleright_{b_{i+1}} a_{i+1} = \begin{cases} a_i, & b_{i+1} = 0 \\ a_{i+1}, & \text{otherwise} \end{cases} \tag{4.5}$$

and we extend these single-bit $\triangleright$ to handle p-bit strings by stacking p of them in parallel. Therefore, we conclude that for an array A having p w-bit elements, $\mathsf{scan}_{\triangleright}(A)$ can be implemented securely and efficiently with a circuit of $O(wp)$ size and $O(\log(p))$ depth. Setting $A = \mathcal{R}$, we conclude that $\mathcal{F}_{\mathsf{copy}}$ can be implemented securely with a circuit of $O(wp)$ size and $O(\log(p))$ depth.

Following the sorting and permutation reversal only necessary in the top-level RAM, we finish our handling of conflicting queries. We want to emphasize once more that every $\mathcal{Q}$ follows this workflow regardless whether it has conflicts, so that the parties' views are indistinguishable in both cases.

4.4.5 New base case position map

To avoid the otherwise overwhelming amount of work for refreshing, we'll explicitly deal with p queries when the number of elements stored in a position map drops below $\mathsf{threshold}$. We stress once again that, when the queries reach the base case, they are still in sorted order thanks to the sorting in our top-level RAM. Based on empirical results, we discover that setting $\mathsf{threshold} = p$ gives the best performance. This is intuitively justified since, in this setting, we have the same amount of parallelism as the number of elements in our RAM, so that each processor can be assigned to handle a RAM element.

To hide access patterns, we'll access all positions of the RAM. However, unlike in the non-parallel base case where we scan through the contents of the RAM p times to retrieve the

queried elements, we'll leverage the parallelism we have and use novel techniques to handle these p queries in parallel. For simplicity, we'll refer to the base case position map as the RAM in the rest of this section.

To help us explain how we explicitly handle the base case, let's first define the **characteristic string** associated with a query vector $\mathcal{Q} = (q_0, \ldots, q_{p-1})$. We denote the **characteristic string associated with** $\mathcal{Q}$ as $\chi_{\mathcal{Q}}$, which is defined as the n-bit string where the ith bit is set to 1 if and only if $i \in \mathcal{Q}$. In other words, the ith bit of $\chi(\mathcal{Q})$ indicates whether the logical address i is a query in $\mathcal{Q}$.

Definition 4.4 (One-hot encoding)**.** The **one-hot encoding** of $q \in \mathcal{Q}$, denoted $\chi(q)$, is the n-bit all-zero string except for the qth bit, with the **most** significant bit being the 0th bit. For example, if $p = n = 10$ and $q = 7$, then $\chi(q) = 0000000100$.

Definition 4.5 (Characteristic string)**.** The **characteristic string** of $\mathcal{Q} = (q_0, \ldots, q_{p-1})$, denoted $\chi(\mathcal{Q})$, is defined as $\chi(\mathcal{Q}) = \bigvee_{i \in [0,p-1]} \chi(q_i)$, where $\bigvee$ denotes bitwise OR. For example, if $p = n = 10$ and $\mathcal{Q} = (1, 1, 3, 3, 3, 4, 5, 5, 7, 8)$, then $\chi(\mathcal{Q}) = 0101110110$.

$\chi(\mathcal{Q})$ plays an important role in our base case. If we can efficiently obtain $\chi(\mathcal{Q})$ from $\mathcal{Q}$, then we can read out each element of the RAM **only once**, and use $\chi(\mathcal{Q})$ as a bit mask to filter out unwanted elements, leaving them as 0s. Specifically, from $\chi(\mathcal{Q})$, we can obtain the query result vector by computing the following element-wise product between $\chi(\mathcal{Q})$ and our RAM elements, where we use $\otimes$ to denote this element-wise product.

$$\mathcal{R} = \chi(\mathcal{Q}) \otimes \text{RAM} = (\chi(\mathcal{Q})[0] \cdot \text{RAM}[0], \ldots, \chi(\mathcal{Q})[n-1] \cdot \text{RAM}[n-1]). \tag{4.6}$$

Example 4.4. In our running example, we have $\mathcal{Q} = (1, 1, 3, 3, 3, 4, 5, 5, 7, 8)$ and $\chi(\mathcal{Q}) = 0101110110$. Writing our RAM elements as $\text{RAM} = [\![(x_0, \ldots, x_{n-1})]\!]$, we can compute $\mathcal{R} = \chi(\mathcal{Q}) \otimes \text{RAM} = (0, x_1, 0, x_3, x_4, x_5, 0, x_7, x_8, 0)$, notice how $\mathcal{R}$ contains precisely the elements we want.

As a consequence of its importance, we devote the rest of this section to explain our protocol for computing the ideal charateristic string generation functionality $\mathcal{F}_\chi$ listed in Figure 4.5.

To start, we note that the duplication indicator string b from $\mathcal{F}_{\mathsf{dedupe}}$, or rather its complement $\overline{b}$, provides a good starting point for computing $\chi(\mathcal{Q})$. Specifically, $\overline{b}[i] = 1$ if and only if q_i was the first query to its target logical address, which we refer to as a **real query**. It is not hard to see that $\overline{b}$ has the same number of 1s as $\chi(\mathcal{Q})$, and that the number of 1s in both $\overline{b}$ and $\chi(\mathcal{Q})$ correspond to the number of real queries. Moreover, if we obtained $\overline{b}$,

Functionality 4.5: $\mathcal{F}_\chi$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a query vector $(Q_0, Q_1) = [\![Q]\!]$. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs Q from Q_0, Q_1.
2. Initialize a n-bit string $\chi(\mathcal{Q}) = 0^n$.
3. Scan through Q, for each $q \in \mathcal{Q}$, set $\chi(\mathcal{Q})[q] = 1$.
4. Secret shares $\chi(\mathcal{Q})$ with $\mathcal{S}_0$ and $\mathcal{S}_1$, $(\chi(\mathcal{Q})_0, \chi(\mathcal{Q})_1) = [\![\chi(\mathcal{Q})]\!]$.
5. $\mathcal{S}_0, \mathcal{S}_1$ output $\chi(\mathcal{Q})_0, \chi(\mathcal{Q})_1$, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.5: The ideal functionality $\mathcal{F}_\chi$.

then to obtain $\chi(\mathcal{Q})$ it suffices to simultaneously move all the 1s from position i to position q_i, for all $i \in [0, p-1]$ such that $\bar{b}[i] = 1$.

Example 4.5. In our running example, we have $\mathcal{Q} = (1, 1, 3, 3, 3, 4, 5, 5, 7, 8)$ and $\chi(\mathcal{Q}) =$ 0101110110. After deduplication, we obtain $\bar{b}(\mathcal{Q}) = 1010011011$, the bit movements required to obtain $\chi(\mathcal{Q})$ from $\bar{b}$ is listed in Figure 4.6.

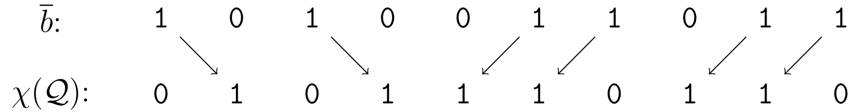

Figure 4.6: Bit movements required to obtain $\chi(\mathcal{Q})$ from $\bar{b}$.

As demonstrated by our example above, some of the bits need to move forward (to the right) and the rest move backward (to the left). We will decompose $\bar{b}$ into a set of bits that need to move forward and a set of bits that need to move backward, and handle these two sets simultaneously. Since the deduplication step ensures each logical address is queried only once, we can XOR the resultant strings from these two separate cases together to obtain $\chi(\mathcal{Q})$.

To move bits from their current positions to their destinations, we calculate the distance between them. For each bit $i \in [0, p-1]$ such that $\bar{b}[i] = 1$, we define its move distance $d(i)$ as $q_i - i$. If a bit needs to move forward or stay in place, its move distance is non-negative ($d(i) \geq 0$). If a bit needs to move backward, its move distance is negative ($d(i) < 0$).

Example 4.6. In our running example, we have $\mathcal{Q} = (1, 1, 3, 3, 3, 4, 5, 5, 7, 8)$ and $\bar{b}(\mathcal{Q}) =$ 1010011011. Then $d(\mathcal{Q}) = (1, 0, 1, 0, 0, -1, -1, 0, -1, -1)$, which indicates that, for example, the 9$^{\text{th}}$ bit of $\bar{b}(\mathcal{Q})$ needs to move backward 1 position.

Functionality 4.6: $\mathcal{F}_{\textsf{forward}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a l-element array $(A_0, A_1) = [\![A]\!]$, a distance vector $(d_0, d_1) = [\![d]\!]$, and n. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs $A = (A[0], \ldots, A[l-1])$ from A_0, A_1 and $d = (d[0], \ldots, d[l-1])$ from d_0, d_1.
2. Initialize a n-element array $A_{\textsf{forward}} = (0, \ldots, 0)$.
3. Scan through b and d. In parallel, for each i, set $A_{\textsf{forward}}[i + d[i]] = A[i]$.
4. Secret shares $A_{\textsf{forward}}$ with $\mathcal{S}_0$ and $\mathcal{S}_1$, $((A_{\textsf{forward}})_0, (A_{\textsf{forward}})_1) = [\![A_{\textsf{forward}}]\!]$.
5. $\mathcal{S}_0, \mathcal{S}_1$ outputs $(A_{\textsf{forward}})_0, (A_{\textsf{forward}})_1$, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.7: The ideal functionality $\mathcal{F}_{\textsf{forward}}$.

Functionality 4.7: $\mathcal{F}_{\textsf{backward}}$

Parameters: Parties $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$. $\mathcal{S}_0$ and $\mathcal{S}_1$ share a l-element array $(A_0, A_1) = [\![A]\!]$, a distance vector $(d_0, d_1) = [\![d]\!]$, and n. $\mathcal{D}$ does not have any input.
Procedure: 1. Reconstructs $A = (A[0], \ldots, A[l-1])$ from A_0, A_1 and $d = (d[0], \ldots, d[l-1])$ from d_0, d_1.
2. Initialize a n-element array $A_{\textsf{backward}} = (0, \ldots, 0)$.
3. Scan through b and d. In parallel, for each i, set $A_{\textsf{backward}}[i + d[i]] = A[i]$.
4. Secret shares $A_{\textsf{backward}}$ with $\mathcal{S}_0$ and $\mathcal{S}_1$, $((A_{\textsf{backward}})_0, (A_{\textsf{backward}})_1) = [\![A_{\textsf{backward}}]\!]$.
5. $\mathcal{S}_0, \mathcal{S}_1$ outputs $(A_{\textsf{backward}})_0, (A_{\textsf{backward}})_1$, respectively, and $\mathcal{D}$ outputs λ.

Figure 4.8: The ideal functionality $\mathcal{F}_{\textsf{backward}}$.

We can deduce the direction of movement from $d(i)$ if we use an appropriate numbering system like two's complement. In two's complement, a leading 0 indicates positivity (i.e., forward movement), while a leading 1 indicates negativity (i.e., backward movement). We can efficiently subtract two n-bit numbers using a circuit of size $O(n^2)$ and depth $O(\log n)$, such as the well-known carry lookahead adder (CLA).

We now introduce ideal functionalities move forward, denoted $\mathcal{F}_{\textsf{forward}}$, and move backward, denoted $\mathcal{F}_{\textsf{backward}}$. They are listed in Figure 4.7 and Figure 4.8, respectively. In both ideal functionalities, the inputs are a l-element array A, a distance vector d with $|d| = |A|$ and $d[i]$ being the distance that $A[i]$ needs to move by, and an integer $n \geq l$. The direction of movement is dictated by the functionality used, with $\mathcal{F}_{\textsf{forward}}$ always moving $A[i]$ to $A[i+d[i]]$ and $\mathcal{F}_{\textsf{backward}}$ always moving $A[i]$ to $A[i-d[i]]$.

Technically, we need to ensure that $i + d[i] \leq n - 1$ in $\mathcal{F}_{\textsf{forward}}$ and $i - d[i] \geq 0$ in $\mathcal{F}_{\textsf{backward}}$. However, it is not hard to see that this will be the case in our applications because logical

addresses are in the range $[0, n-1]$. Another technical detail that we point out is that in $\mathcal{F}_{\mathsf{backward}}$, the distances we use are non-negative, this simplifies our protocol with no loss of generality. Indeed, positive distances are easily obtained from their negative counterpart by taking the absolute value, which is cheap enough to compute in two's complement. Finally, we remark that $\mathcal{F}_{\mathsf{forward}}$ and $\mathcal{F}_{\mathsf{backward}}$ are mutual inverse of each other when using the same distance vector d (this is in fact the major reason we opted for positive distances in $\mathcal{F}_{\mathsf{backward}}$), that is,

$$\mathcal{F}_{\mathsf{forward}}(\mathcal{F}_{\mathsf{backward}}([\![b]\!], [\![d]\!], n), [\![d]\!], n) = [\![b]\!] = \mathcal{F}_{\mathsf{backward}}(\mathcal{F}_{\mathsf{forward}}([\![b]\!], [\![d]\!], n), [\![d]\!], n). \tag{4.7}$$

To implement $\mathcal{F}_{\mathsf{forward}}$ and $\mathcal{F}_{\mathsf{forward}}$, we will use circuits with a hierarchical structure. Our key insight is that we can use the binary expansion of the distance $d[i] = q_i - i$ to determine the sequence of hops needed to move from position i to q_i using only hops of lengths that are powers of 2.

Example 4.7. Suppose for a particular bit i, we have $d(i) = 11 = (1011)_2$, then we can interpret this binary string as saying that to move a distance of 11, we need to take a hop of length 8, no hop of length 4, a hop of length 2, and finally a hop of length 1.

With this in mind, it is perhaps not surprising that we use a circuit of $\lceil \log(n) \rceil$ levels (henceforth called a forwarding network) to compute $\mathcal{F}_{\mathsf{forward}}$, where the ith level consists of hops of length $2^{\lceil \log(n) \rceil - i - 1}$, for $i \in [0, \dots, \lceil \log(n) \rceil - 1]$. Specifically, in the ith level of the forwarding network, we process pairs of indices $(j, j + 2^i)$ in parallel for all $j \in [0, l - 2^i)$, and use the ith bit of $d[j]$ to determine whether to take the hop from j to $j + 2^i$. We set $A[j + 2^{\lceil \log(n) \rceil - i - 1}] = A[j]$ if and only if the ith bit of $d[j]$ is 1, and leave it unchanged otherwise. We can implement this operation using a MUX with the ith bit of $d[j]$ as the control bit and $A[j + 2^{\lceil \log(n) \rceil - i - 1}], A[j]$ as the candidate values. In fact, in order to still maintain access to $d[j]$ after each hop, we will forward $d[j]$ along with $A[j]$. Moreover, this also allows us to run the forwarding network backward later, when all the query results have been obtained.

Example 4.8. For an array of 8 elements, the structure of the forwarding network is shown in Figure 4.9. Note how the hop length decreases by half as we move down the levels.

The circuit to implement $\mathcal{F}_{\mathsf{backward}}$ (henceforth called a backwarding network) is almost the same one, except for the ith level we use hops of length 2^i, and that each hop overwrite a smaller index with a larger index.

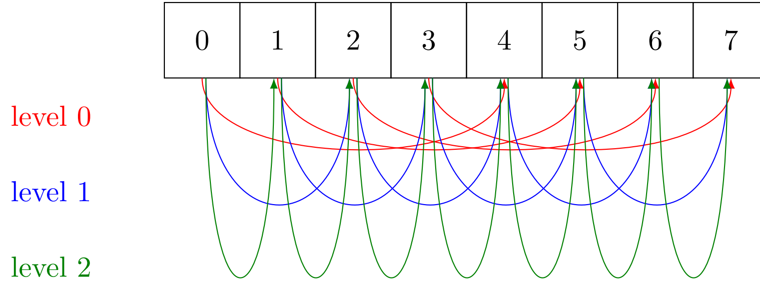

Figure 4.9: The forwarding network for an 8-element array.

Example 4.9. For an array of 8 elements, the structure of the backwarding network is shown in Figure 4.10. Note how the hop length increases by factor of 2 as we move down the levels.

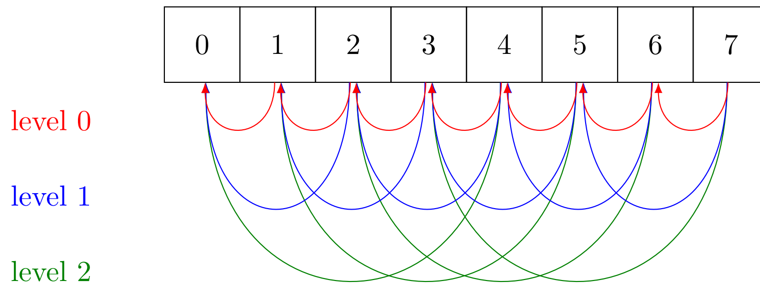

Figure 4.10: The backwarding network for an 8-element array.

While it may appear that our selection of hop lengths in the forwarding and backwarding networks is arbitrary, these choices are in fact deliberately made. Specifically, we aim to avoid write-conflicts, which occur when two distinct elements on the same level attempt to overwrite the same index.

Finally, we can describe our explicit approach to handling queries in the base case. When presented with a vector $\mathcal{Q}$ of sorted queries, our goal is to compute its characteristic string $\chi(\mathcal{Q})$. We break down this process into three steps:

1. To begin, we first deduplicate $\mathcal{Q}$ to obtain a duplication indicator string b. We then compute the complement of b, denoted by $\overline{b}$. $\overline{b}$ is valuable in determining $\chi(\mathcal{Q})$ because we realized that $\chi(\mathcal{Q})$ can be obtained from $\overline{b}$ by correctly rearranging the 1s. Specifically, for each index i where $\overline{b}[i] = 1$, we need to move the 1 to position q_i to form $\chi(\mathcal{Q})$. To accomplish this, we calculate the distance $d(i) = q_i - i$ for each 1 in

$\bar{b}$. Depending on the sign of $d(i)$, some bits will move forward (with a non-negative distance), while others will move backward (with a negative distance).

2. Next, we decompose $\bar{b}$ into two sets, each containing the bits that move in the same direction. For the set of bits that move forward, we apply the forwarding network with their distances unchanged. For the set of bits that move backward, we use the backwarding network with the absolute value of their distances. In both networks, we propagate both the bits and their distances to ensure that intermediate levels remember the forwarding decisions and that later on we can return results to the processor that requested them. Note that the forwarding and backwarding networks are run in parallel on each of the sets.

3. Lastly, since the deduplication step ensured that each logical address is queried for at most once, we take a bitwise XOR between the strings resulting from each set in step 2. The produces the characteristic string $\chi(\mathcal{Q})$ associated $\mathcal{Q}$.

After obtaining $\chi(\mathcal{Q})$, we can proceed to filter out the undesired elements by applying equation (4.6). To ensure that each element is correctly routed to the corresponding query, we rely on an earlier observation made in (4.7), which states that $\mathcal{F}_{\mathsf{forward}}$ and $\mathcal{F}_{\mathsf{backward}}$ are mutual inverses. Thus, we can effectively undo each network by running its inverse, provided that we keep track of which bits belong to which set prior to XORing them together in step 3 (which can actually be skipped altogether). Following the $\mathsf{scan}_{\triangleright}$ network discussed in the previous section, we finish our explicit handling of the base case queries.

CHAPTER 5: SECURITY ANALYSIS

In this chapter, we will formally prove Theorem 2.1, which states that our PRAM protocol is secure in the sense of simulation-based security. The approach that we will take to prove this theorem is by invoking the powerful composition theorem for semi-honest models as in [2]. In this framework, we can first prove that all of the sub-functionalities we use are semi-honest secure, and then treat those sub-functionalities as idealized oracles when proving security for the entire protocol. We choose this modular approach to make the proof piece-wise and thus a lot easier to explain.

5.1 AUTOMATICALLY SECURE PARTS OF OUR PRAM

We note that many sub-functionalities we utilize are implemented as circuits, and thus they are automatically secure when we employ the GMW protocol [21] for secure circuit evaluation. Additionally, in the GMW protocol, secret shares of input values, making these implementations compatible with our ideal functionalities which all require secret shares as inputs. We will list all sub-functionalities that are either inherently secure or can be easily secured in this manner, and we will omit security proofs as they are mainly just applying the GMW protocol.

It is worth mentioning that most of the circuits are executed between only two parties. As a result, the 2PC GMW protocol is not sufficient. This issue can be addressed by recognizing that the 2PC GMW protocol provides simulators for the participating parties, fulfilling the same simulator requirements as in Definition 2.2. Here, we heavily relied on the fact that we have an honest-majority among three parties, and as such it is not necessary to build simulators for corrupted parties of size 2. This fact allows us to combine these GMW-provided simulators with a trivial simulator for the non-participating party to satisfy the requirements of Definition 2.2.

- **Sorting:** For each $\mathcal{Q}$, we use the sorting only once in the top-level RAM. However, since we are using the Batcher sorting network to realize this sorting, which in turn consists of simple compare-and-swap gates, we see immediately that the entire Batcher sorting network is a circuit. Technically, the Batcher sorting network that we use is also revealing, meaning for each compare-and-swap gate $\mathcal{S}_0$ and $\mathcal{S}_1$ will reveal the result of the comparison. However, this is easily simultable thanks to the permutation we used before this sorting. Indeed, it suffices to sample a random bit for each revealed comparison-and-swap gate.

- $\mathcal{F}_{\mathsf{dedupe}}$: For each $\mathcal{Q}$, deduplication is performed at every level of recursion, starting with the top-level RAM. We completely implemented deduplication as a circuit, as a result, it is automatically secure.

- $\mathcal{F}_{\mathsf{copy}}$: For each $\mathcal{Q}$, copying is performed at every level of recursion, starting with the top-level RAM. We employ the scan network [39] with binary operator $\triangleright$ to implement copying, which in turn is just a disguised string MUX when provided with the duplication indicator string from $\mathcal{F}_{\mathsf{dedupe}}$. As a result, our implementation of $\mathcal{F}_{\mathsf{copy}}$ is automatically secure.

- **Addition and subtraction:** For each $\mathcal{Q}$, carry lookahead adders are used in the base case position map for calculating the distance vectors. These adders are well-known basic circuit constructions. As a result, our implementation of addition and subtraction is automatically secure.

- $\mathcal{F}_{\mathsf{forward}}$ **and** $\mathcal{F}_{\mathsf{backward}}$: For each $\mathcal{Q}$, move forward and move backward are only performed in the base case position map. We employ the forwarding (Figure 4.9) and backwarding (Figure 4.10) networks to implement them. The gates in these networks are all string MUXs, and thus making the networks themselves just circuits, when provided with the distance vector. As a result, our implementations of $\mathcal{F}_{\mathsf{forward}}$ and $\mathcal{F}_{\mathsf{backward}}$ are automatically secure.

- **Index calculation in** $\Pi_{\mathsf{binsearch}}$: In our rotation-based (or rather swap-based) binary search protocol $\Pi_{\mathsf{binsearch}}$, we implement the logic of next index calculation as a circuit. In fact, this circuits hard-codes the next index to visit, based on a comparison bit and a swap bit. As a result, the index calculation part of $\Pi_{\mathsf{binsearch}}$ is automatically secure.

These are all the sub-functionalities that are automatically secure by GMW. We also use other quite basic and well-known circuits to implement, for example, comparison, equality comparison, MUX, and so on. Next, we list the sub-functionalities that requires our explicit attention.

- $\mathcal{F}_{\mathsf{perm}}$: We implemented our permutation sub-functionality $\mathcal{F}_{\mathsf{perm}}$ with a custom resharing based shuffling protocol Π_{perm}, first proposed in [35]. This protocol is not implemented as circuit. Instead, it achieves security by having all three parties directly communicate with each other, but only sending messages that looks uniformly random to the participating parties. We will give simulators for each party shortly in section 5.2.

- $\mathcal{F}_{\mathsf{binsearch}}$: We implemented our binary search sub-functionality $\mathcal{F}_{\mathsf{binsearch}}$ with a custom rotation-based (or rather swap-based) binary search protocol $\Pi_{\mathsf{binsearch}}$, first proposed in [37]. Part of this protocol is implemented as circuit as we already pointed out. However, the protocol as a whole still require a formal simulation-based proof. Note that $\Pi_{\mathsf{binsearch}}$ only involves two active parties $\mathcal{S}_0$ and $\mathcal{S}_1$, and $\mathcal{D}$ is just idle in this protocol.

5.2 Π_{perm} IS SECURE

In this section, we will prove that the Π_{perm} protocol we use to compute $\mathcal{F}_{\mathsf{perm}}$ is secure. For easier reference, we have reproduced Π_{perm} in Algorithm 5.1.

Algorithm 5.1: $\Pi_{\mathsf{perm}}(A_0, A_1, \lambda)$

Input: A_0, A_1, λ from $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$, respectively

Output: $\Pi(A)_0, \Pi(A)_1, \Pi$ for $\mathcal{S}_0, \mathcal{S}_1, \mathcal{D}$, respectively

1. $\mathcal{S}_0$ and $\mathcal{S}_1$ reshare $[\![A]\!]$
2. S_0 sends A_0 to $\mathcal{D}$
3. $\mathcal{S}_1$ and $\mathcal{D}$ reshare $[\![A]\!]$
4. $\mathcal{D}$ and $\mathcal{S}_1$ together samples Π_1
5. $\mathcal{S}_1$ sends $\Pi_1(A_1)$ to $\mathcal{S}_0$
6. $\mathcal{S}_0$ and $\mathcal{D}$ reshare $[\![\Pi_1(A_1)]\!]$
7. $\mathcal{D}$ and $\mathcal{S}_0$ together samples Π_0
8. $\mathcal{D}$ sends $\Pi_0(\Pi_1(A_0))$ to $\mathcal{S}_1$
9. Let $\Pi = \Pi_1 \circ \Pi_0$
10. $\mathcal{S}_0, \mathcal{S}_1$ outputs $[\![\Pi(A)]\!]$
11. $\mathcal{D}$ outputs Π

Recall we arbitrarily stipulated that whenever $\mathcal{P}_0$ and $\mathcal{P}_1$ reshare something, $\mathcal{P}_0$ is the party that always sends the random string that it sampled, except the resharing in line 1 where this is reversed (so that line 2 is simulatable for $\mathcal{D}$). The randomness part of $\mathcal{S}_0$'s view only consists of its random choice when resharing $[\![\Pi_1(A_1)]\!]$ with $\mathcal{D}$ in line 6. The message part of its view consists of a random string from $\mathcal{S}_1$ in line 1, the $\Pi_1(A_1)$ from $\mathcal{S}_1$ in line 5, and finally the random permutation Π_0 from $\mathcal{D}$ in line 9.

Although seemingly true, we can't actually simulate all 4 things by uniform randomness, and the reason is that $\mathcal{S}_0$'s random choice in line 6, the messages it receives in line 5 and 7, and $\mathcal{S}_0$'s output are actually related in the real execution. Indeed, using the notations as in the protocol and denote $\mathcal{S}_0$'s random choice in line 6 by s, we must maintain

$$\Pi_0(\Pi_1(A_1) \oplus s) = \mathcal{F}_{\mathsf{perm},0}(A_0, A_1, \lambda). \tag{5.1}$$

Since the output is already "fixed" and provided to the simulator, we must make sure to simulate other parts in accordance with this output. Pseudocode for $\mathcal{S}_0$'s simulator is given in Algorithm 5.2, where n is the number of elements in A and w is the bit length of each element.

Algorithm 5.2: $\mathsf{Sim}_{\mathcal{S}_0}(A_0, \mathcal{F}_{\mathsf{perm},\mathcal{S}_0}(A_0, A_1, \lambda))$

Input: $A_0, \mathcal{F}_{\mathsf{perm},\mathcal{S}_0}(A_0, A_1, \lambda)$, which are $\mathcal{S}_0$'s input and output in the ideal functionality, respectively

Output: A simulated view identically distributed to the actual execution view $\mathsf{VIEW}^{\Pi_{\mathsf{perm}}}_{\mathcal{S}_0}(A_0, A_1, \lambda)$

1. Sample uniformly random $r \xleftarrow{\$} \{0,1\}^{wn}$ `// random string in line 1`
2. Sample uniformly random $\Pi_1(A_1) \xleftarrow{\$} \{0,1\}^{wn}$ `//` $\Pi_1(A_1)$ `received in line 5`
3. Sample uniformly random $\Pi_0 \xleftarrow{\$} \{0,1\}^{n\log(n)}$ `//` Π_0 `received in line 7`
4. Let $s = \Pi_0^{-1}(\mathcal{F}_{\mathsf{perm},\mathcal{S}_0}(A_0, A_1, \lambda)) \oplus \Pi_1(A_1)$ `// random choice in line 6`
5. Output $(A_0, s, r, \Pi_1(A_1), \Pi_0)$

For $\mathcal{S}_1$, the randomness part of its view consists of its random choice when resharing $[\![A]\!]$ with $\mathcal{S}_0$ in line 1 and its random choice when resharing $[\![A]\!]$ with $\mathcal{D}$ in line 3. The message part of its view consists of Π it receives from $\mathcal{D}$ in line 4, and also $\Pi_0(\Pi_1(A_0))$ it receives from $\mathcal{D}$ in line 8, which also happens to be its output. Therefore, the simulator for $\mathcal{S}_1$ is significantly simpler, using uniform randomness to simulate everything except the last message. This simulator is listed in Algorithm 5.3.

Algorithm 5.3: $\mathsf{Sim}_{\mathcal{S}_1}(A_1, \mathcal{F}_{\mathsf{perm},\mathcal{S}_1}(A_0, A_1, \lambda)$

Input: $A_1, \mathcal{F}_{\mathsf{perm},\mathcal{S}_1}(A_0, A_1, \lambda)$, which are $\mathcal{S}_1$'s input and output in the ideal functionality, respectively

Output: A simulated view identically distributed to the actual execution view $\mathsf{VIEW}^{\Pi_{\mathsf{perm}}}_{\mathcal{S}_1}(A_0, A_1, \lambda)$

1. Sample uniformly random $r \xleftarrow{\$} \{0,1\}^{wn}$ `// random choice in line 1`
2. Sample uniformly random $s \xleftarrow{\$} \{0,1\}^{wn}$ `// random choice in line 3`
3. Sample uniformly random $\Pi_1 \xleftarrow{\$} \{0,1\}^{n\log(n)}$ `//` Π_1 `received in line 4`
4. Output $(A_1, (r, s), \Pi_1, \mathcal{F}_{\mathsf{perm},\mathcal{S}_1}(A_0, A_1, \lambda))$

Finally, for $\mathcal{D}$, the randomness part of its view consists of the two random permutations it sends $\mathcal{S}_0$ and $\mathcal{S}_1$, namely, Π_1 in line 4 and Π_0 in line 7. The message part of its view consists of A_0 from $\mathcal{S}_0$ in line 2, a random string from $\mathcal{S}_1$ on line 3, a random string from $\mathcal{S}_0$ in line 6. Among these 5 things, we must make sure that the two random permutations it

samples must be consistent with its output, but otherwise everything else can be simulated by uniform randomness. The simulator for $\mathcal{D}$ is described in Algorithm 5.4

Algorithm 5.4: $\mathsf{Sim}_{\mathcal{D}}(A_1, \mathcal{F}_{\mathsf{perm},\mathcal{D}}(A_0, A_1, \lambda)$

Input: λ, Π, where Π is $\mathcal{D}$'s output in the ideal functionality

Output: A simulated view identically distributed to the actual execution view $\mathsf{VIEW}_{\mathcal{D}}^{\Pi_{\mathsf{perm}}}(A_0, A_1, \lambda)$

1. Sample uniformly random $A_0 \xleftarrow{\$} \{0,1\}^{wn}$ `//` A_0 `received in line 2`
2. Sample uniformly random $r \xleftarrow{\$} \{0,1\}^{wn}$ `// random string in line 3`
3. Sample uniformly random $\Pi_1 \xleftarrow{\$} \{0,1\}^{n \log(n)}$ `// random choice in line 4`
4. Sample uniformly random $s \xleftarrow{\$} \{0,1\}^{wn}$ `// random string in line 6`
5. Let $\Pi_0 = \mathcal{F}_{\mathsf{perm},\mathcal{D}}(A_0, A_1, \lambda) \oplus \Pi_1$ `// random choice in line 7`
6. Output $(\lambda, (\Pi_1, \Pi_0), A_0, r, s)$

Since all three parties are simulatable and the simulated views are identically distributed to the real execution views, we conclude that Π_{perm} is secure in the sense of Definition 2.2.

5.3 $\Pi_{\mathsf{binsearch}}$ IS SECURE

In this section, we formally prove that Π_{perm} is secure, which implements $\mathcal{F}_{\mathsf{perm}}$ (Figure 4.2). Recall that $\mathcal{F}_{\mathsf{perm}}$ plays a central role in $\mathcal{S}_0$ and $\mathcal{S}_1$'s utilization of the timetable, firstly used when they want to fetch secret shares of the physical address of their queried elements, and secondly used to replace the dummy on the same level with this element when $\mathcal{D}$ sends over a list of dummy addresses, one from each level. As a result, it is crucial that Π_{perm} is both secure and efficient, and we have already discussed the efficiency of Π_{perm}. For easier reference, Π_{perm} is reproduced in Algorithm 5.5.

In this protocol, almost all the computations are carried out by circuits, namely, the comparisons in line 3 and line 8, the next index calculation in line 5 (as we mentioned previously), and the current element replacement in line 9. These steps are again automatically secure by GMW, and as such can be treated as idealized functionalities when building our simulators, all thanks to the powerful composition theorem in [2]. The only left to deal with is the revealings of next index to visit in line 6, and as we will see, these are easy to simulate by an uniform random path in the tree because of the random swaps we perform at each node of the binary search tree.

Also note that both servers' views are highly similar in this protocol, and without loss of generality, we will only give a simulator for $\mathcal{S}_0$, which is described in Algorithm 5.6. This

Algorithm 5.5: $\Pi_{\mathsf{binsearch}}((A_0, B_0, k_0), (A_1, B_1, k_1), \lambda)$

Input: $\mathcal{S}_0$ and $\mathcal{S}_1$ hold secret shares of $(A_0, A_1) = [\![a_0, \ldots, a_{l-1}]\!]$, already ordered as a binary search tree. They also holds secret shares of the swap bits $(B_0, B_1) = [\![b_0, \ldots, b_{\log(l)-1}]\!]$ and the search key $(k_0, k_1) = [\![k]\!]$. $\mathcal{D}$ has no input.

Output: $[\![a_i]\!]$ where $i = \max_{j \in [0,l-1]}\{a_j \leq k\}$.

1. $[\![p]\!] = [\![a_0]\!]$ `// the element to return`
2. $[\![d]\!] = 0$ `// the current index in` A
3. $[\![c]\!] = \mathsf{compare}([\![k]\!], [\![a_0]\!])$ `// the current comparison result`
4. **for** $i = 1, \ldots, \log(l) - 1$ **do**
5. $\quad [\![d]\!] = ([\![2d + 1 + c]\!]) \cdot b_i \oplus ([\![2d + c]\!]) \cdot (\neg b_i)$ `// calculating next index`
6. $\quad \mathcal{S}_0$ and $\mathcal{S}_1$ reveal d
7. $\quad [\![a]\!] = [\![a_d]\!]$ `// the current element`
8. $\quad [\![c]\!] = \mathsf{compare}([\![k]\!], [\![a]\!])$
9. $\quad [\![p]\!] = [\![c]\!] \cdot ([\![a]\!] \oplus [\![p]\!]) \oplus [\![p]\!]$ `// replace` $[\![p]\!]$ `by` $[\![a]\!]$ `if` $a < k$
10. **return** $[p]$

simulator will be our first exhibition of the oracle-aided protocol technique, granted to us by the powerful composition theorem. To avoided complicated notation, whenever we write a step using the same notation as in Algorithm 5.5, we mean that we are invoking the oracle for that step. Whenever a variable already has a subscript, i.e. b_i, we write $b_{i,\mathcal{S}_0}$ to mean $\mathcal{S}_0$'s share of this variable.

In this simulator, the crucial observation is that the revealed path in the actual execution view is uniformly random among all the possible paths. Therefore, to make the simulated view identically distributed to the actual one, it suffices for our simulator to choose an uniformly random path and reveal this path. The rest of the simulation are rather pesky details required to simulate an oracle-aided protocol. In particular, for every step where we used an idealized oracle, we are required to put the output of that oracle also into our view. For completeness, we will prove that the revealed path in an actual execution of $\Pi_{\mathsf{binsearch}}$ is uniformly distributed among all possible paths, which is our final missing piece to formally argue that the above simulator produces an identically distributed view.

Lemma 5.1. In an actual execution of $\Pi_{\mathsf{binsearch}}$, the revealed path, denoted $P = (P_0, \ldots, P_{\log(l)-1})$, is uniformly distributed among all possible paths.

Proof. Consider an arbitrary path $P = (P_0, \ldots, P_{\log(l)-1})$, and let $x_0, \ldots, x_{\log(l)-1}$ be random variables denoting nodes on a path. We want to show that

$$\Pr[x_0 = P_0, \ldots, x_{\log(l)-1} = P_{\log(l)-1}] = 2^{-(\log(l)-1)}. \tag{5.2}$$

Algorithm 5.6: $\mathsf{Sim}_{\mathcal{S}_0}((A_0, B_0, k_0), a_0)$

Input: $(A_0, B_0, k_0), \mathcal{F}_{\mathsf{binsearch},0}((A_0, B_0, k_0), (A_1, B_1, k_1), \lambda)$, which are $\mathcal{S}_0$'s input and output in the ideal functionality, respectively.

Output: A simulated view identically distributed to the actual execution view $\mathsf{VIEW}_{\mathcal{S}_0}^{\Pi_{\mathsf{binsearch}}}((A_0, B_0, k_0), (A_1, B_1, k_1), \lambda)$.

1. Let VIEW be a list initially only containing (A_0, B_0, k_0)
2. Let $P = (P_0, P_1, \ldots, P_{\log(l)-1})$ be an uniformly randomly chosen path in plain text, where P_i is the ith level node on this path
3. $p_0 = a_0$ `// not in the view`
4. Samples uniform random $d_0 \xleftarrow{\$} \{0, 1\}$ `// not in the view`
5. $c_0 = \mathsf{compare}(k_0, a_0)$ `// invoking the compare oracle`
6. Append c_0 to VIEW
7. **for** $i = 1, \ldots, \log(l) - 1$ **do**
8. $\quad d_0 = \mathsf{NextIndexOracle}(d_0, c_0, b_{i,\mathcal{S}_0})$ `// invoking the next index oracle`
9. $\quad$ Append d_0 to VIEW
10. $\quad$ Append P_i to VIEW `// the revealing in line 6`
11. $\quad a_0 = a_{d,\mathcal{S}_0}$ `// not in the view`
12. $\quad c_0 = \mathsf{compare}(k_0, a_0)$ `// invoking the compare oracle`
13. $\quad$ Append c_0 to VIEW
14. $\quad p_0 = \mathsf{ReplaceCurrElem}(c_0, a_0, p_0)$ `// invoking the replace current element oracle`

`/* Making sure the simulated view is consistent with actual view by replacing last` p_0 `with the actual output */`

15. Go through VIEW, replace the last p_0 with $\mathcal{F}_{\mathsf{binsearch},0}((A_0, B_0, k_0), (A_1, B_1, k_1), \lambda)$
16. Output VIEW

By the definition of conditional probability, we may write the probability on the LHS as

$$\Pr[x_0 = P_0, \ldots, x_{\log(l)-1} = P_{\log(l)-1}] = \prod_{i=0}^{\log(l)-1} \Pr[x_i = P_i \mid x_0 = P_0, \ldots, x_{i-1} = P_{i-1}]. \quad (5.3)$$

Observe that each term in the product can be simplified to $\Pr[x_i = P_i \mid x_{i-1} = P_{i-1}]$ since the next nodes on a path is only dependent on its parent node, i.e. the random variable x_i's form a Markov chain. Now, we realize that, for all $i \geq 1$,

$$\Pr[x_i = P_i \mid x_{i-1} = P_{i-1}] = \Pr[x_i = \mathsf{Sibling}(P_i) \wedge b_i = 1 \mid x_{i-1} = P_{i-1}] + \Pr[x_i = P_i \wedge b_i = 0 \mid x_{i-1} = P_{i-1}], \quad (5.4)$$

because both events on the RHS are mutually disjoint. These events also both occur with probability 1/4. Thus, for all $i \geq 1$, we have $\Pr[x_i = P_i \mid x_{i-1} = P_{i-1}] = 1/2$. Keeping in

mind that $\Pr[x_0 = P_0] = 1$ as there is only one root, we conclude that

$$\Pr[x_0 = P_0, \ldots, x_{\log(l)-1} = P_{\log(l)-1}] = \prod_{i=0}^{\log(l)-1} \Pr[x_i = P_i \mid x_0 = P_0, \ldots, x_{i-1} = P_{i-1}] \quad (5.5)$$

$$= \prod_{i=1}^{\log(l)-1} \frac{1}{2} \quad (5.6)$$

$$= 2^{-(\log(l)-1)}. \quad (5.7)$$

Since P was arbitrary, the proof is complete. QED.

We are finally able to claim that the simulated view for $\mathcal{S}_0$ produced by Algorithm 5.6 is identically distributed to the real execution view, and the simulator for $\mathcal{S}_1$ is almost the same if not verbatim identical. Technically, we should also simulate $\mathcal{D}$, however, the simulation is trivial if we realize that $\mathcal{D}$ has no part in this protocol. Indeed, a simulator for $\mathcal{D}$ will just be the trivial one which has no inputs and outputs nothing. This shows

5.4 THE REST OF THE PRAM PROTOCOL IS SECURE

With most parts of our protocol proven secure, we believe that the perfect security of our protocol is already obvious. However, for completeness, we shall briefly comment on the parts of our protocol that still need some attention.

Initialization and refreshing. We haven't yet talked about the very step and another periodically performed step in our protocol, namely, initialization and refreshing. Note that these two steps are intimately related, as refreshing is essentially reading out all the elements followed by initializing the next instance of PRAM. During initialization, the servers write their shares of the elements sequentially to level $2^{\log(n)+1}$, which is just enough to hold these elements and accompanying dummies. Since the addresses they are writing into is public knowledge, these access patterns are simulatable. Then all three parties will engage in permuting the $2^{\log(n)+1}$ level, followed by the server sharing the timetable between $\mathcal{S}_0$ and $\mathcal{S}_1$. We have already discussed permutations, to simulate the timetable, we create tables of the appropriate size and fill each row with uniformly random entries. Finally, the servers will ask for $[\![\pi(0)]\!], \ldots, [\![\pi(n)]\!]$, which they will use to initialize their position maps. Since these $[\![\pi(i)]\!]$'s are again secret shares, we can simulate them by uniform random strings of the appropriate length.

During refreshing, the servers will first read out the elements sequentially, which involves fetching the relevant rows of the timetable, and locating in each row the relevant entry documenting each element's current position. To fetch the rows from the timetable, they will consult their position maps for the $[\![\pi(t')]\!]$'s, and each locally read out the rows. The rest of timetable reading is handled by invoking the binary search protocol $\Pi_{\mathsf{binsearch}}$. However, we emphasize that since $m = 2^i n$, whenever a refreshing happens, it's public knowledge that the elements all resides on the largest level of storage. Therefore, there is no need for $\mathcal{D}$ to additionally send list of dummy addresses for each physical address, because the access patterns are now simulatable by just reading random positions from the largest level of storage.

Timetable operations. Timetable operations are largely based on $\Pi_{\mathsf{binsearch}}$, except for fetching the relevant rows. So far, we have treated the timetable as if it's stored in a separate memory structure. However,to argue obliviousness we also need to simulate access patterns into this memory structure. Fortunately, this is not too hard. Remember that each row of the timetable is relevant for only 1 query, since once an element is accessed, it is written back to the stash and starts from the top-level of our storage all over again. Being a static data structure, the timetable has no way to keep track of this change. Therefore, on subsequent access of the same element, a different row of the timetable will be fetched.

Furthermore, recall that the rows of the timetable are permuted to prevent the servers from learning the last access time of an element. Therefore, to simulate the sequence of accesses into the timetable, we can sample uniform random locations without replacement. This includes the refreshing step where we access n rows of the timetable simultaneously, when only n rows of the timetable are left unsampled. At that time, we access all the unsampled rows simultaneously.

Position maps. Perhaps the least obvious component of all to simulate is our position maps, due to their recursively constructed nature. Our idea to simulate the position maps is, perhaps not surprisingly, using recursion. Specifically, we first build the simulator for the entire PRAM except the recursive position maps, and then recursively use the simulator to generate views for the recursive position maps. Notice that in the base case everything is implemented by circuits, including deduplication and the forwarding/backwarding networks. Therefore, the base case simulator is granted by the GMW protocol. Due to its intricacy, it is worth discussing some of the details of this recursive simulator.

- To use the simulator, we must provide it with inputs that are consistent with the rest of the view. Recall that the simulator takes in the input and ideal output. The input

to each recursive position maps is of the form (A, Q), where A is the content of the memory and Q is the sequence of parallel accesses. The content A are already simulated by uniform randomness in the rest of the simulation, for example, when we reveal the $\pi(t')$'s during timetable row fetching. The induced sequence of parallel accesses can be obtained from the sorted Q by dividing each logical address by 2 and then take the flooring, which is the same as a logical right shift. This can be implemented by a simple circuit and thus is automatically secure, so we are allowed to use an idealized oracle to produce the Q's, and use the messages we receive from this oracle as Q. Finally, we need the recursive simulator with the ideal output, however, this output is actually the old $\pi(t')$'s for the queried elements, and are therefore is already part of the view. Thus, both the input and ideal output to the recursive simulator is either part of the rest of the view or can be obtained from idealized oracles, we just need to pass them to the recursive simulator in a consistent manner.

- Secondly, this recursive simulator suffers from one problem, namely, the contents of the generated views are not in the correct order. Indeed, in an actual execution, the views for the recursive position maps will be obtained as soon as each map is queried, but the views produced by this recursive simulator would aggregate these views at one place. However, we can fix this issue by running yet another program to correct the order. This program takes in the views generated by the recursive simulator, and rearrange the views so that they are ordered according to an actual execution of the protocol. Our overall simulator would then consist of two parts, the first being the recursive simulator whose output is not yet correct, and the second part being the program that correct its output.

5.5 THE ENTIRE PROTOCOL IS SECURE

We have either formally or informally argued why each component of our protocol is secure, and we have discussed simulating the entire protocol using a recursive simulator and then a program to correct its output. Thus, we finally conclude that our entire protocol is secure, proving Theorem 2.1.

Proof of Theorem 2.1. For each party, use a recursive simulator to produce a view whose content are not yet ordered correctly, and then run a program to rearrange the content of the view so that it is consistent with an actual execution of the protocol. When building the recursive simulator, we follow the oracle-aided framework and can thus replace each sub-functionality by an idealized oracle implementing that functionality. Since all three

parties' views are identically distributed to the their real execution view, we conclude that the protocol is perfectly secure. QED.

As a consequence of 2.1, notice that we can almost claim our PRAM protocol satisfies Definition 2.3 because almost all of the physical access patterns are included in the views of the parties. However, there are two places where the physical access patterns are not in the views, namely, during the initialization step (including the initialization caused by a refreshing) and in the base case position map. This is readily amendable as we realize that in these cases, the physical access patterns are prescribed by our protocol. Indeed, when initializing, the servers write to the first n positions of level $2^{\log(n)+1}$ in sequence, and in the base case the entire RAM is accessed. As a result, we have the following Corollary:

Corollary 5.1. Our PRAM protocol satisfies the obliviousness requirement of Definition 2.3. In other words, our PRAM is more powerful than a traditional ORAM (or a traditional OPRAM thereof).

CHAPTER 6: EFFICIENCY ANALYSIS

In this chapter, we will summarize the communication complexity and rounds complexity of the different operations in our protocol in Table 6.1, and justify the asymptotics we claimed in the abstract. In Table 6.1, we use the same notation as in Table 3.1. In particular, n is the number of elements stored, m is the number of maximum accesses each instance of our PRAM supports, k is the total number of accesses made by the user, p is the number of processors, and as always, we set $\mathsf{threshold} = p$.

This table mostly only summarizes the costs of these operations in the top-level RAM, since we will use the costs at this level to upper bound all the costs at the intermediate recursive position maps. We note that in the non-base-case recursive position maps, the word size is $O(\log(n))$ instead, and since m would decrease by half at each level of recursion, the $\log(m)$'s effectively decrease by 1. Therefore, our asymptotics are slightly more pessimistic than the reality.

Component	Communication complexity	Rounds complexity
	Top-level RAM	
Initialization	$O(m\log^2(m) + n\log(m) + nw)$	$O(1)$
Permutation ($\mathcal{Q}$)	$O(p\log(m))$	$O(1)$
Batcher sorting	$O(p\log^2(p)\log(m))$	$O(\log^2(p)\log\log(m))$
Deduplication	$O(p\log(m))$	$O(1)$
Copy	$O(pw)$	$O(\log(p))$
Binary search	$O(\log(m)\log\log(m))$	$O((\log\log(m))^2)$
Refreshing	$O(n\log(m)\log\log(m))$ + Initialization	$O((\log\log(m))^2)$
Permutation (i^{th} level)	$O(2^i w)$	$O(1)$
	Base case position map	
Distance calculation	$O(p\log^2(p))$	$O(\log\log(p))$
Forwarding	$O(p\log^2(p))$	$O(\log(p))$
Backwarding	$O(p\log^2(p))$	$O(\log(p))$

Table 6.1: Summary of complexities for different operations

We show another table, Table 6.2, which lists the number of times each of the above operations are performed for a total of k accesses.

Using these tables, it is straightforward to calculate the total costs incurred on the top-level RAM and the base case for k total accesses. Indeed, we mostly just multiply the rows in Table 6.1 by the corresponding rows in Table 6.2. Since the table is too wide, We separately show the communication complexity in Table 6.3 and the rounds complexity in Table 6.4.

Component	Number of times performed
Top-level RAM	
Initialization	1
Permutation ($\mathcal{Q}$)	k/p
Batcher sorting	k/p
Deduplication	k/p
Copy	k/p
Binary search	$2k$
Refreshing	$k/m - 1$
Permutation (i^{th} level)	$2^{\log(m)-i}(k/m)$
Base case position map	
Distance calculation	k/p
Forwarding	$2k/p$
Backwarding	$2k/p$

Table 6.2: Summary of number of times each operation is performed for k accesses

Notice that for the rounds complexity table, the rounds contributed by permutations are not directly multiplying the times each level is permuted by the cost of each permutation. This is because during an actual execution, only a single permutation is performed at each epoch, although the number of items permuted could be large. Thus, the rounds contributed by eviction permutations are really just $O(k/p)$.

Now, to get the final asymptotics, we note that there are only $\log(n/p)$ many levels of intermediate recursion, and the cost of each is upper-bounded by the cost of the top-level RAM. Thus, we can multiply the costs in Table 6.3 and Table 6.4 by $\log(n/p)$ for an upper bound, except the base case position map costs and the permutation and Batcher sorting, since the latter 2 are only performed in the top-level RAM.

Summing up, collecting terms, and keeping in mind that $m = O(n)$ (in fact, $\geq n$) so that terms like $O(kn/m) = O(k)$, it is not hard to see that the total communication complexity of k total accesses is

$$O(k \log^2(p) \log(n)) + \log\left(\frac{n}{p}\right) O(kw \log(n) + k \log^2(n)), \tag{6.1}$$

when amortized over k, the total number of accesses, this becomes

$$O(\log^2(p) \log(n)) + \log\left(\frac{n}{p}\right) O(w \log(n) + \log^2(n)), \tag{6.2}$$

Component	Communication complexity
Top-level RAM	
Initialization	$O(m\log^2(m) + n\log(m) + nw)$
Permutation ($\mathcal{Q}$)	$O(k\log(m))$
Batcher sorting	$O(k\log^2(p)\log(m))$
Deduplication	$O(k\log(m))$
Copy	$O(kw)$
Binary search	$O(k\log(m)\log\log(m))$
Refreshing	$O((kn/m)\log(m)\log\log(m))$ + (k/m) Initialization
Permutation (i^{th} level)	$O(kw)$
Permutation (levels total)	$O(k\log(m)w)$
Base case position map	
Distance calculation	$O(k\log^2(p))$
Forwarding	$O(k\log^2(p))$
Backwarding	$O(k\log^2(p))$

Table 6.3: Summary of total communication complexity of each operation for k accesses

as stated in the abstract.

Component	Rounds complexity
Top-level RAM	
Initialization	$O(1)$
Permutation ($\mathcal{Q}$)	$O(k/p)$
Batcher sorting	$O((k/p)\log^2(p)\log\log(m))$
Deduplication	$O(k/p)$
Copy	$O((k/p)\log(p))$
Binary search	$O((k/p)(\log\log(m))^2)$
Refreshing	$O((k/m)(\log\log(m))^2)$
Permutation (levels total)	$O((k/p))$
Base case position map	
Distance calculation	$O((k/p)\log\log(p))$
Forwarding	$O((k/p)\log(p))$
Backwarding	$O((k/p)\log(p))$

Table 6.4: Summary of total rounds complexity of each operation for k accesses

Similarly, the rounds complexity sums up to

$$O\left(\left(\frac{k}{p}\right)\log^2(p)\log\log(n)\right)+\log\left(\frac{n}{p}\right)O\left(\left(\frac{k}{p}\right)(\log(p)+(\log\log(n))^2)\right), \tag{6.3}$$

and when amortized over k/p, the total number of sets of parallel queries executed, this becomes

$$O(\log^2(p)\log\log(n))+\log\left(\frac{n}{p}\right)O(\log(p)+(\log\log(n))^2), \tag{6.4}$$

as stated in the abstract. In other words, each set of p parallel queries finish in the number of rounds stated in (6.4).

CHAPTER 7: EXPERIMENTS

In this chapter, we describe our experiment setups and then report our empirical findings, demonstrating the concrete efficiency of our PRAM protocol.

7.1 EXPERIMENT SETUPS

To evaluate the concrete efficiency of our protocol, we wrote a C++ program that simulates an execution of our protocol. To avoid confusing this simulation program with the simulator in the security proofs, henceforth we will call this program the **emulator**. The emulator closely follows the protocol in every step, except that it replaces actual communications by counting their costs instead.

Example 7.1. For example, the equality function in Figure 7.1 emulate the cost of an equality gate. Every time in the protocol when the parties need to compute an equality gate, such as in the deduplication step, we invoke this function to count its cost. In our emulator, cost is a struct that has two fields, a n_bits field and a n_rounds field, counting the number of bits and number of rounds required to securely compute some functionality, respectively.

The cost we use for the basic operations such as computing AND gates and XOR gates are hard-coded, with their values determined from the most efficient protocols that we are aware of in the 3PC semi-honest honest-majority setting. The overall emulator program has the following parameters:

- n : the number of elements stored in our PRAM.
- w : the bit length of each element stored in our PRAM. Empirically, we always set this to $\log^2(n)$ since our asymptotic calculations suggest so.

```
#define clog(n) ceil(log2(n))

cost equality(int n) {
  return cost {
    .n_bits = (n - 1) * AND.n_bits,
    .n_rounds = clog(n) * AND.n_rounds
  };
}
```

Figure 7.1: Our code for emulating an `eq` gate.

- k : the total number of queries to our PRAM.
- m : the number of accesses an instance of our PRAM supports, i.e. every m accesses we must refresh our PRAM.
- p : the number of processors in our PRAM machine.
- $\mathsf{threshold}$: the threshold value for entering the base case position map. Empirically, we always set this to p, for the reason we gave in Section 4.3.

We use the emulator to test the concrete performance of our PRAM protocol, and we adjust the parameters to see how well our PRAM scales with n and p.

7.2 EMPIRICAL RESULTS

Empirically, we find that our protocol is concretely efficient, and matches the asymptotic calculations with low constants. In the first set of experiments, we explore how our PRAM scales with n, and in the second set of experiments we explore how it scales with p.

7.2.1 Our PRAM's scalability with n

In this experiment, we fix the number of processors $p = 64$, which is a reasonable amount of processors to expect on a modern day cloud server such as the AWS [40], and then we gradually increase n exponentially, up to $n = 25$. The other parameters are also fixed, with $m = k = n$, $w = \log^2(n)$, and $\mathsf{threshold} = p = 64$. Plots of our experimental findings are shown in Figure 7.2, Figure 7.3, and Figure 7.4, with the accompanying data in Table 7.1. In all plots, the x-axis is drawn on log-scale.

The experimental data presented here aligns well with our asymptotic calculations. Notably, the communication overhead, defined as the number of bits transmitted per bit fetched, exhibits growth on the order of O(log(n)), although some lower-order terms, such as O(log log(n)), contribute to a less curved graph compared to a typical logarithmic graph. Additionally, the rounds of communication are effectively O(log(n)), which is precisely what the asymptotic calculations suggest in our case where p = O(1). This consistency between the experimental results and theoretical predictions shows that our PRAM protocol is concretely efficient with low constants.

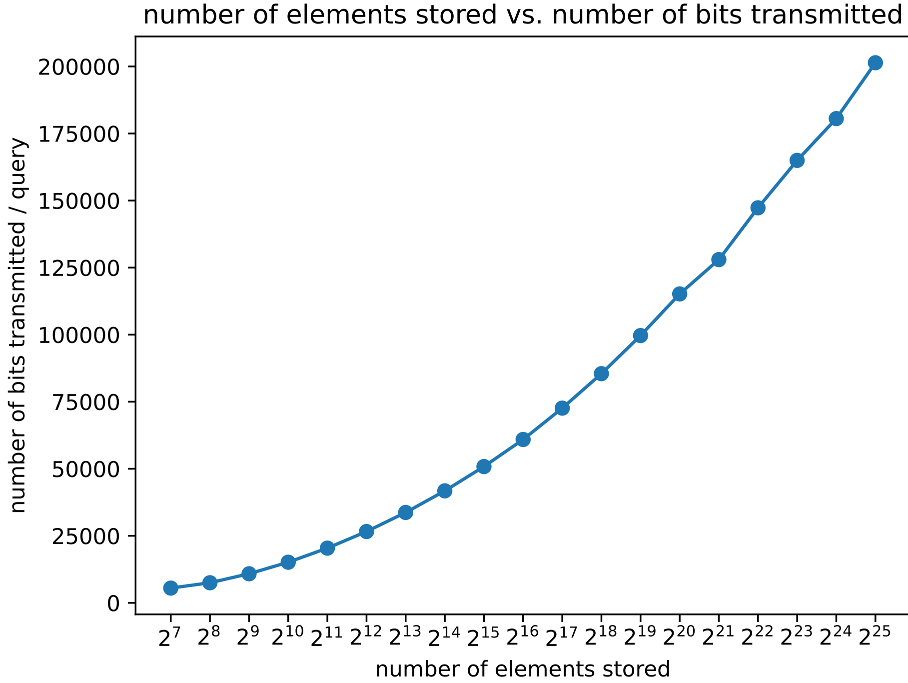

Figure 7.2: Number of elements stored vs. number of bits transmitted

7.2.2 Our PRAM's scalability with p

In the next sets of experiments, we keep n fixed at 2^{25}, and explore how the performance of our PRAM scales with p. We again set $m = k = n$, $w = \log^2(n)$, and $\mathsf{threshold} = p$. Remember that we always assume we have the same number of queries as the number of processors, hence while we have more parallelism available, we are also handling more and more queries at the same time, and it is not directly whether the increase in p will help us. Plots of our experimental findings are shown in Figure 7.5, Figure 7.6, and Figure 7.7, with the accompanying data in Table 7.2. In all plots, the x-axis is drawn on log-scale.

In this scenario, with both n and w fixed, the communication overhead plot may not be strictly necessary, as it merely down-scales the y-axis of Figure 7.5 by 625. However, we still include it to demonstrate that the communication overhead per bit is indeed small. Interestingly, the communication overhead decreases as p increases. This can be attributed to the less frequent refreshing steps as the number of processors, and consequently the $\mathsf{threshold}$, grows. Empirically, this highlights the importance of explicitly query handling the base case, as the work involved in refreshing is quite substantial in the lower level position maps.

The rounds of communication increase as p grows, but this is simply due to the larger $\mathcal{Q}$ that needs to be processed each time. Even so, we observe that the rounds of communication

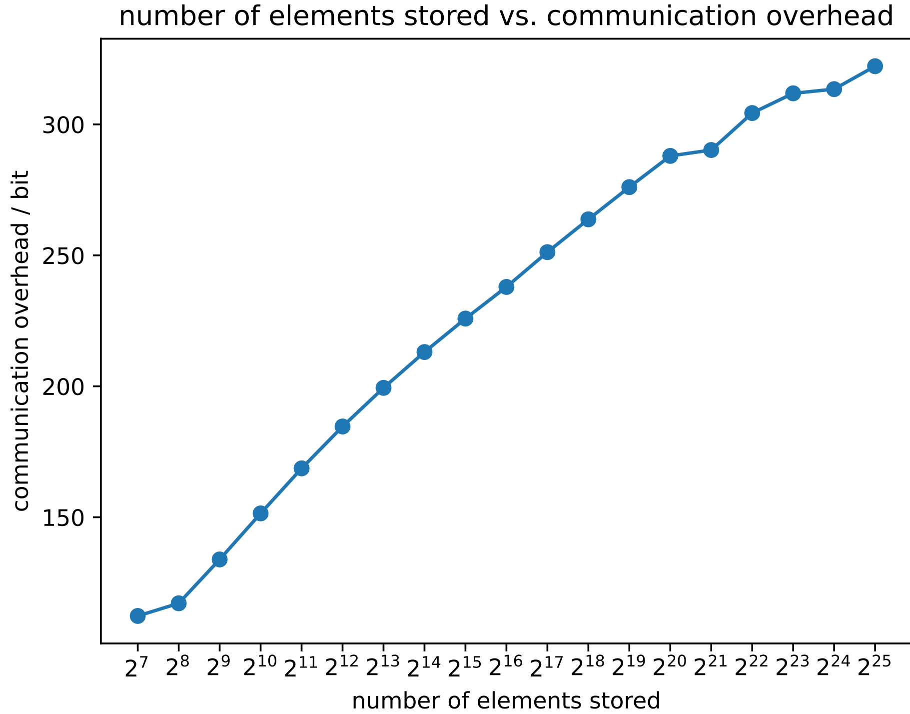

Figure 7.3: Number of elements stored vs. communication overhead

grow on a log-scale empirically, which is better than the $\log^2(n)$ asymptotics suggested by our calculations.

7.2.3 Experiment conclusion

Through our experiments, we find that the asymptotics suggested by calculations match with the empirical figures. In fact, the rounds complexity is even better in practice than in theory, only growing on a log-scale instead of $\log^2$ scale. Therefore, our PRAM is ideal in situations where the number of processors is on the same scale with the number of elements, since the rounds complexity grows very slowly even the number of queries we can handle grow exponentially. In other words, we pay a linear cost for an exponential gain.

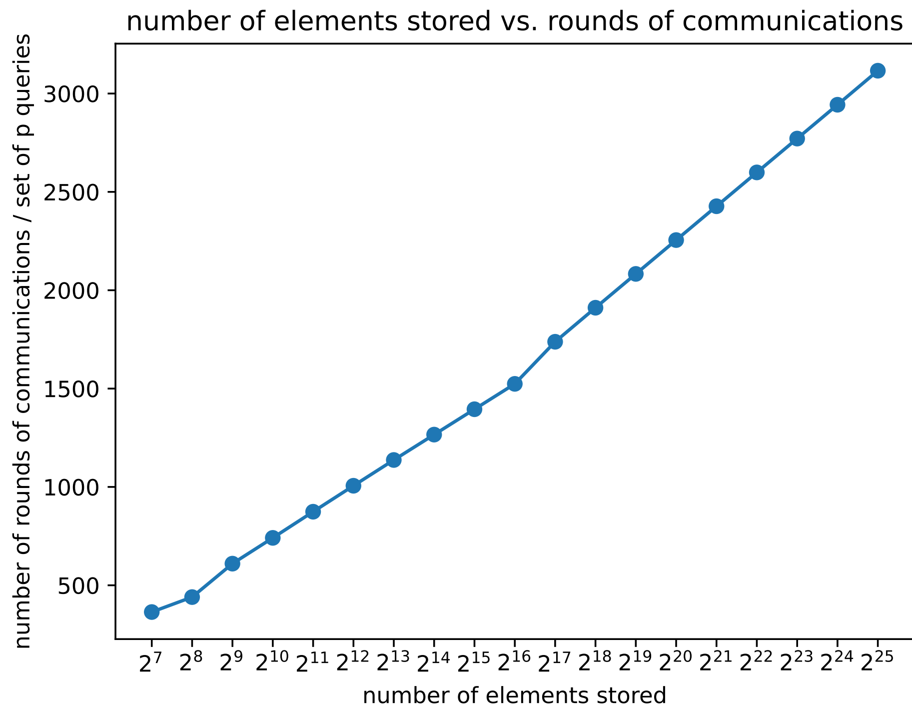

Figure 7.4: Number of elements stored vs. rounds of communications per set of p parallel queries

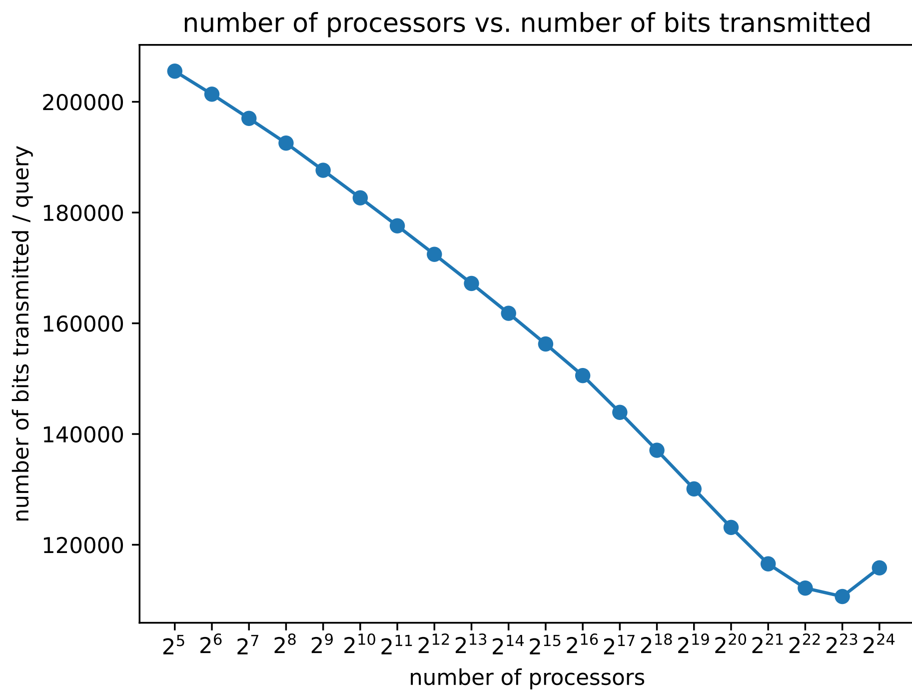

Figure 7.5: Number of elements stored vs. number of bits transmitted

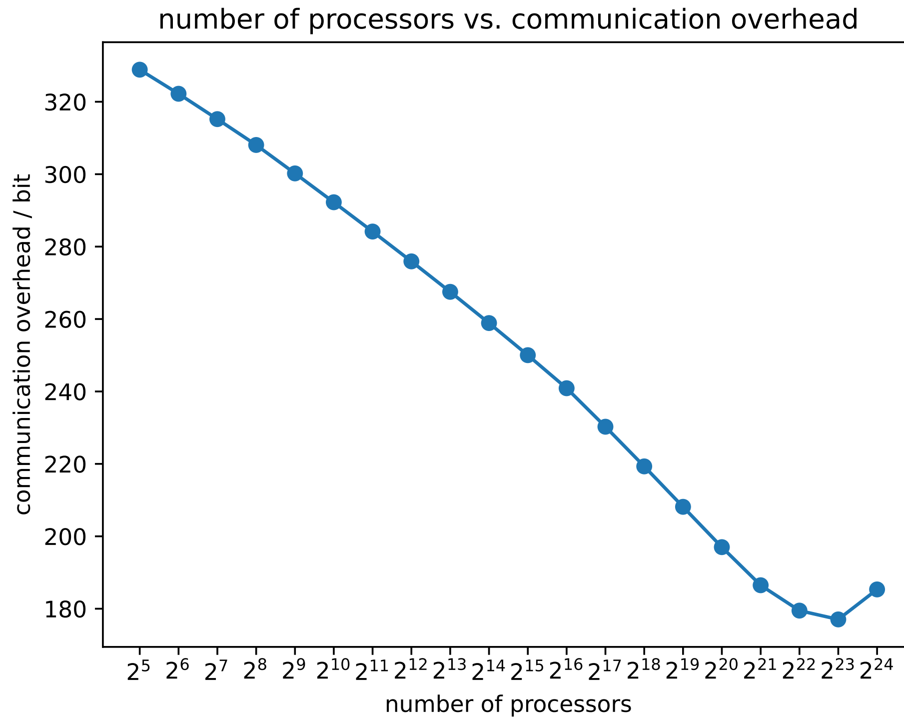

Figure 7.6: Number of elements stored vs. communication overhead

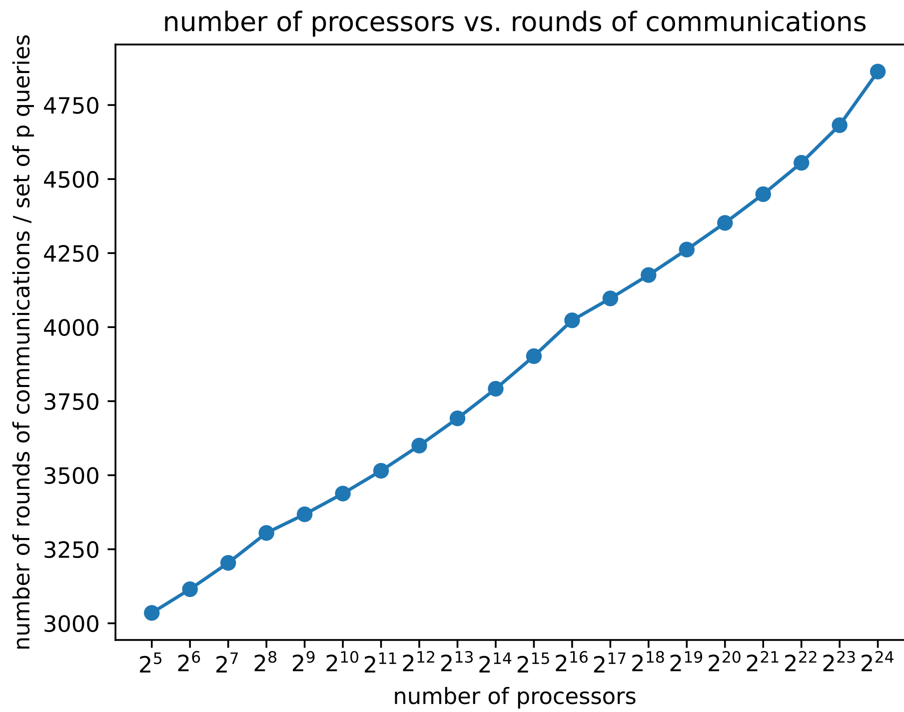

Figure 7.7: Number of elements stored vs. rounds of communications per set of p parallel queries

$\log(n)$	bits	rounds
7	5503	364
8	7497	440
9	10845	610
10	15147	741
11	20407	874
12	26589	1006
13	33698	1137
14	41761	1266
15	50817	1395
16	60909	1524
17	72601	1738
18	85455	1911
19	99653	2083
20	115186	2255
21	127982	2427
22	147293	2599
23	164971	2771
24	180552	2943
25	201380	3116

Table 7.1: Experiment with changing number of elements stored

$\log(p)$	bits	rounds
5	205537	3035
6	201380	3115
7	197006	3204
8	192542	3305
9	187635	3368
10	182657	3438
11	177601	3515
12	172453	3600
13	167194	3692
14	161804	3792
15	156263	3902
16	150557	4023
17	143912	4097
18	137068	4176
19	130088	4262
20	123130	4352
21	116554	4449
22	112174	4555
23	110655	4682
24	115831	4863

Table 7.2: Experiment with changing number of processors

CHAPTER 8: CONCLUSIONS

In this work, we introduced innovative techniques for achieving 3PC secure PRAM computation and constructed a perfectly secure one, as shown in Theorem 2.1. We began by developing a RAM that enables users to make queries without revealing the target logical address or the physical access pattern. Our RAM is most notable for its natural compatibility with parallelism. We then demonstrated how to generalize this RAM to a PRAM capable of handling a large number of queries in parallel. Empirically, we observed that both the communication overheads and round complexity grow logarithmically when $p = O(n)$.

The security and efficiency of our PRAM make it suitable for applications where the available degree of parallelism is high, such as the cloud map application discussed in the introduction chapter. We formally proved that our PRAM is perfectly secure in the 3PC semi-honest honest-majority setting, and we argued that our PRAM is more powerful than a traditional OPRAM as a result. One area where our PRAM outperforms traditional OPRAM is its natural compatibility with MPC-RAM protocols. For instance, the client of our PRAM can be an MPC protocol operating on secret shares, like a circuit running the GMW protocol. In this case, the MPC protocol can directly incorporate our PRAM by treating it as an idealized protocol, thereby transforming the MPC protocol from the circuit computation model to the RAM computation model, which is arguably easier to work with.

In conclusion, our PRAM represents a significant step towards achieving practical and secure PRAM computation. We hope that future research in this area will build on our findings and further develop this work.

REFERENCES

[1] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel RAM and applications," 2016, pp. 175–204.

[2] O. Goldreich, *Foundations of Cryptography: Basic Applications.* Cambridge, UK: Cambridge University Press, 2004, vol. 2.

[3] E. Stefanov, E. Shi, and D. X. Song, "Towards practical oblivious RAM," 2012.

[4] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," 1987, pp. 182–194.

[5] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, p. 431–473, may 1996. [Online]. Available: https://doi.org/10.1145/233551.233553

[6] E. Boyle and M. Naor, "Is there an oblivious RAM lower bound?" 2016, pp. 357–368.

[7] K. G. Larsen and J. B. Nielsen, "Yes, there is an oblivious RAM lower bound!" 2018, pp. 523–542.

[8] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," 2013, pp. 299–310.

[9] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," 2015, pp. 850–861.

[10] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel RAM," 2017, pp. 567–597.

[11] T.-H. H. Chan, K. Nayak, and E. Shi, "Perfectly secure oblivious parallel RAM," 2018, pp. 636–668.

[12] T.-H. H. Chan and E. Shi, "Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs," 2017, pp. 72–107.

[13] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel RAM: Improved efficiency and generic constructions," 2016, pp. 205–234.

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

[15] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," 2013, pp. 253–267.

[16] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, “Verifiable oblivious storage,” 2014, pp. 131–148.

[17] J. L. Dautrich Jr., E. Stefanov, and E. Shi, “Burst ORAM: Minimizing ORAM response times for bursty access patterns,” 2014, pp. 749–764.

[18] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs, “Optimizing ORAM and using it efficiently for secure computation,” 2013, pp. 1–18.

[19] K. Cong, D. Das, G. Nicolas, and J. Park, “Panacea: Non-interactive and stateless oblivious RAM,” Cryptology ePrint Archive, Report 2023/274, 2023, https://eprint.iacr.org/2023/274.

[20] A. C.-C. Yao, “How to generate and exchange secrets (extended abstract),” 1986, pp. 162–167.

[21] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game or A completeness theorem for protocols with honest majority,” 1987, pp. 218–229.

[22] E. Boyle, K.-M. Chung, and R. Pass, “Large-scale secure computation: Multi-party computation for (parallel) RAM programs,” 2015, pp. 742–762.

[23] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky, “Efficient 3-party distributed ORAM,” 2020, pp. 215–232.

[24] S. Faber, S. Jarecki, S. Kentros, and B. Wei, “Three-party ORAM for secure computation,” 2015, pp. 360–385.

[25] S. Jarecki and B. Wei, “3PC ORAM with low latency, low bandwidth, and fast batch retrieval,” 2018, pp. 360–378.

[26] J. Doerner and a. shelat, “Scaling ORAM for secure computation,” 2017, pp. 523–535.

[27] X. S. Wang, Y. Huang, T.-H. H. Chan, a. shelat, and E. Shi, “SCORAM: Oblivious RAM for secure computation,” 2014, pp. 191–202.

[28] S. Lu and R. Ostrovsky, “Distributed oblivious RAM for secure two-party computation,” Cryptology ePrint Archive, Report 2011/384, 2011, https://eprint.iacr.org/2011/384.

[29] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, “High-throughput semi-honest secure three-party computation with an honest majority,” 2016, pp. 805–817.

[30] L. Braun, M. Pancholi, R. Rachuri, and M. Simkin, “Ramen: Souper fast three-party computation for ram programs,” Cryptology ePrint Archive, Paper 2023/310, 2023, https://eprint.iacr.org/2023/310. [Online]. Available: https://eprint.iacr.org/2023/310

[31] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, “Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs,” 2019, pp. 869–886.

[32] D. Heath, V. Kolesnikov, and R. Ostrovsky, “EpiGRAM: Practical garbled RAM,” 2022, pp. 3–33.

[33] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, “Practically efficient multi-party sorting protocols from comparison sort algorithms,” 2013, pp. 202–216.

[34] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, N. Kiribuchi, and B. Pinkas, “An efficient secure three-party sorting protocol with an honest majority,” Cryptology ePrint Archive, Report 2019/695, 2019, https://eprint.iacr.org/2019/695.

[35] S. Laur, J. Willemson, and B. Zhang, “Round-efficient oblivious database manipulation,” 2011, pp. 262–277.

[36] M. Bellare and B. Yee, “Forward-security in private-key cryptography,” Cryptology ePrint Archive, Report 2001/035, 2001, https://eprint.iacr.org/2001/035.

[37] M. Blanton and C. Yuan, “Binary search in secure computation,” Cryptology ePrint Archive, Report 2021/1049, 2021, https://eprint.iacr.org/2021/1049.

[38] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS ’68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968. [Online]. Available: https://doi.org/10.1145/1468075.1468121 p. 307–314.

[39] G. Blelloch, “Scans as primitive parallel operations,” *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[40] A. W. Services, “Amazon ec2 instance types - amazon web services,” https://aws.amazon.com/ec2/instance-types/, accessed: 2023-04-19.