ABSTRACT

Title of dissertation: EFFICIENT DATA-OBLIVIOUS COMPUTATION

Kartik Nayak
Doctor of Philosophy, 2018

Dissertation directed by: Professor Jonathan Katz, Professor Elaine Shi
Department of Computer Science

The rapid increase in the amount of data stored by cloud servers has resulted in growing privacy concerns for users. First, although keeping data encrypted at all times is an attractive approach to privacy, encryption may preclude mining and learning useful patterns from data. Second, companies are unable to distribute proprietary programs to other parties without risking the loss of their private code when those programs are reverse engineered. A challenge underlying both those problems is that how data is accessed – even when that data is encrypted – can leak secret information.

Oblivious RAM is a well studied cryptographic primitive that can be used to solve the underlying challenge of hiding data-access patterns. In this dissertation, we improve Oblivious RAMs and oblivious algorithms asymptotically. We then show how to apply our novel oblivious algorithms to build systems that enable privacy-preserving computation on encrypted data and program obfuscation.

Specifically, the first part of this dissertation shows two efficient Oblivious

RAM algorithms: 1) The first algorithm achieves sub-logarithmic bandwidth blowup while only incurring an inexpensive XOR computation for performing Private Information Retrieval operations, and 2) The second algorithm is the first perfectly-secure Oblivious Parallel RAM with $O(\log^3 N)$ bandwidth blowup, $O((\log m + \log\log N)\log N)$ depth blowup, and $O(1)$ space blowup when the PRAM has m CPUs and stores N blocks of data. The second part of this dissertation describes two systems – HOP and GraphSC – that address the problem of computing on private data and the distribution of proprietary programs. HOP is a system that achieves simulation-secure obfuscation of RAM programs assuming secure hardware. It is the first prototype implementation of a provably secure virtual black-box (VBB) obfuscation scheme in any model under any assumptions. GraphSC is a system that allows cloud servers to run a class of data-mining and machine-learning algorithms over users' data without learning anything about that data. GraphSC brings efficient, parallel secure computation to programmers by allowing them to express computation tasks using the GraphLab abstraction. It is backed by the *first* non-trivial parallel oblivious algorithms that outperform generic Oblivious RAMs.

EFFICIENT DATA-OBLIVIOUS COMPUTATION

by

Kartik Nayak

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Jonathan Katz, Chair/Advisor
Professor Elaine Shi, Co-Chair/Co-Advisor
Professor Michael Hicks
Professor Charalampos Papamanthou
Professor Lawrence Washington

Acknowledgments

First, I would like to thank my advisors Jonathan Katz and Elaine Shi for their constant guidance and support throughout my Ph.D. I started working with Jon when it was perhaps the most difficult phase for me as a graduate student. But he always showed faith in my ability and supported me throughout. Jon has always been kind, fair, forgiving, and just an amazing person to interact with. When discussing a research problem or a proposed solution with him, he would pause for a few seconds and then bombard me with questions and alternatives that seem entirely unrelated. Over time I realized that he would use those few seconds to navigate through the giant web of research that has happened in the last few decades to ensure that I do not reinvent the wheel as also incorporate those ideas to improve results. His ability to present difficult topics lucidly during research discussions and teaching is something that I am still trying to learn.

Elaine's enthusiasm and passion has constantly encouraged me to do high-quality research. I spent most of my time as a graduate student working together with her. She taught me to relentlessly work on research problems and also to present effectively at conferences, both of which I had no clue about when I started my Ph.D. Over the years, she introduced me to an amazing set of collaborators, which added a lot of perspective to the way I thought about research problems. Her positive energy was instrumental during my entire job search process. I will always remember discussing research problems with her in coffee shops (in College Park, Ithaca, Boston, Bay area, and other places), many skype discussions turned interesting results that started late in the night, and also some skype discussions that went on for 3-4 hours trying to arrive at a result.

I am very grateful to Professor Rajiv Gandhi without whom I would not have even considered doing a Ph.D. He believed in my ability to do a Ph.D. long before I knew what research was about!

I spent four summers doing four different internships. My mentors – Stratis, Nina, Udi, Vipul, Nishanth, Satya, Dahlia, Ittai, Manuel, Olya, and Felix, have all been very generous with their advice. Working in an area of research complementary to theirs helped me learn new ways and approaches to my work. I would like to especially thank Dahlia and Ittai for introducing me to the area of distributed computing and blockchains – their enthusiasm was contagious and helped me learn a lot. I got the opportunity to work with Tudor to write my first research paper. I am thankful for everything that I learnt from him during this period; his invaluable advice about a lot of aspects throughout my Ph.D. has helped me. I am grateful to Mike and Jeff for their support through difficult times in the last five years. I am also grateful to Mike, Babis, and Larry for serving on my dissertation committee.

I am thankful to fellow student collaborators: Chris - specifically for his mentorship in HOP, Ling - for showing how one should question every assumption and explore every option, and Xiao - his ingenious way of connecting different results and coming up with new ones never ceased to amaze me. I shared my office space MC2 with a wonderful group of colleagues and friends!

Endless discussions on arbitrary topics with my flatmates – Amit, Bhaskar, Manish, Rama, and Anshul made living at home enjoyable. Chai-pe-charcha and dinner-time discussions was something that I always looked forward to at home, better known as *5002*. My friends – Sudha, Manaswi, Soham, Pallabi, Meethu, and many others made weekend outings and festivals a lot more memorable.

Finally, I would like to thank my parents, Ravidas and Ranjeetha, my sister, Roopa, and my fiancée, Amritha, who have been an endless source of love and understanding throughout this period. Words cannot express the gratitude I owe them.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The rapid increase in the amount of data stored by cloud servers has resulted in growing privacy concerns for users. Keeping data encrypted at all times is an attractive approach to privacy; however, if this is done then cloud servers are unable to derive the benefits of mining user data for large-scale trends. A complementary problem is that of the distribution of proprietary programs. Presently, if a pharmaceutical company outsources its proprietary genomic testing algorithm, it would compromise its intellectual property. A challenge underlying both of these problems is that how data is accessed – even when the data is encrypted – can have significant privacy implications. For instance, even if we can operate on encrypted data, *how* a program (say binary search) makes decisions based on this data, can reveal parts of the input (the element searched for).

To allow mining on user data while still ensuring privacy from the cloud provider, ideally, we would like to enable cloud servers to perform computations on encrypted user data. While cryptographic primitives such as functional encryption [26] can achieve this in theory, such primitives are impractical. An alternative is to leverage efficient multiparty computation [66, 149] run by two servers who each hold shares of user data such that they only learn an aggregate result and nothing

about individual user inputs. A key challenge here is to express the computation efficiently such that the memory trace produced by the computation is *input-data independent*.

In the first problem, the input to the program (i.e., user data) is hidden whereas the program itself (e.g., a data-mining algorithm) is not. Protecting proprietary software is the complementary problem where the program should remain hidden but can be executed by a third party on inputs of its choice. This is generally known as *obfuscation*. Although there has recently been theoretical progress on software-only obfuscation, it is extremely inefficient in practice [98].

Finally, an issue that arises in both the above problems is that memory accesses can leak information. For example, when a user accesses files stored in the cloud, even when those files are encrypted, the memory locations accessed may reveal information about when a particular file is being read. Similarly, when a program accesses memory, the locations in memory being read can reveal information about the internal workings of the program. Informally, data obliviousness means that the physical memory locations accessed are independent of secret information (the total number of accesses are still revealed).

Oblivious RAM (ORAM) introduced by Goldreich and Ostrovsky is a technique for making any RAM program data-oblivious [65, 67]. In this dissertation, we design efficient oblivious techniques and prototype systems to address the problems of computing on encrypted data and protecting proprietary programs. Moreover, we design novel Oblivious RAM algorithms that are efficient and secure.

In this chapter, we will first explain the problem addressed by an Oblivious

RAM and the high-level approach used in the seminal work of Goldreich and Ostrovsky (Section 1.1). We will present a survey of the relevant literature to explain the different approaches used by subsequent works to strengthen the results by Goldreich and Ostrovsky (Section 1.2). Finally, in Section 1.3, we will describe the high-level results in this dissertation. Specifically: 1) our contributions on the design of ORAM algorithms, and 2) our two prototype systems, HOP and GraphSC.

1.1 Protecting Memory Access Patterns via an Oblivious RAM

Oblivious RAM is a cryptographic primitive that hides a program's access patterns to sensitive data. The scenario envisioned by Goldreich and Ostrovsky consisted of programs that are executed on a shielded CPU that is communicating with untrusted memory. During this execution, an adversary observes the execution and tries to learn about the program and its inputs. The shielded CPU has the following characteristics: 1) it is assumed to have a small number of CPU registers as trusted storage, 2) it is assumed to be tamper-proof, i.e., an adversary cannot modify or observe the contents of the CPU registers, and 3) it stores a symmetric secret key that is used to keep the program and data in the memory encrypted. The contents of the memory itself are observable, although they are kept encrypted by the CPU. Moreover, whenever a program is executed, in addition to the state of the memory, the adversary can also observe the memory location that is being fetched or written to.

Intuitively, given that the memory is always encrypted, the key information

available to the adversary is the memory locations that are accessed, whether the access is a read or a write, the times at which these accesses are made, and potentially some other side-channels such as heat dissipation, etc. Among these, an Oblivious RAM answers the following question:

How can a CPU constrained to a few registers access memory without revealing ***which*** *memory location it wants to access?*

```
1: if (secret)
2:     read mem[x]
3: else
4:     read mem[y]
5: ...
```

Figure 1.1: Memory access pattern revealing secret variables

At first sight, one may ask – if an adversary is trying to learn information about a program or its inputs, how does revealing the memory location help the adversary? To understand this, let us consider a simple program shown in Figure 1.1. Suppose the program itself is public and known to the adversary but input variable `secret` should be hidden. Now, depending on the value of `secret`, the program accesses either memory location x or memory location y. This clearly shows that access patterns can reveal secrets from a program.

While the example in Figure 1.1 is simplistic, similar attacks have been used in practice. Ohrimenko et al. [121] used network level access patterns during a MapReduce job to infer sensitive data from an encrypted census dataset. Simplified

to the setting of a shielded CPU and an encrypted untrusted memory, their program computed the frequency of a specific attribute, say, age of the population. This attribute was stored on one array A: each entry of the array referred to a person in the census database. The frequency of ages was computed on another array F: the i-th entry in this array stored the number of people with age i. Initially, F stores all 0s. To compute F, the program would scan through A and increment the data stored at the appropriate index in F. While the scan through A is data-independent, *which* index is accessed in F clearly reveals the age of a person. In effect, age is analogous to the `secret` variable in our example from Figure 1.1, which is revealed by the exact memory location accessed in F.

A generalization of the same idea from Figure 1.1 has also been used by Xu et al. [148]. They use Intel Software Guard Extensions (Intel SGX) [45] as their shielded CPU and could observe only page faults through an untrusted operating system. One of the programs that they considered was used for JPEG decoding. In the program, whenever a portion of the input image consisted of all *zeroes* (analogous to the `secret` variable), the decoding algorithm would incur fewer page-faults than otherwise (analogous to reading memory location x, vs. memory location y). Thus, the number of page-faults effectively reveals a portion of the contents of the encrypted image.

Although these attacks were discovered relatively recently, the theoretical study of protecting access patterns started with the seminal work of Goldreich and Ostrovsky and subsequently improved in a series of works over the last three decades. Before explaining these results, we will make a short remark on the terminology used

in this and subsequent chapters. As explained earlier, a shielded CPU stores the secret key and is trusted to observe the logical access pattern. The memory, on the other hand, is always assumed to store encrypted data observable to the adversary. Moreover, if the adversary has physical access to the data bus between the CPU and memory, he/she can modify the data being passed to/from the memory. Thus, we can also think of the memory as being stored by the adversary. Alternatively, one can model the same idea in a cloud-outsourcing scenario where a client wishes to outsource data blocks to an untrusted server (the cloud). Consequently, we will interchangeably use the term 'client' to refer to the trusted CPU and the term 'server' to refer to an adversary storing the memory. Moreover, we also refer to the data stored in a memory location as a memory block, or a data block, or simply, a block.

A naïve solution. As a naïve solution, a CPU that wants to read a memory location addr could read *all* memory locations. It stores the contents of addr in one of its registers and it ignores the contents of other memory locations. Writes to a memory location are handled in a similar manner – read all memory locations and rewrite the unmodified content back to each of the locations other than addr. For memory location addr, write the content from the register. Recall that the adversary only has access to the encrypted memory, but does not have the key to decrypt it. So, assuming the encryption algorithm is secure, all that the adversary observes is that all memory locations have been accessed. The different treatment of addr happens within the processor and cannot be detected by the adversary. While this solution works, the obvious downside to this approach is the number of memory locations that are read for each memory location that needs to be accessed, a metric

that is also referred to as *bandwidth blowup*. In the naïve scheme, the bandwidth blowup is $O(N)$ when N memory locations are stored memory.

Algorithms like this naïve ORAM scheme, which on execution result in a memory-access trace that is independent of input data, are called "data oblivious algorithms." An ORAM construction makes any RAM program data oblivious. Intuitively, an ORAM construction is said to be secure if (i) for any two memory access sequences $\vec{y}$ and $\vec{z}$ such that $|\vec{y}| = |\vec{z}|$, their access patterns $A(\vec{y})$ and $A(\vec{z})$ are indistinguishable to anyone but the processor, and (ii) the ORAM construction is correct in the sense that it returns on input $\vec{y}$; output data that is consistent with $\vec{y}$ with all-but-negligible probability.

The seminal result. For any construction that does not scan the entire memory, it turns out that the key ingredient to obliviousness is to shuffle memory locations after one or more accesses. If the adversary does not know the permutation used for shuffling, it will not be able correlate it to the input request sequence. Goldreich and Ostrovsky used this idea in their ORAM construction. Their construction stores a hierarchy of $O(\log N)$ levels that are geometrically increasing in size. Specifically, level i is capable of storing 2^i memory locations. One could think of this hierarchical data structure as a hierarchy of caches where smaller levels act as stashes for larger levels. Each level is individually permuted and a memory location can be looked up using a pseudo-random function (PRF) whose secret key is only known to the CPU but not the adversary. To access a memory location at logical address addr, the CPU sequentially looks in every level of the hierarchy (from small to large) for the logical address addr. Once the block has already been found in some level,

for all subsequent levels the CPU just looks for a dummy element, denoted by $\perp$. When a requested block has been found, it is marked as deleted in the corresponding level where it is found. Every 2^i memory requests, a rebuild operation is performed to merge all levels smaller than i (including the memory location just fetched and possibly updated if it was a write request) into level i — at this moment, level i is considered to be rebuilt. Intuitively, this scheme is secure, because every block is always accessed from a permuted list such that the adversary does not know the permutation used. By laying out the memory in hierarchical layers, they improved the bandwidth blowup of the naïve scheme from $O(N)$ to an amortized blowup of $O(\log^3 N)$.

We make multiple observations about this result and the model considered by Goldreich and Ostrovsky. First, since they use a PRF, the access patterns are indisinguishable to a computationally-bounded adversary. Second, their model assumes the server to be a simple storage device that is capable of only read and write operations. Third, they assumed the number of bits stored in a memory location to always be $\Omega(\log N)$ bits. Finally, they only consider a model where there is exactly one server and exactly one client.

1.2 A Short Literature Survey

Over the last three decades, several works have improved on the seminal result. Some of these have been algorithmic improvements while many of the results have modified the modeling assumptions used by Goldreich and Ostrovsky. We

next describe a short summary of the relevant results classified by the modeling assumptions that they use.

1. **Security guarantees: Computational vs. statistical vs. perfect.** The original construction had access patterns that were secure only against a computationally-bounded adversary. Ajtai [11] showed the first ORAM construction that is statistically-secure with a bandwidth blowup of $O(\log^3 N)$. This was followed by the statistically-secure ORAM construction by Shi et al. [134], who introduced the tree-based paradigm. ORAM constructions in the tree-based paradigm have improved the bandwidth blowup from $O(\log^3 N)$ to $O(\log^2 N)$ [43, 61, 134, 140, 143]. Most tree-based ORAMs achieved statistical access pattern security, and obtained the desired bandwidth blowup in the worst-case instead of an amortized blowup.

   Damgård et al. were the first to study a *perfectly* secure ORAM and to show a construction with $O(\log^3 N)$ bandwidth blowup and storing $O(N \log N)$ memory blocks on the server [47].

2. **Non-uniform/large block sizes.** Goldreich and Ostrovsky assumed that the block size addressable by the ORAM is the same as the size of the memory location. Their results hold for any block size $\Omega(\log N)$ bits. However, if we are allowed to assume non-uniform block sizes, i.e., the data blocks addressable in the ORAM is larger than the word size supported by the underlying RAM, then the metadata required by the ORAM algorithm to access a block is asymptotically smaller than the cost to access a block. In such a scenario,

the effective bandwidth blowup can be computed as the ratio of the number of bits transferred while making an ORAM access to the size of a data block. Using this idea and assuming block sizes to be $\Omega(\log^2 N)$ bits, certain tree based ORAM constructions can be improved to achieve a bandwidth blowup of $\omega(\log N)$ [140, 143].

3. **Assuming server computation.** Goldreich and Ostrovsky assume the server to be a read/write store. However, one can reduce the number of data blocks transferred by allowing the server to perform some computation on the data blocks [12, 48, 51, 62, 110, 115, 126, 137, 138, 146, 147, 153]. The key results in this category are the ones by Apon et al. [12], and Devadas et al. [51], which use fully homomorphic encryption and additively homomorphic (or somewhat homomorphic) encryption respectively to achieve an $O(1)$ bandwidth blowup.

4. **Assuming multiple non-colluding servers.** ORAMs in this category assume multiple non-colluding servers to improve bandwidth blowup [80, 97, 106, 144]. A representative construction is by Lu and Ostrovsky [106], which achieved a bandwidth blowup of $O(\log N)$.

5. **Oblivious Parallel RAM (OPRAM).** An Oblivious Parallel RAM (OPRAM) transforms a Parallel RAM (PRAM) program into a secure form such that the resulting PRAM's access patterns leak no information about secret inputs. It was first proposed by Boyle, Chung, and Pass [28], and subsequently improved in several followup works [35, 36, 38, 39, 117]. In addition to the bandwidth blowup, another metric that is relevant for an OPRAM is the depth

blowup. Here, depth characterizes the parallel runtime of a program assuming ample number of CPUs. State of the art OPRAM schemes are statistically secure and achieve a bandwidth blowup of $O(\log^2 N)$ and a depth blowup of $O(\log N \log\log N)$ [35].

1.3 Outline of the Dissertation

In this section, we will describe, at a high-level, the results presented in this dissertation.

1.3.1 An Oblivious RAM with a Sub-logarithmic Bandwidth Blowup

In Chapter 3, we describe a tree-based Oblivious RAM scheme with a sub-logarithmic bandwidth blowup of $O(\log_d N)$ (where d is a free parameter) assuming the servers are capable of handling non-uniform block sizes, there are multiple non-colluding servers that are capable of performing inexpensive computation. We also show a $\Omega(\log_{cD} N)$ lower bound on bandwidth blowup in the modified model involving PIR operations. Here, c is the number of blocks stored by the client and D is the number blocks on which PIR operations are performed. Our construction matches this lower bound implying that the lower bound is tight for certain parameter ranges.

Chapter 3 is based on a paper I co-authored with Ittai Abraham, Christopher W. Fletcher, Benny Pinkas, and Ling Ren and is published in Public Key Cryptography (PKC), 2017 [8].

1.3.2 A Perfectly Secure Oblivious Parallel RAM

In Chapter 4, we show that PRAMs can be obliviously simulated with perfect security, incurring only $O(\log N(\log m + \log\log N))$ blowup in parallel runtime, $O(\log^3 N)$ blowup in total work, and $O(1)$ blowup in space relative to the original PRAM. Prior to our work, no perfectly secure Oblivious Parallel RAM (OPRAM) construction was known; and we are the first in this respect. Even for the sequential special case of our algorithm (i.e., perfectly secure ORAM), we not only achieve logarithmic improvement in terms of space consumption relative to the state-of-the-art, but also significantly simplify perfectly secure ORAM constructions.

Chapter 4 is based on a paper co-authored with Elaine Shi and T-H. Hubert Chan, and is currently in submission to a conference [37].

1.3.3 Executing Obfuscated Programs using HOP

Program obfuscation is a central primitive in cryptography, and has important real-world applications in protecting software from IP theft. However, well known results from the cryptographic literature have shown that software only virtual black box (VBB) obfuscation of general programs is impossible. In Chapter 5 we propose HOP, a system (with matching theoretical analysis) that achieves simulation-secure obfuscation for RAM programs, using secure hardware to circumvent previous impossibility results. To the best of our knowledge, HOP is the *first* implementation of a provably secure VBB obfuscation scheme in any model under any assumptions.

HOP trusts only a hardware single-chip processor. We present a theoretical

model for our hardware design and prove its security in the UC framework. Our goal is both provable security and practicality. To this end, our theoretic analysis accounts for the optimizations used in our practical design, including the use of a hardware Oblivious RAM (ORAM), hardware scratchpad memories, and context switching. We then detail a prototype hardware implementation of HOP. The prototype design requires 72% of the area of a V7485t Field Programmable Gate Array (FPGA) chip. Evaluated on a variety of benchmarks, HOP achieves an overhead of $8\times \sim 76\times$ relative to an insecure system.

Chapter 5 is based on a paper co-authored with Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal, and is published at Network and Distributed Systems Security (NDSS), 2017 [116].

1.3.4 Parallel Secure Computation for Graph-parallel Algorithms

In Chapter 6, we propose parallel oblivious algorithms to enable a secure execution of "graph-parallel algorithms" on large datasets using secure computation. We present GraphSC, a framework that (i) provides a programming paradigm that allows non-cryptography experts to write secure code; (ii) brings parallelism to such secure implementations; and (iii) meets the needs for obliviousness, thereby not leaking any private information. Using GraphSC, developers can efficiently implement an oblivious version of graph-based algorithms that execute in parallel with minimal communication overhead. Specifically, for a graph with V vertices and E edges, and $N = |\mathsf{V}| + |\mathsf{E}|$, the primitives "Scatter" and "Gather" in GraphSC can be imple-

mented with $O(N \log N)$ total work and $O(\log N)$ depth. Thus, our secure version of graph-based algorithms incurs only a small logarithmic overhead in comparison with the non-secure parallel version. We build GraphSC and demonstrate, using several algorithms as examples, that secure computation can be brought into the realm of practicality for big data analysis. Our secure matrix factorization implementation can process 1 million ratings in 13 hours, which is a multiple order-of-magnitude improvement over the only other existing attempt, which requires 3 hours to process 16K ratings.

Chapter 6 is based on a paper co-authored with Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi and is published at IEEE Security and Privacy (SP), 2015 [118].

Chapter 2: Definitions and Preliminaries

2.1 Parallel Random-Access Machines

We review the concepts of a parallel random-access machine (PRAM) and an oblivious parallel random-access machine (OPRAM). Although the definitions are only provided for the parallel case as required in Chapter 4, we point out that this is without loss of generality, since a sequential RAM can be thought of as a special case of PRAM with one CPU.

Parallel Random-Access Machine (PRAM).

A *parallel random-access machine* (PRAM) consists of a set of CPUs and a shared memory denoted by mem indexed by the address space $\{0, 1, \ldots, N-1\}$, where N is a power of 2. We refer to each memory word also as a *block*, which is at least $B = \Omega(\log N)$ bits long.

We consider a PRAM where the number of CPUs is m and the i-th CPU is denoted by $\mathsf{cpustate}_i$. Suppose the input to the PRAM program PRAM is denoted by inp. In each step, each CPU executes a next instruction circuit $\Pi(\mathsf{cpustate}_i, \mathsf{rdata}_i)$ based on its internal state and the most recently fetched B bits of data denoted rdata_i. It updates its CPU state $\mathsf{cpustate}_i$; and further, CPUs interact with memory through request instructions $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m])$. Specifically, at time step t,

CPU i's instruction is of the form $I_i^{(t)} := (\texttt{read}, \mathsf{addr})$, or $I_i^{(t)} := (\texttt{write}, \mathsf{addr}, \mathsf{wdata})$ where the operation is performed on the memory block with address addr and the block content wdata.

If $I_i^{(t)} = (\texttt{read}, \mathsf{addr})$ then CPU i should receive the contents of $\mathsf{mem}[\mathsf{addr}]$ at the beginning of time step t and is stored in rdata_i. Else if $I_i^{(t)} = (\texttt{write}, \mathsf{addr}, \mathsf{wdata})$, CPU i should still receive the contents of $\mathsf{mem}[\mathsf{addr}]$ at the beginning of time step t; further, at the end of step t, the contents of $\mathsf{mem}[\mathsf{addr}]$ should be updated to wdata.

Write conflict resolution. By definition, multiple `read` operations can be executed concurrently with other operations even if they visit the same address. However, if multiple concurrent `write` operations visit the same address, a conflict resolution rule will be necessary for our PRAM to be well-defined. In this dissertation, we assume the following:

- The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitary, parametrizable rule for write conflict resolution. In other words, there exists some priority rule to determine which `write` operation takes effect if there are multiple concurrent writes in some time step t.

- Our compiled, oblivious PRAM (defined below) is a "concurrent read, exclusive write" PRAM (CREW). In other words, our OPRAM algorithm must ensure that there are no concurrent writes at any time.

We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more

powerful but our compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [77].

Henceforth, we assume that *each CPU can only store $O(1)$ memory blocks.* Further, we assume that the runtime T of the PRAM is fixed and *publicly known.* Thus, PRAM is a PRAM program that belongs to a family of programs $\mathcal{PRAM}[\Pi, T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, B, m]$. For any program that is run on the PRAM, ℓ_{in} and ℓ_{out} denote the input and output lengths respectively (in terms of number of words).

ℓ_{in} and ℓ_{out} are relevant in Chapter 5 when the adversary executes obfuscated programs on inputs of his choice to obtain the corresponding output. Moreover, in chapter 5, we only use the special case of $m = 1$, i.e., RAM programs. Finally, in all of the chapters except Chapter 5, we consider PRAMs that are *stateful* and can evaluate a sequence of inputs, carrying state between them. Without loss of generality, we assume each input can be stored in a single memory block.

CPU-to-CPU communication. Some of the algorithms in our constructions involve CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, some earlier works [39] adopt separate metrics for inter-CPU

communication.

2.2 Oblivious Parallel Random-Access Machines

An OPRAM is a (randomized) PRAM with certain security properties, i.e., its access patterns leak no information about the inputs to the PRAM.

Randomized PRAM. A *randomized PRAM* is a special PRAM where the CPUs are allowed to generate private random numbers.

Memory access patterns. Given a PRAM program denoted PRAM and a sequence inp of inputs, we define the notation $\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp})$ as follows:

- Let T be the total number of parallel steps that PRAM takes to evaluate inputs inp.
- Let $A_t := (\mathsf{addr}_1^t, \mathsf{addr}_2^t, \ldots, \mathsf{addr}_m^t)$ be the list of addresses such that the i-th CPU accesses memory address addr_i^t in time step t.
- We define $\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp})$ to be the random object $[A_t]_{t\in[T]}$.

Oblivious PRAM (OPRAM). We say that a PRAM is *oblivious*, iff for any two input sequences inp_0 and inp_1 of equal length and for a negligible function $\epsilon(.)$, it holds that

$$\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_0) \overset{\epsilon(N)}{\equiv} \mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_1)$$

If a computationally-bounded adversary can distinguish the addresses produced on inp_0 and inp_1 with probability $\leq \epsilon(N)$, we say that the OPRAM is computationally secure. Similarly, if an unbounded adversary can distinguish between

the distribution of addresses with probability $\leq \epsilon(N)$, we say that the OPRAM is statistically secure. If the distribution of addresses are identically distributed, i.e., $\epsilon(N) = 0$, for all inputs, the OPRAM is said to be perfectly secure.

In this dissertation, we always assume that the original PRAM has a fixed number of CPUs (denoted m) in all steps of execution. For the compiled OPRAM, we consider two models 1) when the OPRAM always consumes exactly m CPUs in every step (i.e., the same number of CPUs as the original PRAM); and 2) when the OPRAM can consume an unbounded number of CPUs in every step; in this case, the actual number of CPUs consumed in each step may vary. However, for an ORAM scheme, we always assume that the compiled ORAM uses a single CPU.

Oblivious parallel algorithms. If the PRAM program or the parallel algorithm belongs to a class of algorithms $\mathcal{ALG}$ that is fixed and available to the adversary, we can define the notion of an *oblivious parallel algorithm*. A parallel algorithm $\mathsf{ALG} \in \mathcal{ALG}$ is said to be *oblivious*, iff for any two inputs inp_0 and inp_1 to the algorithm such that $|\mathsf{inp}_0| = |\mathsf{inp}_1|$ and the adversary knows $\mathcal{ALG}$, and for a negligible function $\epsilon(.)$, it holds that

$$\mathsf{Addresses}[\mathsf{ALG}](\mathsf{inp}_0) \overset{\epsilon(N)}{\equiv} \mathsf{Addresses}[\mathsf{ALG}](\mathsf{inp}_0)$$

If the addresses are indistinguishable only to a computationally-bounded adversary, we say that the oblivious algorithm is computationally-secure. Statistical and perfect security are analogously defined.

Oblivious simulation metrics. We adopt the following metrics to characterize

the overhead of (parallel) oblivious simulation of a PRAM.

- *Simulation overhead (when the OPRAM consumes the same number of CPUs as the PRAM).* If a PRAM that consumes m CPUs and completes in T parallel steps can be obliviously simulated by an OPRAM that completes in $\gamma \cdot T$ steps also with m CPUs (i.e., the same number of CPUs as the original PRAM), then we say that the simulation overhead is γ. Note that this means that every PRAM step is simulated by *on average* γ OPRAM steps.

- *Bandwidth blowup (when the OPRAM consumes the same number of CPUs as the PRAM).* If a PRAM that consumes m CPUs and accesses B bits of information can be obliviously simulated by an OPRAM that accesses $\gamma \cdot B$ bits, also with m CPUs, then we say that the bandwidth blowup is γ.

- *Total work blowup (when an unbounded number of CPUs maybe consumed).* A PRAM's total work is the number of steps necessary to simulate the PRAM under a single CPU, and is equal to the sum $\sum_{t \in [T]} m_t$. If a PRAM of total work W can be obliviously simulated by an OPRAM of total work $\gamma \cdot W$ we say that the total work blowup of the oblivious simulation is γ. Similarly, for a PRAM algorithm of total work W can be obliviously simulated by an oblivious parallel algorithm of total work $\gamma \cdot W$, we say that the total work blowup of the oblivious simulation is γ.

- *Depth blowup (when an unbounded number of CPUs maybe consumed).* A PRAM's depth is defined to be its parallel runtime when there are an unbounded number of CPUs. If a PRAM of depth D can be obliviously simulated

by an OPRAM of depth $\gamma \cdot D$ we say that the depth blowup of the oblivious simulation is γ. Similarly, for a PRAM algorithm of depth D can be obliviously simulated by an oblivious parallel algorithm of depth $\gamma \cdot D$, we say that the depth blowup of the oblivious simulation is γ.

- *Space blowup*. Space blowup refers to the multiplicative blowup in space required on the server when comparing the OPRAM (or ORAM) and that of the original PRAM (or RAM).

Note that the simulation overhead and bandwidth blowup are good standalone metric if we assume that the OPRAM must consume the same number of CPUs as the PRAM. If the OPRAM is allowed to consume more CPUs than the PRAM, we typically use the metrics total work blowup and depth blowup in conjunction with each other: total work blowup alone does not characterize how much the OPRAM preserves parallelism; and depth blowup alone does not capture the extent to which the OPRAM preserves total work.

Note that for ORAMs/OPRAMs with uniform block sizes, i.e., all blocks have the same size, simulation overhead and bandwidth blowup are exactly the same. For an ORAM, since we are only concerned with the scenario of using a single CPU, bandwidth blowup and simulation overhead are the only relevant metrics.

Finally, the following simple fact is useful for understanding the complexity of (oblivious) parallel algorithms.

Fact 1. *Let $C > 1$. If an (oblivious) parallel algorithm* Alg *can complete in T steps consuming m CPUs, then it can complete in CT steps consuming $\lceil \frac{m}{C} \rceil$ CPUs.*

2.3 Private Information Retrieval Protocols

Private information retrieval (PIR) allows a user to download one item from an unprocessed database known to a server, without revealing to the server which item is downloaded [41]. More formally, the setting has a server which is holding a list of records $Y = (y_1, y_2, \cdots, y_m)$, and a user who wants to download record y_i without revealing i to the server. A PIR scheme must enable this operation while requiring communication that is strictly smaller than the size of the database (otherwise, a trivial solution could have the user hide i by simply downloading the entire database.) This problem is similar to the problem addressed by an ORAM except for the following differences: (i) In PIR, the database records are usually public data records, or records which are owned by the server, and therefore the user cannot encrypt or otherwise preprocess them, and (ii) Typically, only read operations are performed on this database.

Two categories of PIR techniques exist – one operates in a setting with a single server and the other requires the existence of two or more non-colluding servers. Single-server PIR protocols, such as [30,64,94], have been adopted by Path-PIR [110] and Onion ORAM [51] to improve bandwidth. A downside, however, is that they require the server to perform operations on homomorphically encrypted ciphertexts [101], making server computation the new bottleneck. PIR in the presence of two or more non-colluding servers is conceptually simpler and involves much less computation — typically only simple XOR operations. It can also guarantee security against an unbounded adversary.

The original investigation of two-server PIR assumed that each database record is a single bit. The initial PIR paper described a two-server PIR protocol with $O(m^{1/3})$ communication [41] (and more efficient protocols with more than two servers). This result was only recently improved to obtain a communication of $m^{O(\sqrt{\log\log m/\log m})}$ [53].

In the setting of ORAM, we are interested in a PIR of long records, where the number of bits in each record $|y_j|$ is in the same order as the total number of records m. In this case there is a simple PIR protocol that was adopted in [122]: The database of records is replicated across the two servers, $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose that the user is interested in retrieving record i. For the request, the user generates a random bit string of length m, $X = (x_1, x_2, \cdots, x_m)$. He then generates $X' = (x'_1, x'_2, \cdots, x'_m)$ by flipping the i-th bit in X, i.e., $x'_i = \bar{x}_i$ and $x'_j = x_j$ for $j \neq i$. The user then sends X to $\mathcal{S}_1$, and X' to $\mathcal{S}_2$. $\mathcal{S}_1$ computes and responds with $\sum_j x_j \cdot y_j$ while $\mathcal{S}_2$ computes and responds with $\sum_j x'_j \cdot y_j$. Here, the sums represent a bit-wise XOR, and $\cdot$ represents a bit-wise AND. The user then sums up (XORs) the two responses to obtain $\sum_j (x_j + x'_j) \cdot y_j = y_i$. The above protocol is denoted as $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, Y, i)$. The communication overhead is $O(|y_j| + m) = O(|y_j|)$.

PIR-writes. Analogous to PIR, we can define PIR-write operations. A PIR-write operation lets a user update one record among a list of records on a server without revealing to the server which record is updated. Notice that now the records can no longer be public data; they have to be encrypted. Otherwise, the server can trivially figure out which record is updated by comparing their values before and after the update.

Chapter 3: Asymptotically Tight Bounds for Composing ORAM with PIR

The standard ORAM model assumes the server to be a simple storage device that only supports *read* and *write* operations. In this model, numerous works have improved the bandwidth blowup from $O(\log^3 N)$ to $O(\log N)$ where N is the number of logical data blocks. But none could achieve sub-logarithmic bandwidth blowup so far. In this sense, though not provably insurmountable [29], the $\Omega(\log N)$ bandwidth blowup barrier does seem hard to surpass.

To this end, a line of work deviates from the standard model and assumes the existence of two non-colluding servers [106, 122, 136, 144] with inexpensive server computation (e.g., XOR) or no server computation. But these constructions have been unable to surpass the $\Omega(\log N)$ bandwidth blowup barrier.

Another line of work allows the server to perform some computation. The most recent works involving server computation achieved $O(1)$ bandwidth blowup [12, 51, 114, 115]. But this improvement in bandwidth comes with a huge cost in the amount of server computation. In practice, in both schemes, the time for server computation will far exceed the time for server-client communication and become the new bottleneck.

Thus, the state of the art leaves the following natural question:

Can we construct a sub-logarithmic ORAM without expensive computation?

In this chapter, we answer this question positively with a concrete and secure construction. Our construction relies on a d-ary ORAM tree and a private information retrieval (PIR) protocol involving two non-colluding servers, where the servers perform simple XOR computations. Our construction achieves $O(\log_d N)$ bandwidth blowup with $c = O(1)$ blocks of client storage and PIR operations on $D = d \cdot \mathsf{polylog}(N)$ blocks. PIR is closely related to ORAM as they both hide access patterns. In fact, PIR is frequently used in ORAM constructions to improve bandwidth blowup [110, 114, 115, 122, 153]. This led us to ask the following question:

What is the asymptotically optimal bandwidth blowup one can achieve by using PIR in an ORAM construction?

In order to answer this question, we build on the seminal work of Goldreich and Ostrovsky [67] and derive a $\Omega(\log_{cD} N)$ bandwidth lower bound for ORAMs that leverage only PIR and PIR-write on top of the traditional model. Here, c is the number of blocks stored by the client and D is the number of blocks on which PIR/PIR-write operations are performed.

Our ORAM construction, in fact, matches this $\Omega(\log_{cD} N)$ lower bound when $d = \Omega(\log N)$, implying that under certain parameter ranges our construction is asymptotically optimal and the lower bound is asymptotically tight. Moreover, existing constructions such as C-ORAM [115] and CHf-ORAM [114] violate this lower bound, and thus cannot be secure.

We remark that there is a concurrent and independent work, MSKT-ORAM, that achieves comparable bandwidth blowup using similar techniques [154].[1] Our construction has several advantages over this work and we make a more detailed comparison in Section 3.5.

Our Contributions

Our contributions in this chapter can be summarized as follows:

1. **ORAM with sub-logarithmic bandwidth blowup.** We show a provably secure ORAM construction that achieves a bandwidth blowup of $O(\log_d N)$ (where d is a parameter) using $O(1)$ blocks of client storage. Our construction uses a d-ary tree and a PIR protocol (Section 3.2).

2. **Extending the Goldreich-Ostrovsky lower bound to allow PIR operation.** For a client storing c blocks of data and performing a PIR on D blocks at a time, we show that the ORAM bandwidth blowup is lower bounded by $\Omega(\log_{cD} N)$ (Section 3.4). Our construction matches this lower bound implying that the lower bound is tight and that our construction is asymptotically optimal for certain parameter ranges.

Table 3.1: **Comparison with existing Oblivious RAM schemes.** N denotes the number of logical blocks stored by the ORAM. In [126,140], a stash of $\Omega(\lambda)$ blocks ensures a failure probability of $e^{\Omega(-\lambda)}$. For a negligible (in N) failure probability, these works set $\lambda = \omega(\log N)$. Our work requires two non-colluding servers while others require a single server.

Construction	**Bandwidth Blowup**	**Client Storage**	**Block Size**	**Server Computation**
Path ORAM [140]	$8 \log N$	$O(\lambda)$	$\Omega(\log^2 N)$	-
Ring ORAM [126]	$2.5 \log N$	$O(\lambda)$	$\Omega(\log^2 N)$	XOR
Onion ORAM [51]	$O(1)$	$O(1)$	$\tilde{\Omega}(\log^5 N)$	Homomorphic enc.
This work	$4 \log_d N$	$O(1)$	$\Omega(d\lambda \log N)$	XOR
(with $d = \log N$)	$4 \log_{\log N} N$		$\Omega(\lambda \log^2 N)$	

Overview of our Construction

In tree-based ORAMs, the memory is organized in the form of a binary tree, where every tree node is a bucket. Buckets hold blocks, where each block is either dummy or real. The main invariant of tree-based ORAMs is that every block is assigned to the path from the root to a randomly chosen leaf node. Accessing a

[1]The title of that paper claims "constant bandwidth", which would have been immediately ruled out by our lower bound. On a closer look, the bandwidth blowup is actually $O(\log_d N)$. This calls for our lower bound to clear the confusion in this direction.

block in tree-based ORAMs has two phases. The first phase, called *retrieval*, fetches and possibly updates the data block requested by the client. For security, the block is also assigned a new random path. The second phase, called *eviction*, reshuffles some data blocks on the server subject to the tree-ORAM invariant. Many recent ORAM constructions [51, 126, 140, 143] are based on binary trees, in which the bandwidth blowup on retrieval and eviction are both $\Theta(\log N)$ due to the tree height.

Our construction uses a tree with larger fan-out $d = \omega(1)$, which decreases the tree height to $O(\log_d N) = O(\frac{\log N}{\log d})$. Based on a d-ary tree, we design a new eviction algorithm whose bandwidth blowup is $O(\log_d N)$. However, it increases the bandwidth blowup by more than a factor of d on retrieval in the standard model. We then use two-server private information retrieval (c.f. preliminaries Section 2.3) to reduce the retrieval bandwidth to $O(1)$ (assuming moderately large block size). Our basic eviction algorithm also requires $\Omega(d \log N)$ blocks of client storage. We again rely on two-server PIR to reduce the client storage to $O(1)$. Overall, we obtain a two-server ORAM with $O(1)$ client storage and $O(\log_d N)$, i.e., sub-logarithmic bandwidth blowup.

Although our bandwidth blowup decreases with the tree fan-out d, we cannot keep increasing d for free due to block metadata. We discuss the trade-off regarding d in Section 3.3.4.

```
1: function Access(addr, op, data)
2:     l ← PosMap[addr]
3:     data ← ReadBlock(l, addr)
4:     l′ ← UniformRandom(0, d^L − 1)
5:     PosMap[addr] ← l′
6:     if op = read then
7:         return data to client
8:     else
9:         data ← data′
10:    Write data to the root bucket
11:    evict()
```

Figure 3.1: **Tree-based ORAM data access algorithm.** Here, PosMap is a map from an address addr to a leaf l of the tree. ReadBlock(l, addr) retrieves a block of data with address addr from a path of buckets along leaf l.

3.1 Tree-based ORAM

We first describe a generic tree-based ORAM construction. This will aid the description of our final protocol. In a tree-based ORAM, server storage is organized as a binary tree [134]. As mentioned in the introduction, instead of a binary tree, in this work we use a d-ary tree. Hence this brief introduction presents the general case and considers d as an independent parameter.

Server storage. We consider d-ary tree with $L + 1$ levels, from level 0 to level L.

Thus, level i has d^i nodes. Recall that N is the total number of logical blocks stored by the client. Then L is roughly $\log_d N$. Each node in the tree is called a *bucket* and each bucket contains Z slots for logical blocks. A slot can also be empty — in this case, we say it contains a *dummy* block; otherwise, we say it contains a *real* block. Each block stores B bits of information. Dummy blocks and real blocks are both encrypted using randomized symmetric encryption.

Metadata. Aside from the B bits of block data, tree-based ORAMs also store some metadata for each block. The metadata stores the block identifier and whether the block is real or dummy. The client also maintains a position map PosMap that maps each real block to a random leaf in the tree.

In this chapter, we first assume that the client stores all the metadata locally. We then describe how this metadata can be offloaded to the server (Section 3.3.3) to achieve $O(1)$ client storage.

Invariant. Tree-based ORAM maintains the invariant that if a block is mapped to a leaf l of the tree, the block must be in some bucket on the path from the root to the leaf l. Since a leaf uniquely determines a path and vice versa, we use the two terms interchangeably.

Access. The pseudo-code for an access algorithm in a tree-based ORAM is described in Figure 3.1. To access a block with logical address addr, the client performs the following operations:

1. Look up the local PosMap to figure out the path l it is mapped to (line 2).

2. Download and decrypt every block on path p, discarding every block that does

not have address addr. Due to the invariant, the client is guaranteed to find block addr on path l. This is done by ReadBlock(l, addr) in Figure 3.1 line 3.

3. Remap block addr to a new random path l' (i.e., update PosMap), i.e. logically remove block addr from its old position (lines 4 and 5).

4. Re-encrypt block addr and append it to the root bucket (line 10, encryption is not shown in the figure).

5. Invoke an eviction procedure to percolate blocks towards leaves (line 11).

The first four steps correspond to the retrieval phase, and are similar for many tree-based ORAMs [51, 134, 140]. Tree-based ORAMs differ in their eviction procedures (which also affect the bucket size Z). Existing tree-based ORAM schemes when extended to use a d-ary tree do not achieve sub-logarithmic bandwidth blowup due to inefficient eviction. Hence, a main contribution of this chapter is to construct such an eviction scheme (Section 3.2).

3.2 Main Construction

Our construction follows the tree-based ORAM paradigm in the previous section. In this section, we present the changes in server storage and the retrieval and eviction strategies to obtain a sub-logarithmic bandwidth blowup. Figures 3.2 and 3.3 show the pseudocode of our construction. Figure 3.4 shows how servers store blocks and an example eviction for our construction.

Server storage. Our construction uses two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, both storing identical

1: Persistent variables cnt, G initialized to 0

2: cnt is the number of accesses performed so far since the previous eviction

3: G is the number of evictions performed so far, represented in base d

4: Let $\mathcal{P}(l)$ be the path from root to leaf l, and $\mathcal{P}(l, k)$ be the k-th bucket on $\mathcal{P}(l)$.

5: **function** Access(addr, op, data$'$)

6: $\quad l \leftarrow \mathsf{PosMap}[\mathsf{addr}]$

7: $\quad \mathsf{data} \leftarrow \mathsf{ReadBlock}(l, \mathsf{addr})$

8: $\quad$ **if** $\mathsf{op} = \mathsf{read}$ **then**

9: $\quad\quad$ return data to client

10: $\quad$ **else**

11: $\quad\quad \mathsf{data} \leftarrow \mathsf{data}'$

12: $\quad l' \leftarrow \mathsf{UniformRandom}(0, d^L - 1)$

13: $\quad \mathsf{PosMap}[a] \leftarrow l'$

14: $\quad$ Write data to the cnt-th slot of the root bucket

15: $\quad \mathsf{cnt} := \mathsf{cnt} + 1 \mod Z/2$

16: $\quad$ **if** $\mathsf{cnt} = 0$ **then**

17: $\quad\quad l_e \leftarrow \mathsf{reverse}(G)$

18: $\quad\quad \mathsf{EvictAlongPath}(l_e)$

19: $\quad\quad G \leftarrow G + 1 \mod d^L$

Figure 3.2: **Access and eviction algorithm for our oblivious RAM construction.**

1: **function** ReadBlock(l, addr)

2: $(\mathsf{id}_1, \mathsf{id}_2, \ldots, \mathsf{id}_{ZL}) \leftarrow$ Retrieve block identifiers on $\mathcal{P}(l)$

3: Suppose $\mathsf{id}_i = \mathsf{addr}$

4: **return** TwoServerPIR$(\mathcal{S}_1, \mathcal{S}_2, \mathcal{P}(l), i)$

5: **function** EvictAlongPath(l_e)

6: **for** $k \leftarrow 0$ to $L - 1$ **do**

7: Let s be the $(k+1)$-th digit of G *// For each bucket, $(k+1)$-th digit accesses slices in a round-robin manner.*

8: EvictToSlices(l_e, k, s)

9: *// Additional processing for the leaf bucket $\mathcal{P}(l_e, L)$ to make it empty*

10: Read all blocks in $\mathcal{P}(l_e, L)$ and its auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$

11: Move all real blocks from $\mathcal{P}(l_e, L)$ to $\mathcal{P}(l_e, \mathsf{aux})$

12: **function** EvictToSlices(l_e, k, s)

13: *// Evict from bucket $\mathcal{P}(l_e, k)$ to the s-th slice of each of its d children*

14: Download all blocks in $\mathcal{P}(l_e, k)$

15: **for** $t \leftarrow 1$ to d

16: Let S be the s-th slice of the t-th child of $\mathcal{P}(l_e, k)$

17: Let T be the set of real blocks in $\mathcal{P}(l_e, k)$ that can be evicted to S

18: Upload T to S and pad remaining slots in S with dummy blocks

Figure 3.3: **Access and eviction algorithm for our oblivious RAM construction. (..contd)**

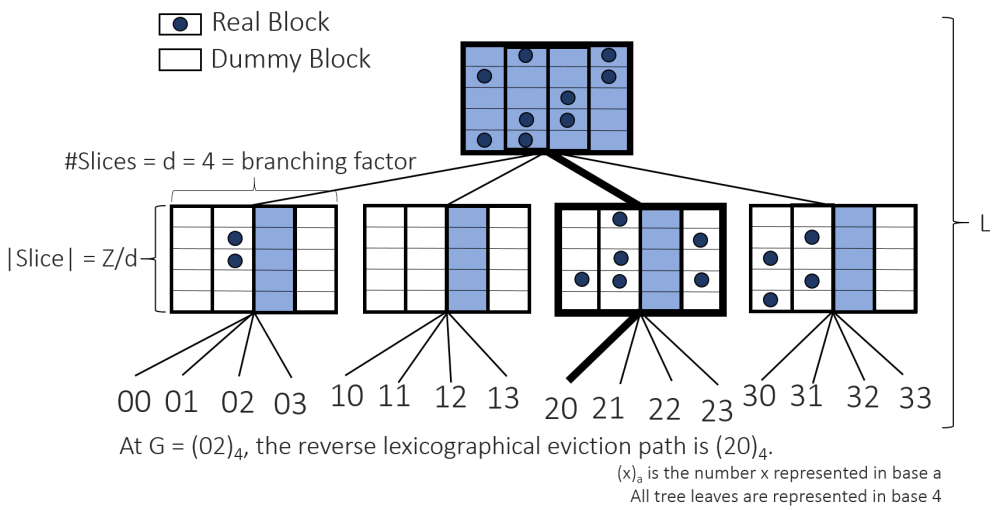


Figure 3.4: **Example eviction path for a three-level 4-ary tree at $G = 2$ i.e. $G = (02)_4$.** For evicting the root bucket into its children buckets, the client downloads blue colored root bucket and writes to the blue colored slices of its children. The figure shows load of the buckets just before eviction from the root bucket.

information (hence, Figure 3.4 shows only one tree). Our d-ary tree has $L+1$ levels, numbered from 0 (the root) to L (the leaves). Each node in the tree is called a *bucket*. Each bucket consists of Z slots that can each store one block. Slots from the non-root buckets are equally divided into d *slices*, each of size Z/d. Each leaf bucket has an auxiliary bucket aux that can store Z blocks.

Metadata. Our construction requires metadata similar to the description in Section 3.1, i.e., the position map PosMap and a block identifier for each slot. As mentioned, we assume the client stores all metadata locally for the cloud storage application, but can easily outsource them to the server without asymptotically increasing bandwidth blowup (Section 3.3.3).

Initialization. Initially, the ORAM tree at both servers contain all dummy blocks. The position map is initialized to contain independent and uniformly random numbers for each block. The client initializes each block using a logical write operation. If the client issues a logical read operation to a block that has never been initialized, the behavior of the ORAM is undefined.

Access. Each client request is represented as a tuple $(\mathsf{addr}, \mathsf{op}, \mathsf{data}')$ where addr is the address of the block, $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}\}$ and data' is the data to be written ($\mathsf{data}' = \perp$ if $\mathsf{op} = \mathsf{Read}$). The client maintains a counter cnt for the total number of accesses made so far. For each access $(\mathsf{addr}, \mathsf{op}, \mathsf{data}')$, the client does the following (refer Figure 3.2 and 3.3):

1. The client looks up position map $\mathsf{PosMap}[\mathsf{addr}]$ to obtain the leaf l associated with block addr (Figure 3.2, line 6).

2. Let $\mathcal{P}(l)$ represent the path from root to leaf l, and $\mathcal{P}(l, k)$ represent the k-th bucket on $\mathcal{P}(l)$. The client retrieves the block identifiers on the path $(\mathsf{id}_1, \mathsf{id}_2 \ldots, \mathsf{id}_{ZL})$ from its local storage. Due to the tree-based ORAM invariant, one of the identifiers on the path will be addr. Without loss of generality, assume $\mathsf{id}_i = \mathsf{addr}$ (Figure 3.3, lines 2 and 3).

3. The client invokes a two-server PIR protocol $\mathsf{TwoServerPIR}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{P}(l), i)$ to retrieve the block with address addr (Figure 3.3, line 4).

4. The client updates the data field of the block addr to data' if $\mathsf{op} = \mathsf{Write}$. It sets a new leaf l' for the block and updates PosMap. It updates the metadata

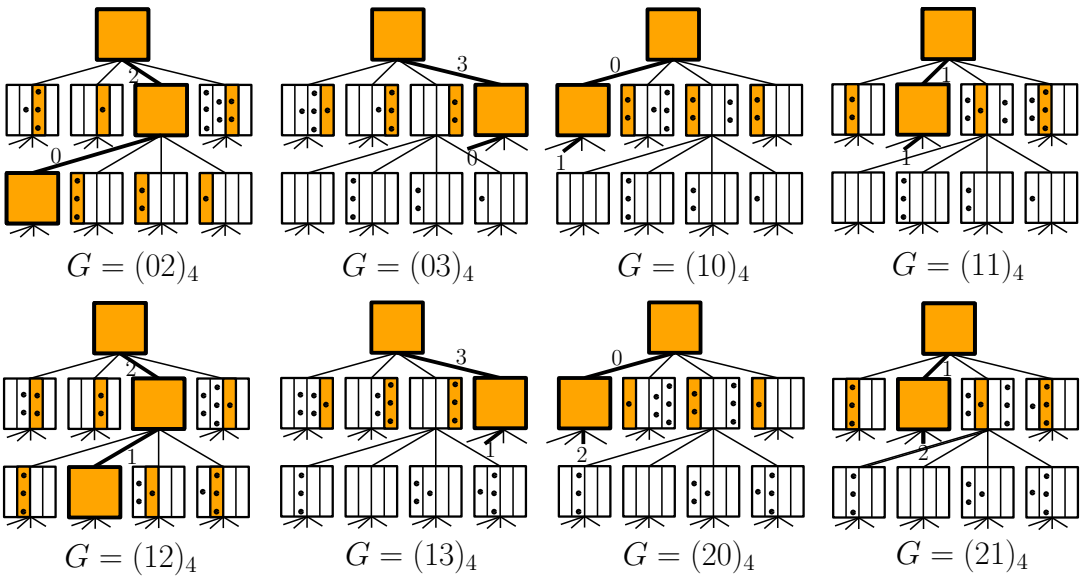


Figure 3.5: **Buckets and slices accessed for $2d$ consecutive evictions.** Here, $d = 4$ and $G = \#$ evictions $\mathsf{mod}\ d^L$. $(x)_a$ denotes the number x represented in base a. The dots in the slices represent real blocks at the end of the eviction operation. Note that for each bucket, slices are accessed (written into) in a round-robin manner. If an eviction path passes through a bucket at level i at t-th eviction then it passes through it again at $t + d^i$ evictions.

to remove the block from the tree. It appends the block addr to the cnt-th slot of the root bucket (Figure 3.2, lines 8-14).

5. The client increments cnt. If $\mathsf{cnt} = Z/2$, the client resets cnt and performs the eviction procedure described below (Figure 3.2, lines 15-19).

Eviction. The eviction procedure of our construction is a generalization of the eviction procedure of Onion ORAM [51]. It differs from Onion ORAM in the following two ways. First, we apply the eviction scheme on a reverse lexicographical ordering [61] over a d-ary tree instead of a binary tree. Second, when evicting from

each bucket along a path, we write to only one slice of each child bucket (instead of writing to the entire child buckets). This is essential for our construction to achieve sub-logarithmic bandwidth blowup.

As shown in Figure 3.3, we evict every $Z/2$ accesses along reverse lexicographical ordering of paths. Given that we have a d-ary tree instead of a binary tree, we represent the paths as numbers with base d. We use a counter G to maintain the next path l_e that should be evicted. Eviction is performed for each non-leaf bucket on path $\mathcal{P}(l_e)$. For the k-th bucket from the root, denoted $\mathcal{P}(l_e, k)$, the client first downloads the bucket $\mathcal{P}(l_e, k)$. It then uploads all real blocks to the s-th slice (which will be empty before this operation) of each of its children where s is the $(k+1)$-th digit of G. (We show in Section 3.3.2 that there will be sufficient room in these slices.) After this operation, the bucket $\mathcal{P}(l_e, k)$ will be empty. Due to the reverse lexicographical order of eviction paths, $\mathcal{P}(l_e, k)$ will be a child bucket for the next $d-1$ evictions involving it (refer Figure 3.5 for an example), during each of which the slice being written to will be empty. For the last level (level L), the client downloads all blocks in the leaf bucket $\mathcal{P}(l_e, L)$ and its auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$. It moves all real blocks to the auxiliary bucket $\mathcal{P}(l_e, \mathsf{aux})$ and uploads both buckets to the server.

Example. An example showing $2d$ consecutive evictions is in Figure 3.5 for $d = 4$. In the example, we start with eviction number $G = (02)_4$. Observe that the third child of the root bucket is emptied at $G = (02)_4$ as the reverse lexicographic eviction path $(20)_4$ passes through it. In the next $d-1$ evictions, one slice of the bucket is written to in a round-robin manner. Finally, at eviction number $G = (12)_4$, when

the path $(21)_4$ passes through it again, the last slice is written into after which the entire bucket is emptied again. Similarly, it can be easily seen that for each bucket at level i, a slice is written into every d^{i-1} evictions and the bucket is emptied every d^i evictions.

3.3 Analysis

3.3.1 Overflow Analsysis

We show that the buckets (and slices) in the tree overflow with negligible probability. In our construction, the root bucket and the auxiliary buckets are not partitioned into slices. Eviction is performed every $Z/2$ accesses, so the root bucket never overflows. Below, Lemma 1 analyzes auxiliary buckets while Lemma 2 analyzes slices in non-root non-auxiliary buckets.

Lemma 1. *If the size of auxiliary buckets Z_{aux} satisfies $N \leq d^L \cdot Z_{\mathsf{aux}}/2$, the probability that an auxiliary bucket overflows is bounded by $e^{-\frac{Z_{\mathsf{aux}}}{6}}$.*

Proof. For an auxiliary bucket b, define $Y(b)$ to be the number of real blocks in b. Each of the N blocks in the ORAM has a probability of d^{-L} to be mapped to b independently. Thus, $E[Y(b)] \leq N \cdot d^{-L} \leq Z_{\mathsf{aux}}/2$, and a simple Chernoff bound completes the proof. □ □

The following lemma generalizes Onion ORAM [51] Lemma 1 to the scenario of a d-ary tree.

Lemma 2. *The probability that a slice of a non-root and non-auxiliary bucket overflows after an eviction operation is bounded by* $e^{-\frac{Z}{6d}}$.

Proof. Consider a bucket b, and its i-th slice b_i. Define $Y(b)$ to be the number of real blocks in b, and $Y(b_i)$ to be the number of blocks in b_i after an eviction operation.

We will first assume that all slices have infinite capacity and show that $E[Y(b_i)] \leq Z/2d$, i.e., the expected number of blocks in a non-root slice after an eviction operation is no more than $Z/2d$ at any time. Then, we bound the overflow probability given a finite capacity.

For a non-root and non-auxiliary bucket b, we define variables $\overline{m}$ and $m_i, 1 \leq i \leq d$: the last EvictAlongPath operation where b is on the eviction path is the $\overline{m}$-th EvictAlongPath operation, and the EvictAlongPath operation where b is a sibling bucket with eviction happening to slice i is the m_i-th EvictAlongPath operation. Clearly, during eviction to one of the d slices, the bucket b is on the eviction path. Thus, one of m_i is equal to $\overline{m}$. We also time-stamp the blocks as follows. When a block is accessed and remapped, it gets a time stamp m^*, if the next EvictAlongPath would be the m^*-th EvictAlongPath operation.

Now consider b_i and $Y(b_i)$. There exist the following cases:

1. If $\overline{m} \geq m_i$, then $Y(b_i) = 0$, because the entire bucket b becomes empty when it is a parent bucket during the $\overline{m}$-th EvictAlongPath operation, and the next eviction that evicts blocks to slice b_i has not occurred.

2. If $\overline{m} < m_i$, we must have $m_{i-1} < m_i$. Otherwise, m_i is the smallest among $m_1, \ldots, m_d$ and it must be that $\overline{m} \geq m_i$. We consider blocks with what time

stamp range can end up in b_i.

- Blocks with time stamp $m^* \leq \overline{m}$ will not be in b_i as these blocks would have been evicted out of b in the $\overline{m}$-th EvictAlongPath operation.
- Blocks with time stamp $\overline{m} < m^* \leq m_{i-1}$ or $m^* > m_i$ will not be in b_i as these blocks are evicted to either slices $\leq i-1$ or slices $> i$ respectively.
- Blocks with time stamp $m_{i-1} < m^* \leq m_i$ can be evicted to b_i.

There are at most $(m_i - m_{i-1})Z/2$ blocks with time stamp $m_{i-1} < m^* \leq m_i$. Each of these blocks go to bucket b independently with probability d^{-j}, where j is the level of b. Due to the deterministic reverse lexicographic ordering of eviction paths, it is easy to see that $m_i - m_{i-1} = d^{j-1}$. Therefore, $E[Y(b_i)] \leq d^{j-1} \cdot Z/2 \cdot d^{-j} = Z/2d$.

In either case, we have $\mu = E[Y(b_i)] \leq Z/2d$. Now that we have independence and the expected number of blocks in a bucket, using a Chernoff bound with $\delta = 1$, a slice b_i overflows with probability

$$\Pr[Y(b_i) > (1+\delta)u] \leq e^{-\frac{\delta^2 \mu}{3}} = e^{-\frac{Z}{6d}}.$$

□

Combining the two lemmas, we can set $Z = \Omega(d\lambda)$ and $Z_{\mathsf{aux}} = \Omega(\lambda)$. The probability that any slice or any bucket overflows is $e^{-\Omega(\lambda)}$. Following prior work [51, 126, 140], it suffices to set $\lambda = \omega(\log N)$ for $N^{-\omega(1)}$ failure probability, i.e., negligible in N.

Server Storage The amount of server storage in our construction is

$$Z_{\mathsf{aux}} \cdot d^L + Z \cdot \Sigma_{i=0}^{L} d^i = \Theta(N).$$

3.3.2 Security Analysis

Lemma 3. *The above ORAM construction satisfies obliviousness and is statistically-secure as per the definition in Section 2.2.*

Proof. Similar to all tree based ORAMs, for each access, the client performs the retrieval phase on a random path. The use of PIR hides the location of the requested block on that random path. Moreover, the instance of the two-server PIR scheme we use does not use any computational assumptions. Eviction is performed on a publicly known reverse lexicographical ordering of paths. Along the eviction path, each bucket and a predetermined slice in each child buckets are downloaded/uploaded. Thus, all client operations observed by the servers are independent of the logical client access patterns. □

3.3.3 Reducing Client Storage

In the construction described so far, the client stores the $\Theta(N \log N)$-bit position map, $\Theta(N \log N)$-bit metadata for all block and uses $\Theta(d\lambda)$ blocks of temporary storage during the eviction operation. In this section, we optimize our scheme to reduce the client storage to $O(1)$ blocks.

A. Position map. The position map for the main ORAM has a $\Theta(\log N)$-bit entry for each of the N blocks, amounting to $\Theta(N \log N)$ bits of storage.

Position map can be stored recursively in smaller ORAMs as discussed by Shi et al. [134]. As discussed in [139], when the data block size is $\Omega(\log^2 N)$ (which is the case for our scheme), using a small block size for recursive position map ORAMs, the asymptotic cost of recursion would be insignificant compared to the main ORAM tree. Hence, recursion does not increase to the bandwidth blowup asymptotically.

B. Metadata for each block in the tree. For each block of the tree, we store whether the block is real or dummy. If it is real, the identifying address is stored. This amounts to another $\Theta(N \log N)$ bits of storage.

We can store the metadata of each block along with the block data on the server. However, this would require downloading metadata from the server during retrieval before performing each PIR operation. For $Z = O(d\lambda)$, $L < \log_d N$ and a size of $O(\log N)$ bits for storing the identifier and whether the block is dummy, the total amount of metadata downloaded for an access is $O(d\lambda \log N \log_d N)$. Thus, for a block size of $\Omega(d\lambda \log N \log_d N)$ bits, the asymptotic bandwidth for downloading this metadata is absorbed.

C. Temporary storage for an eviction operation. During an eviction operation, the client downloads a bucket and a slice from each of its d children. This is equivalent to downloading two buckets. Thus, for each step of the eviction operation the client needs to store $Z = O(d\lambda)$ blocks.

We now show how this client storage can be reduced to $O(1)$. At a high level, the client needs to perform the eviction from a bucket to its children buckets without downloading the entire buckets. If the client can only store one block, it needs to download one block at a time from the parent bucket and upload it to one of its

children buckets. And the client needs to do so obliviously. We achieve this by hiding which block from the parent bucket is downloaded, again using PIR, and letting the client upload to the children buckets in a deterministic order. The new EvictToSlices algorithm for evicting a parent bucket to its children slices is shown in Figure 3.6.

```
1: function EvictToSlices(l_e, k, s)
2:     // Evict from bucket P(l_e, k) to the s-th slice of each of its d children
3:     Download metadata for bucket P(l_e, k) from S_1
4:     for t ← 1 to d
5:         Let S be the s-th slice of the t-th child of P(l_e, k) and S_i be its i-th slot
       // S is empty
6:         for each S_i ∈ S
7:             if ∃j such that the j-th block in P(l_e, k) can be evicted to S then
8:                 block = TwoServerPIR(S_1, S_2, P(l_e, k), j)
9:                 Locally update the metadata for the j-th block in P(l_e, k) to be
       dummy
10:                Upload block along with its metadata to S_i on both servers
11:            else // no such j exists, do a dummy PIR and a dummy upload
12:                Run TwoServerPIR(S_1, S_2, P(l_e, k), 1) and discard its output
13:                Upload a dummy block with a dummy identifier to S_i on both servers
14:    Upload the updated metadata of P(l_e, k) to S_1
```

Figure 3.6: **Evicting to children slices using $O(1)$ blocks of client storage.**

To perform the eviction from a bucket $\mathcal{P}(l_e, k)$ to a slice S of its t-th child, the client first downloads the metadata corresponding to $\mathcal{P}(l_e, k)$ (line 3). The client uploads to each slot i in S (denoted S_i) sequentially, one slot at a time (line 6). Before this eviction, each slot S_i will be empty due to Lemma 2. There are two cases:

1. If there exists a real block in $\mathcal{P}(l_e, k)$ that can be evicted to S, the client downloads that block from $\mathcal{P}(l_e, k)$ using PIR (thus hiding its location in $\mathcal{P}(l_e, k)$), and uploads it (re-encrypted) to S_i (lines 7-10).

2. If no real block in $\mathcal{P}(l_e, k)$ can be evicted to S, the client performs a dummy PIR to download an arbitrary block from $\mathcal{P}(l_e, k)$, discards the PIR output, and uploads an encrypted dummy block to S_i (lines 11-13).

Thus, for each $S_i \in S$ in order, the client downloads a block from the parent bucket using PIR (without revealing its position or whether its a dummy PIR) and uploads a block to S_i. This eviction process requires $O(1)$ blocks of storage.

3.3.4 Bandwidth Analysis

Lemma 4. *Our ORAM construction requires a bandwidth blowup of* $O(\log_d N)$ *for block sizes* $B = \Omega(d\lambda \log N)$.

Proof. Let us first analyze the bandwidth blowup of our construction by ignoring the number of bits accessed for metadata. We only read two blocks of data for PIR; thus, the bandwidth blowup for retrieving a block using PIR is $O(1)$. On evictions, for each bucket on the path, the client downloads the parent bucket and uploads

to one slice from each of the d child buckets, which is equivalent to two buckets of bandwidth. Thus, an eviction costs $2ZL$ blocks of bandwidth and it is performed every $Z/2$ accesses, giving an amortized bandwidth blowup of $4L < 4\log_d N$. Thus, ignoring metadata, the bandwidth blowup of our scheme is $O(\log_d N)$.

Although our bandwidth blowup decreases with d, we cannot keep increasing d for free. The reason is that the client needs to download a $\Theta(\log N)$-bit metadata for all $d\lambda \log_d N$ blocks on a path, on each access and eviction. Recursion contributes another $O(\log^3 N)$ bits, but that is no greater than the blowup due to the metadata. So the raw bandwidth (in bits) per access is $O(B \log_d N + d\lambda \log_d N \log N)$. While we usually focus on the multiplicative blowup term, when d becomes too large, the additive term will dominate. Thus, due to the metadata in the PIR operation, the aforementioned bandwidth blowup only holds if the block size is $B = \Omega(d\lambda \log N)$.

□

As a consequence, the optimal d should be determined as a function of the block size B and the number of blocks N. For instance, for an application using moderately large block size $B = \Omega(\lambda \log^2 N)$, we can set $d = \Theta(\log N)$ and the bandwidth blowup is $O(\log N / \log\log N)$. If some application uses very large blocks such as $B = \Omega(\sqrt{N}\lambda \log N)$, then we can set $d = \Theta(\sqrt{N})$ and achieve a bandwidth blowup of $O(1)$.

Bandwidth vs. server computation in practice. Our scheme achieves a sublogarithmic bandwidth blowup of $O(\log_d N)$ but also incurs XOR computation on polylogarithmic number of blocks (specifically, $O(d\lambda \log_d N)$ blocks). When implemented

in a cloud-server scenario, the computation will require the CPUs to read these poly-logarithmic data blocks from disk to perform XOR computations. Thus, the gain in bandwidth will improve performance only when the server computation (and the time to read disks is small). In practice, this will be useful when the available bandwidth is very small compared to the time required for server computation.

3.4 Extending the Goldreich-Ostrovsky Lower Bound

Goldreich and Ostrovsky [67] gave an $\Omega(\log_c N)$ lower bound on the bandwidth blowup assuming perfect correctness, perfect security and assuming the client to be restricted to the following operations: reading from a memory location and writing to a memory location. Here, N represents the number of logical blocks stored on the server and c is the number of logical blocks stored by the client.

In this section, we first review (a slight variant of) the original Goldreich-Ostrovsky lower bound and its proof from [67]. We then extend the model to include PIR and PIR-write as possible client operations, and analyze how this changes the bound.

Following Goldreich-Ostrovsky and Boyle-Naor [29, 67] (both papers considered the client as a CPU) we will use client and CPU interchangeably, and refer to client storage as c registers.

3.4.1 Original Lower Bound

We present a slight variant of the CPU used in the original lower bound work [67]:

Definition 1. *A CPU is modeled as a probabilistic random-access-machine (*RAM*) which has c registers and is restricted to the following operations:*

- READ_i *reads a block from memory into a CPU register: For an* addr $\in [N]$*, set* $\mathsf{Reg}[i] \leftarrow$ val*, where* val *is the content of the memory at location* addr*.*
- WRITE_i *moves a block from a CPU register to memory: For some* $i \in [c]$ *and* addr $\in [N]$*, write the data in* $\mathsf{Reg}[i]$ *to location* addr *in the memory.*

Our RAM model has slightly different operations than the original analysis in [67] to simplify the exposition. This will cause only constant factor differences in the bound. Given this model, the original lower bound can be stated as:

Theorem 2. *Every statistically oblivious simulation of a probabilistic* RAM *(Definition 1), which can access N blocks of data and hold c blocks in the CPU, on input y makes at least $max\{|y|, \Omega(t \cdot \log_c N)\}$ accesses to simulate $t = \mathsf{poly}(N)$ steps.*

Proof. As described by Goldreich and Ostrovsky [67], the task of simulating an ORAM can be modeled as a game in between three players

1. The CPU, which can hold at most c blocks and has access to a random function f
2. the program request sequence of length t

3. the observer

The CPU knows the program request sequence ahead of time. Suppose the request access sequence is represented as $r_1, \ldots, r_t$. The game takes place on a memory consisting of N cells, each capable of storing a single block.

The game proceeds for t rounds. In each round i, the CPU loads r_i in one of its registers in such a way that the observer does not learn r_i. The CPU does so by making a sequence of operations as described in Definition 1. At any time during the game, the CPU can store at most c blocks in memory. Round i ends when the CPU holds block r_i.

For a t sized request sequence, the CPU's response can be represented by two q sized sequences. The first sequence consists of an externally visible access pattern. Each entry in this sequence is a single address (each of which is sufficient to implement either READ_i or WRITE_i). The second sequence consists of hidden operations performed by the CPU. Let us consider the possible number of *hidden actions* that can be taken by the CPU, *for a fixed visible access pattern.* For each visible read address, the block that is read can be stored in one of the c registers in the CPU. Likewise, for each visible write address, the block written back could have been removed from any one of the c registers in the CPU. Thus, hidden from the observer, the system can evolve in c ways on a READ and c ways on a WRITE.

Given that the CPU stores c blocks, a q length sequence can satisfy at most c^q program access sequences. Therefore, each visible access sequence can satisfy at most $(2c)^q c^q$ program request sequences. For perfect security, the visible access

sequence should be able to satisfy all N^t possible request sequences, i.e.,

$$N^t \leq (2c)^q c^q$$

or $q \geq \frac{t \log N}{\log c + \log(2c)} = \Omega\left(t \log_c N\right)$. □ □

The above is a bound on the number of operations. Since each operation incurs at least 1 block of bandwidth, we also obtain an amortized bandwidth blowup lower bound of $\Omega(\log_c N)$.[2]

3.4.2 Augmented Lower Bound (after adding PIR)

We now extend the above result to allow the CPU to perform PIR and PIR-write.

Definition 2. *A PIR-augmented CPU is modeled as a probabilistic random-access-machine* PIR-RAM *which has c registers and is restricted to the following operations:*

- READ_i *as described in Definition 1.*
- WRITE_i *as described in Definition 1.*
- $\mathsf{PIR\text{-}READ}_i$ *reads a block from memory into a CPU register using PIR: For a set of at most D addresses, set* $\mathsf{Reg}[i] \leftarrow \mathsf{val}$*, where* val *can be the content of the memory at any of the locations in the set.*

[2] If we assume that the memory is initially permuted by the CPU unknown to the server, then the total number of program request sequences is at most $M^M(2c)^q c^q$ where $M = \mathsf{poly}(N)$ is the physical memory size. Hence, we have $q = \Omega((t - M)\log_c N)$.

- $\mathsf{PIR\text{-}WRITE}_i$ *moves a block from a CPU register into memory privately using a PIR-WRITE operation: For a set of at most* D *addresses, write the data in* $\mathsf{Reg}[i]$ *to a location among one of the* D *addresses.*

Theorem 3. *Every statistically oblivious simulation of a probabilistic* PIR-RAM *(Definition 2), which can access* N *blocks of data and hold* c *blocks in the CPU and perform PIR on a maximum of* D *blocks, on input* y *makes at least* $max\{|y|, \Omega(t \cdot \log_{cD} N)\}$ *accesses to simulate* $t = \mathsf{poly}(N)$ *steps.*

Proof. The proof follows the same framework as the original lower bound. The number of operations in the visible and hidden sequences due to READ_i or WRITE_i operations is unchanged. Now, the visible sequence additionally reveals the set of D addresses accessed on a PIR request for $\mathsf{PIR\text{-}READ}_i$/$\mathsf{PIR\text{-}WRITE}_i$. In each of these operations, the client can select one out of D possible memory blocks to read/write in the visible memory. Furthermore, for each of the above D outcomes, the client can add the read block to (or remove the written block from) any one of the c local registers. Thus, the system can evolve in cD possible ways for each of the PIR-READ and PIR-WRITE operations.

Extending the original argument, each visible access sequence can satisfy $(2c + 2cD)^q c^q$ program request sequences. For perfect security, the visible access sequence should be able to satisfy all N^t possible request sequences, i.e.,

$$N^t \leq (2c + 2cD)^q c^q$$

or $q \geq \frac{t \log N}{\log c + \log(2c+2cD)} = \Omega\left(\frac{t \log N}{\log(cD)}\right)$. □

Again, the bound is on the number of operations. Since each of the four operations incurs at least 1 block of bandwidth, a bound on the number of operations translates to a bound on amortized bandwidth blowup.

3.4.3 Discussion

Accounting for failure probability. The above lower bound assumes perfect security, i.e., each visible physical access sequence should be able to satisfy all possible program request sequences. However, using an argument similar to Wang et al. [143], the same lower bound can be extended to work for up to $O(1)$ failure probability (and hence, negligible failure probability).

PIR as a black box. Our lower bound is independent of the implementation details of the PIR and PIR-write operations. The bound is applicable to any statistically secure PIR construction that meets the interface in Definition 2, regardless of the number of servers it uses. We also note that although the lower bound considers PIR-WRITE as a possible operation, our construction does not use this primitive.

Our construction and the lower bound. Our construction matches this lower bound for certain parameter ranges. We use $c = O(1)$ registers and perform a PIR operation on $D = O(d \cdot \mathsf{poly}(\log N))$ blocks. Thus, our lower bound is asymptotically tight for $d = \Omega(\log N)$ when the data block size $B = \Omega(d\lambda \log N)$.

C-ORAM, CHf-ORAM and the lower bound. C-ORAM and CHf-ORAM introduced three new operations on top of the standard ORAM model: download a block from a path of poly-logarithmic blocks using PIR-READ, upload a block to one

hidden location in a bucket using PIR-WRITE, and an oblivious merge operation. In an oblivious merge operation, the server applies plaintext permutations (chosen by the client) to buckets before merging them. This operation creates only one possible outcome to the system state, since no action is hidden from the server. Thus oblivious merge does not affect the lower bound in Section 3.4.

CHf-ORAM achieves statistical security with negligible failure probability and is thus subject to the lower bound in Theorem 3. The number of operations required for t logical accesses is $\Omega(\frac{t \log N}{\log(cD)})$ where $c = O(1)$ and $D = \mathsf{polylog}(N)$. Thus, its bandwidth blowup is lower bounded by $\Omega(\frac{\log N}{\log \log N})$. Instead, CHf-ORAM claims to have achieved $O(1)$ bandwidth, implying a flaw in its construction.

C-ORAM achieves computational security due to the use of single-server PIR-READ/ PIR-WRITE, and thus does not directly violate the lower bound. However, unless carefully shown otherwise, it is extremely unlikely that any security flaw of CHf-ORAM can be fixed by merely replacing information theoretically secure PIR with computationally secure PIR.

Circumventing the lower bound. The lower bound on bandwidth only applies to black-box usage of PIR. Onion ORAM [51] circumvents the lower bound and achieves $O(1)$ bandwidth blowup. The reason is that the homomorphic select operation in Onion ORAM (a non-black-box usage of PIR) does not consume one unit of bandwidth. Therefore, while the number of operations in Onion ORAM is still subject to the bound, the bound does not translate to a bound on bandwidth blowup. It is also possible to circumvent the lower bound by adding other operations (e.g., FHE [12]).

3.5 Related Work

Before ending this chapter, we mention works that are closely related to the techniques used in this chapter. The idea of using a d-ary tree was first used by Kushilevitz et al. [95] who achieveed $O(\log^2 N/\log\log N)$ bandwidth blowup using $\Theta(\log N)$ buffers for every large level. Gentry et al. [61] uses a $\Theta(\log N)$-ary tree and a push-to-leaf procedure along a deterministic path to achieve $O(\log^2 N/\log\log N)$ blowup. A concurrent work [154] uses a $\Theta(\log N)$-ary tree, which we compare to in detail later. In all cases, the idea is to balance the (sometimes implicit) bandwidth mismatch between the retrieval phase and the eviction phase.

Many works deviated from the traditional ORAM model defined by Goldreich and Ostrovsky by introducing multiple non-colluding servers and/or server-side computation. Some of these papers refer to their work as oblivious outsourced storage, but we still refer to them as ORAMs. We review these works below.

ORAMs using multiple non-colluding servers. Constructions in this category so far have not been able to surpass the $\Omega(\log N)$ bandwidth barrier (except CHf-ORAM [114] which we discuss later in this section) [106, 122, 136]. Lu and Ostrovsky [106] achieved a bandwidth blowup of $O(\log N)$. In their scheme, each non-colluding server performs permutations that are hidden to the other server due to which the Goldreich-Ostrovsky lower bound does not apply. Stefanov and Shi [136] implemented a practical system using two servers and $O(\sqrt{N})$ client storage. Their client storage can be reduced to $O(1)$ using the standard recursion technique [134]. Their construction required $O(1)$ client-to-server bandwidth blowup and $O(\log N)$

server-to-server bandwidth blowup.

ORAMs with server computation. There exist many ORAM schemes that allow the server to do computation on data blocks [12, 48, 51, 62, 110, 115, 126, 137, 138, 146, 147, 153]. Most of these works still require $\Omega(\log N)$ bandwidth blowup, except the following ones. Apon et al. [12] use fully homomorphic encryption to achieve an $O(1)$ bandwidth blowup. However, the large overhead of FHE makes the scheme impractical. Onion ORAM [51] improves upon Apon et al. to achieve an $O(1)$ bandwidth blowup by using only additively homomorphic encryption or somewhat homomorphic encryption. The amount of server computation is significantly reduced (compared to FHE) but is still quite large. In addition, the $O(1)$ bandwidth blowup of Onion ORAM can only be achieved for very large block sizes of $\Omega(\log^5 N)$. Both these schemes circumvent the Goldreich-Ostrovsky lower bound by using homomorphic operations on the server side that require little client intervention.

Independent and concurrent work. MSKT-ORAM [154] is an independent and concurrent work that achieves comparable bandwidth blowup using similar techniques, i.e., a d-ary tree and two-server PIR applied to a poly-logarithmic number of blocks. Our construction has several advantages stemming from the following major differences: While we extended the most recent tree-based ORAM, Onion ORAM [51], to a d-ary tree, MSKT-ORAM builds on top of the very first tree-based ORAM by Shi et al. [134] and extends it to a d-ary tree. Thus, MSKT-ORAM does not take advantage of the new techniques invented afterwards, such as small block recursion [139], reverse lexicographical order [61], higher bucket load [126],

reduced eviction frequency [126], and an empty bucket invariant [51]. As a result, MSKT-ORAM requires a block size as large as $\Omega(N^{\epsilon})$ for some constant ϵ, while we only require blocks of size $\mathsf{polylog}(N)$ bits; MSKT-ORAM has a $\omega(\log N)$ server storage blowup, while our construction has a constant size server storage blowup (Section 3.3); MSKT-ORAM needs a PIR, a physical read and a physical write operation to evict each block, while we can eliminate the need for the physical read due to the empty bucket/slice invariant (cf. Lemma 2 and Section 3.3.3); MSKT-ORAM also spends at least 2× more bandwidth for both blocks and metadata during eviction, since Shi et al. [134] requires two evictions after every access.

Oblivious RAM lower bound. As mentioned earlier, Goldreich and Ostrovsky presented a lower bound of $\Omega(\log_c N)$ where c is the amount of client storage in blocks. Their lower bound modeled the server as a simple storage device capable of reading and writing blocks. Boyle and Naor revisit the ORAM lower bound to relate it to the size of circuits for sorting [29]. In our work, we extend the lower bound suggested by Goldreich and Ostrovsky to encompass private information retrieval (PIR) as a possible operation performed by the client and obtain a lower bound of $\Omega(\log_{cD}(N))$ in Section 3.4. Here, c is the number of blocks stored by the client and D is the number of blocks that a PIR is performed on. C-ORAM [115] and CHf-ORAM [114] violate the lower bound and must have security flaws. Boyle and Naor showed that an ORAM lower bound is difficult to obtain in a general model, i.e., if the client is not restricted to a small set of operations.

Private information retrieval. A Private information retrieval (PIR) protocol allows a user to retrieve some data block from a server without revealing the block

that was retrieved. It was first introduced by Chor et al. [41]. In our work, we use a simple two server $O(N)$ scheme from [41] to reduce the bandwidth cost of accessing a block.

3.6 Conclusion, Subsequent Work, and Open Problems

In this work, we design an Oblivious RAM with sub-logarithmic bandwidth blowup where the servers only perform XOR operations. We achieve this by using a novel eviction scheme over a d-ary tree to obtain a blowup of $O(\log_d N)$ and using two-server PIR to reduce the cost to retrieve a block. We show a lower bound of $\Omega(\log_{cD} N)$ for bandwidth blowup for a client storing c blocks of data and performing a PIR on D blocks of data at a time. Our construction matches our lower bound under certain parameter ranges. C-ORAM [115] and CHf-ORAM [114] violate the lower bound and thus have security flaws.

Subsequent to our work, Kushilevitz et al. [97] have shown a construction that achieves a sub-logarithmic bandwidth blowup for a smaller block size of $\Omega(d \log N)$ bits (instead of $\Omega(d\lambda \log N)$ bits). However, their scheme is secure only against a computationally-bounded adversary.

It is still an open question whether a sub-logarithmic bandwidth blowup can be obtained in the original model defined by Goldreich and Ostrovsky. Also, all known ORAM schemes that achieve $O(\log N)$ bandwidth blowup require a block size of $\Omega(\log^2 N)$. Whether this bound (or a sub-logarithmic bound) can be obtained for smaller block sizes remains open.

Chapter 4: Perfectly Secure Oblivious RAM

In this chapter, we present a perfectly-secure OPRAM and a perfectly-secure ORAM scheme. As mentioned in Chapter 2, we consider *ORAMs to be a special case of OPRAMs*, i.e., when both the original PRAM and the OPRAM have only one CPU. Thus, our final scheme description only describes an OPRAM.

The original ORAM schemes, proposed by Goldreich and Ostrovsky [65, 67], achieved poly-logarithmic overheads but required the usage of pseudo-random functions (PRFs); thus they defend only against computationally bounded adversaries. Various subsequent works [11, 38, 43, 47, 134, 140, 143], starting from Ajtai [11] and Damgård et al. [47] investigated information-theoretically secure ORAM/OPRAM schemes, i.e., schemes that do not rely on computational assumptions and defend against even unbounded adversaries. As earlier works point out [11, 47], the existence of efficient ORAM schemes without computational assumptions is not only theoretically intriguing, but also has various applications in cryptography. For example, information-theoretically secure ORAM schemes can be applied to the construction of efficient RAM-model, information-theoretically secure multi-party computation (MPC) protocols [17]. Among known information-theoretically secure ORAM/OPRAM schemes [11, 28, 38, 39, 43, 47, 134, 140, 143], almost all of them

achieve only *statistical* security [11,28,38,39,43,134,140,143], i.e., there is still some non-zero failure probability — either correctness or security failure — but the failure probability can be made negligibly small in N where N is the RAM/PRAM's memory size. To the best of our knowledge, the only known *perfectly* secure ORAM construction is the elegant work by Damgård et al. [47] — they achieve 0 failure probability against computationally unbounded adversaries. Although recent works have constructed statistically secure OPRAMs [28,38,39], there is no known (non-trivial) *perfectly* secure OPRAM scheme to date.

Motivation for perfect security. Perfectly secure ORAMs/OPRAMs are theoretically intriguing for various reasons:

1. First, to achieve $2^{-\kappa}$ failure probability (either in security or in correctness), the best known statistically secure OPRAM scheme [35,38] incurs a $O(\kappa \log N)$ total work blowup and $O(\log \kappa \log N)$ depth blowup where N is the PRAM's memory size. Although for negligibly small in N failure probability the blowups are only poly-logarithmic in N, they can be as large as N^c for some constant $c < 1$ if one desires (sub-)exponentially small failure probability in N.

2. Second, perfectly secure ORAM schemes have been used as a buildling block in recent results in searchable encryption schemes [50]. Typically these algorithmic constructions rely on divide-and-conquer to break down a problem into smaller sizes and then apply ORAM to a small instance — since the instance size N is small (e.g., logarithmic in the security parameter), negligible in N failure probability is not sufficient and thus these works demand

perfectly secure ORAMs/OPRAMs and existing statistically secure schemes result in asymptotically poorer performance.

3. Third, understanding the boundary of perfect and statistical security has been an important theoretical question in cryptography. For example, a long-standing open problem in cryptography is to separate the classes of languages that admit perfect ZK and statistical ZK proofs. For ORAMs/OPRAMs too, it remains open whether there are any separations between statistical and perfect security (and we believe that this is an exciting future direction).

Our Results and Contributions

In this chapter, we prove the following result which significantly advances our theoretical understanding of *perfectly* secure ORAMs and OPRAMs in multiple respects. We present the informal theorem statement below and then discuss its theoretical significance.

Theorem 4 (Informal statement of main theorem)**.** *Any PRAM with m CPUs that consumes N memory blocks each of which is at least $\log N$-bits long*[1] *can be simulated by a perfectly oblivious PRAM, incurring $O(\log^3 N)$ total work blowup, $O(\log N(\log m + \log\log N))$ depth blowup, and $O(1)$ space blowup.*

The above theorem improves the theoretical state of the art on perfectly secure ORAMs/OPRAMs in multiple dimensions:

[1] All existing ORAM and OPRAM works [65, 67, 70, 96, 134] make this assumption.

1. First, our work gives rise to the first perfectly secure (non-trivial) OPRAM construction. No such construction was known before and it is not clear how to directly parallelize the perfectly secure ORAM scheme by Damgård et al. [47].

2. Second, even for the sequential special case, we improve Damgård et al. [47] asymptotically by reducing a $\log N$ factor in the ORAM's space consumption.

3. Finally, when (sub-)exponentially small (in N) failure probabilities are required, our perfectly secure OPRAM scheme asymptotically outperforms all known statistically secure constructions in terms of total work blowup! For example, suppose that we require $2^{-\kappa}$ failure probability and $N = \mathsf{poly}(\kappa)$ — then all known statistically secure OPRAM constructions [28, 38, 39] would incur at least N^c total work blowup and $\Omega(\log^2 N)$ depth blowup and thus our new perfectly secure OPRAM construction is asymptotically better for this scenario.

The above Theorem 4 applies to general block sizes. We additionally show that for sufficiently large block sizes, there exists a perfectly secure OPRAM construction with $O(\log^2 N)$ total work blowup and $O(\log N(\log m + \log\log N))$ depth blowup where m denotes the number of CPUs of the original PRAM. Finally, we point out that this work focuses mostly on the theoretical understanding of perfect security in ORAMs/OPRAMs, and we leave it as a future research direction to investigate their practical performance (see also Section 4.5).

4.1 Technical Roadmap

In this section, we present an informal roadmap of our technical approach to aid understanding.

4.1.1 Simplified Perfectly Secure ORAM with Asymptotically Smaller Space

First, we propose a new perfectly secure ORAM scheme that is conceptually simpler than that of Damgård et al. [47] and asymptotically gains a logarithmic factor in space. Our construction is inspired by the hierarchical ORAM paradigm originally proposed by Goldreich and Ostrovsky [65, 67] — however, most existing hierarchical ORAMs achieve only computational security since they rely on a pseudorandom function (PRF) for looking up hash tables in the hierarchical data structure. Thus our focus is how to get rid of this PRF and achieve perfect security.

Background: hierarchical ORAM. The recent work by Chan et al. [36] gave a clean and modular exposition of the hierarchical paradigm. A hierarchical ORAM consists of $O(\log N)$ levels that are geometrically increasing in size. Specifically, level i is capable of storing 2^i memory blocks. One could think of this hierarchical data structure as a hierarchy of stashes where smaller levels act as stashes for larger levels. In existing schemes with computational security, each level is an *oblivious hash-table* [36]. To access a block at logical address addr, the CPU sequentially looks up every level of the hierarchy (from small to large) for the logical address

addr. The physical location of a logical address addr within the oblivious hash-table is determined using a PRF whose secret key is known only to the CPU but not to the adversary. Once the block has already been found in some level, for all subsequent levels the CPU would just look for a dummy element, denoted by $\perp$. When a requested block has been found, it is marked as deleted in the corresponding level where it is found. Every 2^i memory requests, we perform a rebuild operation and merge all levels smaller than i (including the block just fetched and possibly updated if this is a write request) into level i — at this moment, the oblivious hash-table in level i is rebuilt, where every block's location in the hash table is determined using a PRF.

As Chan et al. [36] point out, the hierarchical ORAM paradigm effectively reduces the problem of constructing ORAM to constructing an oblivious hash-table supporting two operations: 1) **rebuild** takes in a set of blocks each tagged with its logical address, and constructs a hash-table data structure that facilitates lookups later; and 2) **lookup** takes a request that is either a logical address addr or dummy (denoted $\perp$), and returns the corresponding block requested. Obliviousness (defined w.r.t. the joint access patterns of the rebuild and lookup phases) is guaranteed as long as during the life-time of the oblivious hash-table, the sequence of lookup requests never ask for the same real element twice — and this invariant is guaranteed by the specific way the hierarchical ORAM framework uses the oblivious hash-table as a building block (more specifically, the fact that once a block is found, it is moved to a smaller level and a dummy block is requested from all subsequent levels).

Removing the PRF. As mentioned, an oblivious hash-table relies on a PRF to

determine each block's location within a hash-table instance; and both the rebuilding phase and the lookup phase use the same PRF for placing and fetching blocks respectively. Since we wish to achieve perfect security, we would like to remove the PRF. One simple idea is to randomly permute all blocks within a level — this way, each lookup of a real block would visit a random location and we could hope to retain security as long as every real block is requested *at most once* for every level (in between rebuilds)[2]. Using techniques from earlier works [35, 38], it is possible to obliviously perform such a random permutation without disclosing the permutation; however, the difficulty arises when one wishes to perform a look up — if blocks are randomly permuted within a level during rebuild, lookup must know where each block resides to proceed successfully. Thus if the CPU could hold a position map for free to remember where each block is in the hierarchical data structure, the problem would have been resolved: during every lookup, the CPU could first look up the physical location of the logical address requested, and then proceed accordingly.

Actually storing such a position map, however, would consume too much CPU space. To avoid storing this position map, we are inspired by the recursion technique that is commonly adopted by tree-based ORAM schemes [134] — however, as we point out soon, making the recursion idea work for the hierarchical ORAM paradigm is more difficult. The high-level idea is to recursively store the position map in a smaller ORAM rather than storing it on the CPU side; we could then recurse and store the position map of the position map in an even smaller ORAM, and so on

[2]As we point out later, randomly permuting real blocks is in fact not sufficient; we also need to allow dummy lookups by introducing an oblivious dummy linked list.

— until the ORAM's size becomes $O(1)$ at which point we would have the CPU store the entire ORAM. Henceforth, we use the notation ORAM_D to denote the ORAM that stores the actual data blocks where $D = O(\log N)$; and we use ORAM_d to denote the ORAM at depth d of this recursion where $d \in [0..D-1]$. Thus, the larger d is, the larger the ORAM.

Although this recursion idea was very simple in the tree-based ORAM paradigm, it is not immediately clear how to make the same recursion idea work in the hierarchical ORAM paradigm. One trickiness arises since in a hierarchical ORAM, every 2^i requests, the ORAM would reshuffle and merge all levels smaller than i into level i — this is called a rebuild of level i. When a level-i rebuild happens, the position labels in the position-map ORAM must be updated as well to reflect the blocks' new locations. In a similar fashion, the position labels in all of $\mathsf{ORAM}_0, \mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_{D-1}$ must be updated. We make the following crucial observation that will enable a *coordinated rebuild* technique which we will shortly explain:

> *(Invariant necessary for coordinated rebuild:)* If a data block resides at level i of ORAM_D, then its position labels in all recursion depths must reside in level i or smaller[3].

This invariant enables a *coordinated rebuild* technique: when the data ORAM (i.e., ORAM_D) merges all levels smaller than i into level i, all smaller recursion

[3] A similar observation was adopted by Goodrich et al. [71] in their statistically secure ORAM construction.

depths would do the same (unless the recursion depth is too small and does not have level i, in which case the entire ORAM would be rebuilt). During this coordinated rebuild, ORAM_D would first perform its rebuild, and propagate the position labels of all blocks involved in the rebuild to recursion depth $D-1$; then ORAM_{D-1} would perform its rebuild based on the position labels learned from ORAM_D, and propagate the new position labels involved to recursion depth $D-2$, and so on. As we shall discuss in the technical sections, rebuilding a level (in any recursion depth) can be accomplished through the help of $O(1)$ oblivious sorts and an oblivious random permutation.

Handling dummy blocks with oblivious linked lists. The above idea almost works, but not quite so. There is an additional technical subtlety regarding how to handle and use dummy blocks. Recall that during a memory access, if a block requested actually resides in a hierarchical level, we would read the memory location that contains the block (and this memory location could be retrieved through a special recursive position map technique). If a block does not reside in a level (or has been found in a smaller level), we still need to read a dummy location within the level to hide the fact that the block does not reside within the current level.

Recall that the i-th level must support up to 2^i lookups before the level is rebuilt. Thus, one idea is to introduce 2^i dummy blocks, and obliviously and randomly permute all blocks, real and dummy alike, during the rebuild. All dummy blocks may be indexed by a dummy counter, and every time one needs to look up a dummy block in a level, we will visit a new dummy block. In this way, we can retain obliviousness by making sure that every real block and every dummy block

is visited at most once before the level is rebuilt again.

To make this idea fully work, there must be a mechanism for finding out where the next dummy block is every time a dummy lookup must be performed. One naïve idea would be to use the same recursion technique to store position maps for dummy blocks too — however, since each memory request might involve reading $O(\log N)$ dummy blocks, one per level, doing so would incur extra blowup in runtime and space. Instead, we use an *oblivious dummy linked list* to resolve this problem — this oblivious dummy linked list is inspired by technical ideas in the Damgård et al. construction [47]. In essence, each dummy block stores the pointer to the next dummy block, and the head pointer of the linked list is stored at a designated memory location and updated upon each read of the linked list. In the subsequent technical sections, we will describe how to rely on oblivious sorting to rebuild such an oblivious dummy linked list to support dummy lookups.

Putting it altogether. Putting all the above ideas together, the formal presentation of our perfectly secure ORAM scheme adopts a modular approach[4]. First, we define and construct an abstraction called an "oblivious one-time memory". An oblivious one-time memory allows one to obliviously create a data structure given a list of input blocks. Once created, one can look up real or dummy blocks in the data structure, and to look up a real block one must provide a correct position label indicating where the block resides (imagine for now that the position label comes from an "oracle" but in the full ORAM scheme the position label comes from the

[4]In fact, later in this chapter, we omit the sequential version and directly present the parallel version of all algorithms.

recursion). An oblivious one-time memory retains obliviousness as long as every real block is looked up *at most once* and moreover, dummy blocks are looked up at most n times where n is a predetermined parameter (that the scheme is parametrized with).

Once we have this "oblivious one-time memory" abstraction, we show how to use it to construct an intermediate abstraction referred to as a "position-based ORAM". A position-based ORAM contains a hierarchy of oblivious one-time memory instances, of geometrically growing sizes. A position-based ORAM is almost a fully functional ORAM except that we assume that upon every memory request, an "oracle" will somehow provide a correct position label indicating where the requested block resides in the hierarchy.

Finally, we go from such a "position-based ORAM" to a fully functional ORAM using the special recursive position-map technique as explained.

At this point, we have constructed a perfectly secure ORAM scheme with $O(\log^3 N)$ simulation overhead. Specifically, one $\log N$ factor arises from the $\log N$ depths of recursion, the remaining $\log^2 N$ factor arises from the cost of the ORAM at each recursion depth. Intuitively, our perfectly secure ORAM is a logarithmic factor more expensive than existing computationally-secure counterparts in the hierarchical framework [36, 70, 96] since the computationally-secure schemes [36, 70, 96] avoid the recursion by adopting a PRF to compute the pseudorandom position labels of blocks.

Making our ORAM scheme parallel. Our next goal is to make our ORAM scheme parallel. Instead of compiling a sequential RAM program to a sequential

ORAM, we are now interested in compiling a PRAM program to an OPRAM. Suppose that the original program is a PRAM that completes in T parallel steps consuming m CPUs. First, using standard techniques, it would not be too difficult to parallelize our earlier ORAM scheme and construct an OPRAM that completes in $T \cdot O(\log^3 N)$ parallel steps consuming also exactly m CPUs. We stress that the simplicity of our sequential ORAM construction makes it easy to parallelize — in comparison, we are not aware how to parallelize Damgård et al. [47]'s construction. The main technique needed for this parallelization is *oblivious routing*: when the m CPUs at recursion depth d have fetched the position labels for the next recursion depth, the m CPUs at depth d must now obliviously route the position labels to the correct fetch CPU at the next recursion depth. As shown in earlier works [28,35,38], such oblivious routing can be accomplished with m CPUs in $O(\log m)$ parallel steps.

4.1.2 Building Blocks

We now introduce several useful oblivious algorithms building blocks. With the exception of oblivious random permutation, we assume that all remaining building blocks are deterministic: for a deterministic algorithm, obliviousness means that the algorithm's memory access pattern is independent of its input.

Oblivious sort. Ajtai, Komlós, and Szemerédi [10] show how to construct a circuit with $n \log n$ comparators that can correctly sort any input sequence containing n comparable elements. This immediately gives rise to a parallel oblivious sorting algorithm with $O(n \log n)$ total work and $O(\log n)$ depth.

Oblivious routing. Oblivious routing solves the following problem. Suppose n source CPUs each holds a data block with a distinct key (or a dummy block). Further, n destination CPUs each holds a key and requests a data block identified by its key — multiple destination CPUs can possibly request the same key. An oblivious routing algorithm routes the requested data block to the destination CPU in an oblivious manner. We may assume that the destination CPUs are represented by an ordered array X. Initially the payload of each entry of X is left empty. After the routing, each entry of X receives a data block (the received data block is dummy if no source CPUs hold the same key as requested). The ordering of elements in X is preserved between the input and output.

Boyle et al. [28] showed that through a combination of oblivious sorts and oblivious aggregation, oblivious routing can be achieved in $O(\log n)$ parallel runtime with $O(n)$ CPUs.

Obliviously computing the routing permutation. Suppose that we are given a source array src of length n where each entry holds a distinct key, and a destination array dst also of length n where each entry holds a distinct key. Further, it is guaranteed that the set of keys in src is the same as the set of keys in dst. We would like to write down a permutation π (henceforth referred to as the routing permutation) such that applying π to src would result in the same order of keys as dst. The recent work by Chan and Shi [38] showed how to implement the above task obliviously using $O(1)$ number of oblivious sorts. Thus, with $O(n)$ CPUs the routing permutation can be computed in $O(\log n)$ parallel runtime.

Oblivious select. Consider the following problem: given a set of n elements among

which at most one element is distinguishing, output the distinguishing element (and if no element is distinguishing, output $\perp$). It is not difficult to see that by building an aggregation tree over the n elements, one can accomplish oblivious select with n CPUs in $\log n$ parallel steps.

Oblivious prefix sum. Given an array X of length n, every $i \in [n]$ wants to compute the sum of the prefix $X[1..i]$. There exists a parallel oblivious algorithm to achieve this in $O(\log n)$ steps consuming n CPUs [78].

Oblivious random permutation. Let ORP be an algorithm that upon receiving an input array X, outputs a permutation of X. Let $\mathcal{F}_{\text{perm}}$ denote an ideal functionality that upon receiving the input array X, outputs a perfectly random permutation of X.

We say that ORP is a *perfectly oblivious* random permutation, iff there exists a simulator Sim such that the joint distribution $(\mathcal{F}_{\text{perm}}(X), \mathsf{Sim}(|X|))$ is identically distributed as the joint distribution of the output and the addresses incurred by running ORP on X. Note that the simulator Sim is given only the input length $|X|$ but not the contents of X.

Chan, Chung, and Shi [35] recently describe a perfectly oblivious random permutation algorithm, which, except with negligible in λ probability, completes in $O(\log n + \alpha(\lambda))$ parallel steps consuming n CPUs assuming that the each block is large enough to store $\log \lambda$ bits (where α is a suitable super-constant function). We summarize their construction in the following theorem where we choose $\alpha(\lambda) := \log \log \lambda$ that will suffice for the purpose of this chapter.

Theorem 5 (Perfectly oblivious random permutation [35])**.** *Assume that each memory block is large enough to store at least* $\log \lambda$ *bits and that* $n \leq \lambda \leq 2^{O(n^2)}$*. Then, there exists a perfectly oblivious random permutation algorithm that consumes* n *CPUs.*

Except with λ *probability, the algorithm completes in* $O(\log n + \log\log \lambda)$ *parallel steps and* $O(n \log n)$ *work.*

We note that the failure is in terms of the algorithm's runtime — there is a negligibly small probability that the algorithm will run for longer, but the algorithm guarantees perfect security regardless.

4.2 Parallel One-Time Oblivious Memory

We define and construct an abstract datatype to process non-recurrent memory lookup requests. Although the abstraction is similar to the oblivious hashing scheme in Chan et al. [36], our one-time memory scheme needs to be perfectly secure and does not use a hashing scheme. Furthermore, we assume that every real lookup request is *tagged with a correct position label* that indicates where the requested block is — in this section, we simply assume that the correct position labels are simply provided during lookup; but later in our full OPRAM scheme, we will use a recursive ORAM/OPRAM technique reminiscent of those used in binary-tree-based ORAM/OPRAM schemes [38, 43, 134, 140, 143] such that we can obtain the position label of a block first before fetching the block.

4.2.1 Definition: One-Time Oblivious Memory

We describe the intuition using the *sequential* special case but our formal presentation later will directly describe the parallel version. An oblivious one-time memory supports three operations: 1) Build, 2) Lookup, and 3) Getall. Build is called once upfront to create the data structure: it takes in a set of real blocks (each tagged with its logical address) and creates a data structure that facilitates lookup. After this data structure is created, a sequence of lookup operations can be performed: each lookup can request a real block identified by its logical address or a dummy block denoted $\perp$ — if the requested block is a real block, we assume that the correct position label is supplied to indicate where in the data structure the requested block is. Finally, when the data structure is no longer needed, one may call a Getall operation to obtain a list of blocks (tagged with their logical addresses) that have not been looked up yet — in our OPRAM scheme later, this is the set of blocks that need to be preserved during rebuilding.

We require that our oblivious one-time memory data structure retain obliviousness as long as 1) the sequence of real blocks looked up all exist in the data structure (i.e., it appeared as part of the input to Build), and moreover, each logical address is looked up at most once; and 2) at most $\widetilde{n}$ number of dummy lookups may be made where $\widetilde{n}$ is a predetermined parameter (that the scheme is parametrized with).

4.2.1.1 Formal Definition

Our formal presentation will directly describe the parallel case. In the parallel version, lookup requests come in batches of size $m > 1$.

A (parallel) one-time memory scheme denoted $\mathsf{OTM}^{[n,m,t]}$ is parametrized by three parameters: n denotes the upper bound on the number of real elements; m is the batch size for lookups; t is the upper bound on the number of batch lookups supported.

The (parallel) one-time memory scheme $\mathsf{OTM}^{[n,m,t]}$ is comprised of the following possibly randomized, stateful algorithms to be executed on a *Concurrent-Read, Exclusive-Write* PRAM — note that since the algorithms are stateful, every invocation will update an implicit data structure in memory. Henceforth we use the terminology key and value in the formal description but in our OPRAM scheme later, a real key will be a logical memory address and its value is the block's content.

- $U \leftarrow \mathsf{Build}(\{(k_i, v_i) : i \in [n]\})$: given a set of n key-value pairs (k_i, v_i), where each pair is either real or of the form $(\perp, \perp)$, the Build algorithm creates an implicit data structure to facilitate subsequent lookup requests, and moreover outputs a list U of exactly n key-position pairs where each pair is of the form (k, pos). Further, every real key input to Build will appear exactly once in the list U; and the list U is padded with $\perp$ to a length n. Note that U does not include the values v_i's. Later in our scheme, this key-position list U will be

propagated back to the parent recursion depth during a coordinated rebuild[5].

- $(v_i : i \in [m]) \leftarrow \mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$: there are m concurrent Lookup operations in a single batch, where we allow each key k_i requested to be either real or $\perp$. Moreover, in each batch, at most n/t of the keys are real.

- $R \leftarrow \mathsf{Getall}$: the Getall algorithm returns an array R of length n where each entry is either $\perp$ or real and of the form (k, pos). The array R should contain all real entries that have been inserted during Build but have not been looked up yet, padded with $\perp$ to a length of n.

Valid request sequence. Our oblivious one-time memory ensures obliviousness only if lookups are non-recurrent (i.e., never look for the same real key twice); and moreover the number of lookups requests must be upper bounded by a predetermined parameter. More formally, a sequence of operations is valid, iff the following holds:

- The sequence begins with a single call to Build upfront; followed by a sequence of at most t batch Lookup calls, each of which supplies a batch of m keys and the corresponding position labels; and finally the sequence ends with a single call to Getall.

[5]Note that we do not explicitly denote the implicit data structure in the output of Build, since the implicit data structure is needed only internally by the current oblivious one-time memory instance. In comparison, U is explicitly output since U will later on be (externally) needed by the parent recursion depth in our OPRAM construction.

- The Build call is supplied with an input array $S := \{(k_i, v_i)\}_{i \in [n]}$, such that any two real entries in S must have distinct keys.

- For every $\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ query in the sequence, if each k_i is a real key, then k_i must be contained in S that was input to Build earlier. In other words, Lookup requests are not supposed to ask for real keys that do not exist in the data structure[6]; moreover, each (k_i, pos_i) pair supplied to Lookup must exist in the U array returned by the earlier invocation of Build, i.e., pos_i must be a correct position label for k_i; and

- Finally, in all Lookup requests in the sequence, no two keys requested (either in the same or different batches) are the same.

Correctness. Correctness requires that

1. for any valid request sequence, with probability 1, for every $\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ request, the i-th answer returned must be $\perp$ if $k_i = \perp$; else if $k_i \neq \perp$, Lookup must return the correct value v_i associated with k_i that was input to the earlier invocation of Build.

2. for any valid request sequence, with probability 1, Getall must return an array R containing every (k, v) pair that was supplied to Build but has not been looked up; moreover the remaining entries in R must all be $\perp$.

[6]We emphasize this is a major difference between this one-time memory scheme and the oblivious hashing abstraction of Chan et al. [36]); Chan et al.'s abstraction [36] allows lookup queries to ask for keys that do not exist in the data structure.

Perfect obliviousness. We say that two valid request sequences are *length-equivalent*, if the input sets to Build have equal size, and the number of Lookup requests (where each request asks for a batch of m keys) in the two sequences are the same.

We say that a (parallel) one-time memory scheme is perfectly oblivious, iff for any two length-equivalent request sequences that are valid, the distribution of access patterns resulting from the algorithms are *identically distributed*.

4.2.2 Construction

We first explain the intuition for the sequential case, i.e., $m = 1$. The intuition is simply to permute all elements received as input during Build. However, since subsequent lookup requests may be dummy (also denoted $\perp$), we also need to pad the array with sufficiently many dummies to support these lookup requests. The important invariant is that *each real element as well as each dummy will be accessed at most once* during lookup requests. For reals, this is guaranteed since the definition of a valid request sequence requires that each real key be requested no more than once, and that each real key requested must exist in the data structure. For dummies, every time a $\perp$-request is received, we always look for an unvisited dummy. To implement this idea, one tricky detail is that unlike real lookup requests, dummy requests do not carry the position label of the next dummy to be read — thus our data structure itself must maintain an *oblivious linked list* of dummies such that we can easily find out where the next dummy is. Since all real and dummies are

randomly permuted during Build, and due to the aforementioned invariant, every lookup visits a completely random location of the data structure thus maintaining perfect obliviousness.

It is not too difficult to make the above algorithm parallel (i.e., for the case $m > 1$). To achieve this, one necessary modification is that instead of maintaining a single dummy linked list, we now must maintain m dummy linked lists. These m dummy linked lists are created during Build and consumed during Lookup.

4.2.2.1 Detailed Construction

At the end of Build, our algorithm creates an in-memory data structure consisting of the following:

1. An array A of length $n + \widetilde{n}$, where $\widetilde{n} := tm$ denotes the number of dummies and n denotes the number of real elements. Each entry of the array A (real or dummy alike) has four fields $(\mathsf{key}, \mathsf{val}, \mathsf{next}, \mathsf{pos})$ where

    - key is a key that is either real or dummy; and val is a value that is either real or dummy.
    - the field $\mathsf{next} \in [0..n+\widetilde{n})$ matters only for dummy entries, and at the end of the Build algorithm, the next field stores the position of the next entry in the dummy linked list (recall that all dummy entries form m linked lists); and
    - the field $\mathsf{pos} \in [0..n+\widetilde{n})$ denotes where in the array an entry finally wants to be — at the end of the Build algorithm it must be that $A[i].\mathsf{pos} = i$.

However, during the algorithm, entries of A will be permuted transiently; but as soon as each element i has decided where it wants to be (i.e., $A[i].\mathsf{pos}$), it will always carry its desired position around during the remainder of the algorithm.

2. An array that stores the head pointers of all m dummy linked lists. Specifically, we denote the m head pointers as $\{\mathsf{dpos}_i : i \in [m]\}$ where each $\mathsf{dpos}_i \in [0..n+\widetilde{n})$ is the head pointer of one dummy linked list.

These in-memory data structures, including A and the dummy pointers will then be updated during Lookup.

Build. Our oblivious $\mathsf{Build}(\{(k_i, v_i)\}_{i\in[n]})$ algorithm proceeds as follows.

1. *Initialize.* Construct an array A of length $n + \widetilde{n}$ whose entries are of the form described above. Specifically, the keys and values for the first n entries of A are copied from the input. Recall that the input may contain dummies too, and we use $\perp$ to denote a dummy key from the input.

   The last $\widetilde{n}$ entries of A contain *special* dummy keys that are numbered. Specifically, for each $i \in [1..\widetilde{n}]$, we denote $A_n[i] := A[n-1+i]$, and the entry stored at $A_n[i]$ has key $\perp_i$ and value $\perp$.

2. *Every element decides at random its desired final position.* Specifically, perform a perfectly oblivious random permutation on the entries of A — this random permutation decides where each element finally wants to be.

   Now, for each $i \in [0..n+\widetilde{n})$, let $A[i].\mathsf{pos} := i$. At this moment, $A[i].\mathsf{pos}$ denotes

where the element $A[i]$ finally wants to be. Henceforth in the algorithm, the entries of A will be moved around but each element always carries around its desired final position.

3. *Construct the key-position map U.* Perform oblivious sorting on A using the field key. We assume that real keys have the highest priority followed by $\perp < \perp_1 < \cdots < \perp_{\widetilde{n}}$ (where smaller keys come earlier).

   At this moment, we can construct the key-position map U from the first n entries of A — recall that each entry of U is of the form (k, pos).

4. *Construct m dummy linked lists.* Observe that the last $\widetilde{n}$ entries of A contain special dummy keys, on which we perform the following to build m disjoint singly-linked lists (each of which has length t). For each $i \in [1..\widetilde{n}]$, if $i \mod t \neq 0$ we update the entry $A_n[i].\mathsf{next} := A_n[i+1].\mathsf{pos}$, i.e., each dummy entry (except the last entry of each linked list) records its next pointer.

   We next record the positions of the heads of the m lists. For each $i \in [m]$, we set $\mathsf{dpos}_i := A_n[t(i-1)].\mathsf{pos}$.

5. *Move entries to their desired positions.* Perform an oblivious sort on A, using the fourth field pos. (This restores the ordering according to the previous random permutation.)

At this moment, the data structure $(A, \{\mathsf{dpos}_i : i \in [m]\})$ is stored in memory. The key-position map U is explicitly output and later in our OPRAM scheme it will be passed to the parent recursion depth during coordinated rebuild.

Fact 6. *Consuming* $O(\widetilde{n}+n)$ *CPUs and setting* $(\widetilde{n}+n)^2 \le \lambda \le 2^{\widetilde{n}+n}$, *the* Build *algorithm completes in* $O(\log(\widetilde{n}+n)+\log\log\lambda)$ *parallel steps, except with probability negligible in* λ.

Proof. Observe that the algorithm's cost is dominated by $O(1)$ number of oblivious sorts which can be realized with the AKS sorting network [10].

Moreover, the algorithm incurs one application of oblivious random permutation, whose performance is stated in Theorem 5. □

Lookup. We implement a batch of m concurrent lookup operations $\{\mathsf{Lookup}(\{(k_i, \mathsf{pos}_i) : i \in [m]\})$ as follows. For each $i \in [m]$, we perform the following *in parallel*.

1. *Decide position to fetch from.* If $k_i \neq \perp$ is real, set $\mathsf{pos} := \mathsf{pos}_i$, i.e., we want to use the position label supplied from the input. Else if $k_i = \perp$, set $\mathsf{pos} := \mathsf{dpos}_i$, i.e., the position to fetch from is the next dummy in the i-th dummy linked lists. (To ensure obliviousness, the algorithm can always pretend to execute both branches of the if-statement.)

   At this moment, pos is the position to fetch from (for the i-th request out of m concurrent requests).

2. *Read and remove.* Read the value from $A[\mathsf{pos}]$ and mark $A[\mathsf{pos}] := \perp$.

3. *Update dummy head pointer if necessary.* If $\mathsf{pos} = \mathsf{dpos}_i$, update the dummy head pointer $\mathsf{dpos}_i := \mathsf{next}$. (To ensure obliviousness, the algorithm can pretend to modify dpos_i in any case.)

4. *Return.* Return the value read in the above Step 2.

The following fact is straightforward from the description of the algorithm.

Fact 7. *The* Lookup *algorithm completes in* $O(1)$ *parallel steps with* $O(m)$ *CPUs.*

Getall. Getall is implemented by the following simple procedure: obliviously sort A by the key such that all real entries are packed in front. Return the first n entries of the resulting array (and removing the metadata entries next and pos in the result).

Fact 8. *The* Getall *algorithm completes in* $\log(\widetilde{n}+n)$ *parallel steps consuming* $O(\widetilde{n}+n)$ *CPUs.*

Proof. Straighforward by observing that the algorithm's cost is dominated by $O(1)$ number of oblivious sorts which can be realized with the AKS sorting network [10].

□

Lemma 5 (Perfect obliviousness of the one-time memory scheme)**.** *The above (parallel) one-time memory scheme satisfies perfect obliviousness.*

Proof. It suffices to prove that for any valid request sequence, the memory access patterns are identically distributed as those output by the following simulator that knows only n, m and the number of Lookup requests in the sequence.

First, almost all parts of Build are deterministic and data oblivious and thus the algorithm's access patterns can be simulated in the most straightforward fashion. The only randomized part of access patterns for Build is due to the oblivious random permutation. To simulate this part, the simulator calls the oblivious random permutation's simulator algorithm.

Second, to simulate the access patterns of Lookup, the simulator would read the memory location storing dpos_i for every $i \in [m]$. Then, it reads a random unread

index of the array A and writes to it once too. Finally, it writes to dpos_i for every $i \in [m]$.

Third, simulating the access patterns of Getall is done in the most natural manner since Getall is deterministic.

It is not difficult to see that the real-world access patterns are identically distributed as the simulated ones due to the definition of oblivious random permutation (see Section 4.1.2) Particularly, observe that the above way of simulating the access patterns of Build is the same in nature as if we randomly permuted the data structure A upfront by a random permutation, (that is chosen independently from the simulated access patterns), then every real element and $\perp_i$ will be in a random location. Note also that as long as no two real keys requested collide and every real key requested exists in the data structure A, then the real-world algorithm accesses each real or $\perp_i$ element at most once, and thus every real-world access visits a random position of the array A (besides reading and writing $\{\mathsf{dpos}_i : i \in [m]\}$). □

Summarizing the above, we conclude with the following theorem.

Theorem 9 (One-time oblivious memory)**.** *Let $\lambda \in \mathbb{N}$ be a parameter related to the probability that the algorithm's runtime exceeds a desired bound. Assume that each memory block can store at least $\log n + \log \lambda$ bits. There exists a perfectly oblivious one-time scheme such that* Build *takes $O(\log n + \log\log \lambda)$ parallel steps (except with negligible in λ probability) consuming n CPUs,* Lookup *for a batch of m requests takes $O(1)$ parallel steps consuming m CPUs, and* Getall *takes $O(\log n)$ parallel steps consuming n CPUs.*

4.3 OPRAM with $O(\log^3 N)$ Simulation Overhead

We briefly explain the technical roadmap of this section:

- In Section 4.3.1, we will first describe a *position-based OPRAM* that supports two operations: Lookup and Shuffle. A position-based OPRAM is *an almost fully functional OPRAM scheme except that every real lookup request must supply a correct position label*. In our OPRAM construction, these position labels will have been fetched from small recursion depths and therefore will be ready when looking up the position-based OPRAM.

  Our position-based OPRAM relies on the hierarcial structure proposed by Goldreich and Ostrovsky [65,67], as well as techniques by Chan et al. [36] that showed how to parallelize such a hierarchical framework.

- In Section 4.3.2, we explain how to leverage "coordinated rebuild" and recursion techniques to build a recursive OPRAM scheme that composes logarithmically many instances of our position-based OPRAM, of geometrically decreasing sizes.

4.3.1 Position-Based OPRAM

Our OPRAM scheme (Section 4.3.2) will consist of logarithmically many position-based OPRAMs of geometrically increasing sizes, henceforth denoted OPRAM_0, OPRAM_1, OPRAM_2, ..., OPRAM_D where $D := \log_2 N - \log_2 m$. Specifically, OPRAM_d stores $\Theta(2^d \cdot m)$ blocks where $d \in \{0, 1, \ldots, D\}$. The last one OPRAM_D stores the

actual data blocks whereas every other OPRAM_d where $d < D$ recursively stores the position labels for the next depth $d + 1$.

4.3.1.1 Data Structure

As we shall see, the case OPRAM_0 is trivial and is treated specially at the end of this section (Section 4.3.1.1). Below we focus on describing OPRAM_d for some $1 \leq d \leq D = \log N - \log m$. For $d \neq 0$, each OPRAM_d consists of $d + 1$ *levels* geometrically growing in size, where each level is a *one-time oblivious memory scheme* as defined and described in Section 4.2. We specify this data structure more formally below.

Hierarchical levels. The position-based OPRAM_d consists of $d+1$ levels henceforth denoted as $(\mathsf{OTM}_j : j = 0, \ldots, d)$ where level j is a one-time oblivious memory scheme,

$$\mathsf{OTM}_j := \mathsf{OTM}^{[2^j \cdot m, m, 2^j]}$$

with at most $n = 2^j \cdot m$ real blocks and m concurrent lookups in each batch (which can all be real). This means that for every OPRAM_d, the smallest level is capable of storing up to m real blocks. Every subsequent level can store twice as many real blocks as the previous level. For the largest OPRAM_D, its largest level is capable of storing N real blocks given that $D = \log N - \log m$ — this means that the total space consumed is $O(N)$.

Every level j is marked as either *empty* (when the corresponding OTM_j has not been rebuilt) or *full* (when OTM_j is ready and in operation). Initially, all levels

are marked as empty, i.e., the OPRAM initially is empty.

Position label. Henceforth we assume that a position label of a block specifies 1) which level the block resides in; and 2) the position within the level the block resides at.

Additional assumption. We assume that each block is of the form (logical address, payload), i.e., each block carries its own logical address.

4.3.1.2 Operations

Each position-based OPRAM supports two operations, Lookup and Shuffle. For every OPRAM_d consisting of $d+1$ levels, we rely on the following algorithms for Lookup and Shuffle.

Lookup. Every batch lookup operation, denoted $\mathsf{Lookup}(\{(\mathsf{addr}_i, \mathsf{pos}_i) : i \in [m]\})$ receives as input the logical addresses of m blocks as well as a correct position label for each requested block. To complete the batch lookup request, we perform the following.

1. For each level $j = 0, \ldots, d$ *in parallel*, perform the following:

   - For each $i \in [m]$ in parallel, first check the supplied position label pos_i to see if the requested block resides in the current level j: if so, let $\mathsf{addr}'_i := \mathsf{addr}_i$ and let $\mathsf{pos}'_i := \mathsf{pos}_i$ (and specifically the part of the position label denoting the offset within level j); else, set $\mathsf{addr}'_i := \perp$ and $\mathsf{pos}'_i := \perp$ to indicate that this should be a dummy request.
   - $(v_{ij} : i \in [m]) \leftarrow \mathsf{OTM}_j.\mathsf{Lookup}(\{\mathsf{addr}'_i, \mathsf{pos}'_i : i \in [m]\})$.

2. At this point, each of the m CPUs has d answers from the d levels respectively, and only one of them is the valid answer. Now each of the m CPUs chooses the correct answer as follows.

   For each $i \in [m]$ in parallel: set val_i to be the only non-dummy element in $(v_{ij} : j = 0, \ldots, d)$, if it exists; otherwise set $\mathsf{val}_i := \perp$. This step can be accomplished using an oblivious select operation (see Section 4.1.2) in $\log d$ parallel steps consuming d CPUs.

3. Return $(\mathsf{val}_i : i \in [m])$.

We remark that in Goldreich and Ostrovsky's original hierarchical ORAM [65, 67], the hierarchical levels must be visited sequentially — for obliviousness, if the block is found in some smaller level, all subsequent levels must perform a dummy lookup. Here we can visit all levels in parallel since the position label already tells us which level it is in. Now the following fact is straightforward to observe:

Fact 10. *For* OPRAM_d, Lookup *consumes* $O(\log d)$ *parallel steps consuming* $m \cdot d$ *CPUs where* m *is the batch size.*

Shuffle. Similar to earlier hierarchical ORAMs [65, 67] and OPRAMs [36], a shuffle operation merges consecutively full levels into the next empty level (or the largest level). However, in our $\mathsf{Shuffle}$ abstraction, there is an input U that contains some logical addresses together with new values to be updated. Moreover, the shuffle operation is associated with an update function that determines how the new values in U should be incorporated into the OTM during the rebuild.

In our full OPRAM scheme later, the update array U will be passed from the immediate next depth OPRAM_{d+1}, and contains the new position labels that OPRAM_{d+1} has chosen for recently accessed logical addresses. These position labels must then be recorded by OPRAM_d appropriately.

More formally, each position-based OPRAM_d supports a shuffle operation, denoted $\mathsf{Shuffle}(U, \ell; \mathsf{update})$, where the parameters are explained as follows:

1. An update array U in which each (non-dummy) entry contains a logical address that needs to be updated, and a new value for this block. (Strictly speaking, we allow a block to be partially updated.)

   We will define additional constraints on U subsequently.

2. The level ℓ to be rebuilt during this shuffle.

3. An update function that specifies how the information in U is used to compute the new value of a block in the OTM.

   The reason we make this rule explicit in the notation is that a block whose address that appears in U may only be partially modified; hence, we later need to specify this update function carefully. However, to avoid cumbersome notation, we may omit the parameter update, and just write $\mathsf{Shuffle}(U, \ell)$, when the context is clear.

For each OPRAM_d, when $\mathsf{Shuffle}(U, \ell; \mathsf{update})$ is called, it must be guaranteed that $\ell \leq d$; and moreover, either level ℓ must either be empty or $\ell = d$ (i.e., this is the largest level in OPRAM_d). Moreover, there is an extra OTM'_0; jumping ahead,

we shall see that OTM'_0 contains the blocks that are freshly fetched.

The $\mathsf{Shuffle}$ algorithm then combines levels $0, 1, \ldots, \ell$ (of OPRAM_d), together with the extra OTM'_0, into level ℓ, updating some blocks' contents as instructed by the update array U and the update function update. At the end of the shuffle operation, all levels $0, 1, \ldots, \ell-1$ are now marked as empty and level ℓ is now marked as full.

We now explain the assumptions we make on the update array U and how we want the update procedure to happen:

- We require that each logical address appears at most once in U.
- Let A be all logical addresses remaining in levels 0 to ℓ in OPRAM_d: it must hold that the set of logical addresses in U is a subset of those in A. In other words, a subset of the logical addresses in A will be updated before rebuilding level ℓ.
- If some logical address addr exists only in A but not in U, after rebuilding level ℓ, the block's value from the current OPRAM_d should be preserved. If some logical address addr exists in both A and in U, we use the update function to modify its value: update takes a pair of blocks $(\mathsf{addr}, \mathsf{data})$ and $(\mathsf{addr}, \mathsf{data}')$ with the same address but possibly different contents (the first of which coming from the current OPRAM_d and the second coming from U), and computes the new block content data^* appropriately.

  We remark that the new value data^* might depend on both data and data'. Later, we will describe how the update rule is implemented.

Upon receiving $\mathsf{Shuffle}(U, \ell; \mathsf{update})$, proceed with the following steps:

1. Let $A := \cup_{i=0}^{\ell} \mathsf{OTM}_i.\mathsf{Getall} \cup \mathsf{OTM}'_0.\mathsf{Getall}$, where the operator $\cup$ denotes concatenation. Moreover, for an entry in A that comes from OTM_i, then it also carries a label i.

   At this moment, the old $\mathsf{OTM}_0, \ldots, \mathsf{OTM}_\ell$ instances may be destroyed.

2. We obliviously sort $A \cup U$ in increasing order of logical addresses, and moreover, placing all dummy entries at the end. If two blocks have the same logical address, place the entry coming from A in front of the one coming from U.

   At this moment, in one linear scan, we can operate on every adjacent pair of entries using the aforementioned update operation, such that if they share the same logical address, the first entry is preserved and updated to a new value, and the second entry is set to dummy.

   At this moment, we obliviously sort the resulting array moving all dummies to the end. We truncate the resulting array preserving only the first $2^\ell \cdot m$ elements and let A' denote the outcome (note that only dummies and no real blocks will truncated in the above step).

3. Next, we call $U' \leftarrow \mathsf{Build}(A')$ that builds a new OTM' and U' contains the positions of blocks in OTM'.

4. OTM' is now the new level ℓ and henceforth it will be denoted OTM_ℓ. Mark level ℓ as full and levels $0, 1, \ldots, \ell - 1$ as empty. Finally, output U' (in our full OPRAM construction later, U' will be passed to the next (i.e., immedi-

ately smaller) position-based OPRAM as the update array for performing its shuffle).

If we realize the oblivious sort with the AKS network [10] that sorts n items in $O(\log n)$ parallel steps consuming n CPUs, we easily obtain the following fact — note that there is a negligible in N probability that the algorithm runs longer than the stated asymptotic time due to the oblivious random permutation building block (see Section 4.1.2).

Fact 11. *Suppose that the* update *function can be evaluated by a single CPU in* $O(1)$ *steps. For* OPRAM_d*, let* $\ell \leq d$*, then except with negligible in* N *probability,* $\mathsf{Shuffle}(U, \ell)$ *takes* $O(\log(m \cdot 2^\ell) + \log\log N)$ *parallel steps consuming* $m \cdot 2^\ell$ *CPUs.*

Observe that in the above fact, the randomness comes from the oblivious random permutation subroutine used in building the one-time oblivious memory data structure.

Trivial case: OPRAM_0**.** In this case, OPRAM_0 simply stores its entries in an array $A[0..m)$ of size m and we assume that the entries are indexed by a $(\log_2 m)$-bit string. Moreover, each address is also a $(\log_2 m)$-bit string, whose block is stored at the corresponding entry in A.

- *Lookup.* Upon receiving a batch of m depth-m truncated addresses where all the real addresses are distinct, use oblivious routing to route $A[0..m)$ to the requested addresses. This can be accomplished in $O(m \log m)$ total work and $O(\log m)$ depth. Note that OPRAM_0's lookup does not receive any position labels.

- *Shuffle.* Since there is only one array A (at level 0), $\mathsf{Shuffle}(U, 0)$ can be implemented by oblivious sorting.

4.3.2 OPRAM Scheme from Position-Based OPRAM

Recursive OPRAMs. The OPRAM scheme consists of $D + 1$ position-based OPRAMs henceforth denoted as $\mathsf{OPRAM}_0, \mathsf{OPRAM}_1, \mathsf{OPRAM}_2, \ldots, \mathsf{OPRAM}_D$. OPRAM_D stores the actual data blocks, whereas every other OPRAM_d where $d \neq D$ recursively stores the position labels for the next data structure OPRAM_{d+1}. Our construction is in essence recursive although in presentation we shall spell out the recursion for clarity. Henceforth we often say that OPRAM_d is at *recursion depth* d or simply *depth* d.

Although we are inspired by the recursion technique for tree-based ORAMs [134], using this recursion technique in the context of hierarchical ORAMs/OPRAMs raises new challenges. In particular, we cannot use the recursion in a blackbox fashion like in tree-based constructions since all of our (position-based, hierarchical) OPRAMs must reshuffle in sync with each other in a non-blackbox fashion as will become clear later.

Format of depth-d block and address. Suppose that a block's logical address is a $\log_2 N$-bit string denoted $\mathsf{addr}^{\langle D \rangle} := \mathsf{addr}[1..(\log_2 N)]$ (expressed in binary format), where $\mathsf{addr}[1]$ is the most significant bit. In general, at depth d, an address $\mathsf{addr}^{\langle d \rangle}$ is the length-$(\log_2 m + d)$ prefix of the full address $\mathsf{addr}^{\langle D \rangle}$. Henceforth, we refer to $\mathsf{addr}^{\langle d \rangle}$ as a depth-d address (or the depth-d truncation of addr).

When we look up a data block, we would look up the full address $\mathsf{addr}^{\langle D \rangle}$ in recursion depth D; we look up $\mathsf{addr}^{\langle D-1 \rangle}$ at depth $D-1$, $\mathsf{addr}^{\langle D-2 \rangle}$ at depth $D-2$, and so on. Finally at depth 0, the $\log_2 m$-bit address uniquely determines one of the m blocks stored at OPRAM_0. Since each batch consists of m concurrent lookups, one of them will be responsible for this block in OPRAM_0.

A block with the address $\mathsf{addr}^{\langle d \rangle}$ *in* OPRAM_d *stores the position labels for two blocks in* OPRAM_{d+1}*, at addresses* $\mathsf{addr}^{\langle d \rangle}||0$ *and* $\mathsf{addr}^{\langle d \rangle}||1$ *respectively.* Henceforth, we say that the two addresses $\mathsf{addr}^{\langle d \rangle}||0$ and $\mathsf{addr}^{\langle d \rangle}||1$ are *siblings* to each other; $\mathsf{addr}^{\langle d \rangle}||0$ is called the left sibling and $\mathsf{addr}^{\langle d \rangle}||1$ is called the right sibling. We say that $\mathsf{addr}^{\langle d \rangle}||0$ is the left child of $\mathsf{addr}^{\langle d \rangle}$ and $\mathsf{addr}^{\langle d \rangle}||1$ is the right child of $\mathsf{addr}^{\langle d \rangle}$.

4.3.2.1 Operations

Each batch contains m requests denoted as $((\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i) : i \in [m])$, where for $\mathsf{op}_i = \texttt{read}$, there is no data_i. We perform the following steps.

1. **Conflict resolution.** For every depth $d \in \{0, 1, \ldots, D\}$ in parallel, perform oblivious conflict resolution on the depth-d truncation of all m addresses requested.

   For $d = D$, we suppress duplicate addresses. If multiple requests collide on addresses, we would prefer a write request over a read request (since write requests also fetch the old memory value back before overwriting it with a new value). In the case of concurrent write operations to the same address, we use the properties of the underlying PRAM to determine which write operation

prevails.

For $0 \leq d < D$, we perform the following:

(a) Consider the depth-(d+1) truncated address: $A^{\langle d+1 \rangle} := (\mathsf{addr}_1^{\langle d+1 \rangle}, \ldots, \mathsf{addr}_m^{\langle d+1 \rangle})$, and use oblivious sorting to suppress duplicates of depth-(d+1) addresses, i.e., each repeated depth-$(d+1)$ address is replaced by a dummy. Let $\widehat{A}^{\langle d+1 \rangle}$ be the resulting array (of size m) sorted by the (unique) depth-$(d+1)$ addresses.

(b) For each $i \in [1..m]$, we produce an entry $(\mathsf{addr}_i, \mathsf{flags}_i)$ according to the following rules:

i. If $\mathsf{addr}_i^{\langle d+1 \rangle}$ is a dummy, then $\mathsf{addr}_i := \perp$ is also dummy.

ii. If $\mathsf{addr}_i^{\langle d+1 \rangle}$ does not share its length-d prefix with $\mathsf{addr}_{i-1}^{\langle d+1 \rangle}$ or $\mathsf{addr}_{i+1}^{\langle d+1 \rangle}$, then addr_i is set to be the length-d prefix of $\mathsf{addr}_i^{\langle d+1 \rangle}$. Moreover, if $\mathsf{addr}_i^{\langle d+1 \rangle}$ ends with 0, then $\mathsf{flags}_i := 10$; otherwise, $\mathsf{flags}_i := 01$.

iii. If $\mathsf{addr}_i^{\langle d+1 \rangle}$ and $\mathsf{addr}_{i-1}^{\langle d+1 \rangle}$ share the same length-d prefix, then $\mathsf{addr}_i := \perp$; otherwise, if $\mathsf{addr}_i^{\langle d+1 \rangle}$ and $\mathsf{addr}_{i+1}^{\langle d+1 \rangle}$ share the same length-d prefix, then addr_i is set to the shared length-d prefix of the address, and $\mathsf{flags}_i := 11$.

(c) Then, the batch access for OPRAM_d is $((\mathsf{addr}_i, \mathsf{flags}_i) : i \in [m])$ where each non-dummy depth-d truncated address $\mathsf{addr}_i^{\langle d \rangle}$ is distinct and has a two-bit flags_i that indicates whether each of two addresses $(\mathsf{addr}_i^{\langle d \rangle}||0)$ and $(\mathsf{addr}_i^{\langle d \rangle}||1)$ is requested in OPRAM_{d+1}.

2. **Fetch.** For $d = 0$ to D sequentially, perform the following:

   - For each $i \in [m]$ in parallel: let $\mathsf{addr}_i^{\langle d \rangle}$ be the depth-d truncation of $\mathsf{addr}_i^{\langle D \rangle}$.
   - Call $\mathsf{OPRAM}_d.\mathsf{Lookup}$ to look up the depth-d addresses $\mathsf{addr}_i^{\langle d \rangle}$ for all $i \in [m]$; observe that position labels for the lookups of non-dummy addresses will be available from the lookup of the previous OPRAM_{d-1} for $d \geq 1$, which is described in the next step. Recall that for OPRAM_0, no position labels are needed.
   - If $d < D$, each lookup from a non-dummy $(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{flags}_i)$ will return two positions for the addresses $\mathsf{addr}_i^{\langle d \rangle}||0$ and $\mathsf{addr}_i^{\langle d \rangle}||1$ in OPRAM_{d+1}. The two bits in flags_i will determine whether each of these two position labels are needed in the lookup of OPRAM_{d+1}.

     We can imagine that there are m CPUs at recursion depth $d+1$ waiting for the position labels corresponding to $\{\mathsf{addr}_i^{\langle d+1 \rangle} : i \in [m]\}$. Now, using oblivious routing (see Section 4.1.2), the position labels can be delivered to the CPUs at recursion depth $d+1$.
   - If $d = D$, the outcome of Lookup will contain the data blocks fetched.

     Recall that conflict resolution was used to suppress duplicate addresses. Hence, oblivious routing can be used to deliver each data block to the corresponding CPUs that request it.
   - In any case, the freshly fetched blocks are updated if needed in the case of $d = D$, and are placed in OTM_0' in each OPRAM_d.

3. **Maintain.** We first consider depth D. Set depth-D's update array $U^{\langle D \rangle} := \emptyset$.

   Suppose that $\ell^{\langle D \rangle}$ is the smallest empty level in OPRAM_D.

   We have the invariant that for all $0 \le d < D$, if $\ell^{\langle D \rangle} < d$, then $\ell^{\langle D \rangle}$ is also the smallest empty level in OPRAM_d.

   For $d := D$ downto 0, do the following:

   - If $d < \ell^{\langle D \rangle}$, set $\ell := d$; otherwise, set $\ell := \ell^{\langle D \rangle}$.
   - Call $U \leftarrow \mathsf{OPRAM}_d.\mathsf{Shuffle}(U^{\langle d \rangle}, \ell; \mathsf{update})$ where update is the following natural function: recall that in $U^{\langle d \rangle}$ and OPRAM_{d-1}, each depth-$(d-1)$ logical address stores the position labels for both children addresses. For each of the child addresses, if $U^{\langle d \rangle}$ contains a new position label, choose the new one; otherwise, choose the old label previously in OPRAM_{d-1}.
   - If $d \ge 1$, we need to send the updated positions involved in U to depth $d-1$.

     Now, set $U^{\langle d-1 \rangle} \leftarrow \mathsf{Convert}(U, d)$, which will be used in the next iteration for recursion depth $d-1$ to perform its shuffle.

     The $\mathsf{Convert}$ subroutine takes an array that stores the position labels within OPRAM_d for depth-d addresses, and converts the array to one that contains depth-$(d-1)$ addresses where each entry may pack up to two position labels for its child addresses at depth-d.

     The subroutine $\mathsf{Convert}(U, d)$ proceeds as follows. First, perform oblivious sort on the depth-d addresses to produce an array denoted as $\{(\mathsf{addr}_i^{\langle d \rangle}, \mathsf{pos}_i) : i \in [|U|]\}$.

Next, for $i \in [|U|]$ in parallel, look to the left and look to the right and do the following:

- If $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_{i+1}^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my right neighbor is my sibling, then write down $u'_i = (\mathsf{addr}, (\mathsf{pos}_i, \mathsf{pos}_{i+1}))$, i.e., both siblings' positions need to be updated.
- If $\mathsf{addr}_{i-1}^{\langle d \rangle} = \mathsf{addr}||0$ and $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||1$ for some addr, i.e., if my left neighbor is my sibling, then write down $u'_i = \perp$.
- Else if i does not have a neighboring sibling, parse $\mathsf{addr}_i^{\langle d \rangle} = \mathsf{addr}||b$ for some $b \in \{0, 1\}$, then write down $u'_i = (\mathsf{addr}, (\mathsf{pos}_i, *))$ if $b = 0$ or write down $u'_i = (\mathsf{addr}, (*, \mathsf{pos}_i))$ if $b = 1$. In these cases, only the position of one of the siblings needs to be updated in OPRAM_{d-1}.
- Let $U^{\langle d-1 \rangle} := \{u'_i : i \in [|U|]\}$. Note here that each entry of $U^{\langle d-1 \rangle}$ contains a depth-$(d-1)$ address of the form addr, as well as the update instructions for two position labels of the depth-d addresses $\mathsf{addr}||0$ and $\mathsf{addr}||1$ respectively.

  We emphasize that when $*$ appears, this means that the position of the corresponding depth-d address does not need to be updated in OPRAM_{d-1}.
- Output $U^{\langle d-1 \rangle}$.

4.3.3 Analysis and Extensions

We now give detailed analysis and proofs for our OPRAM scheme.

4.3.3.1 Correctness and Obliviousness

Fact 12. *The above construction maintains correctness. More specifically, at every recursion depth d, the correct position labels will be input to the* Lookup *operations of* OPRAM_d*; and every batch of requests will return the correct answers.*

Proof. Straightforward by construction. □

In our OPRAM construction, for every OPRAM_d at recursion depth d, the following invariants are respected by construction as stated in the following facts.

Fact 13. *For every* OPRAM_d*, every* OTM_i *instance at level* $i \leq d$ *that is created needs to answer at most* 2^i *batches of* m *requests before* OTM_i *instance is destroyed.*

Proof. For every OPRAM_d, the following is true: imagine that there is a $(d+1)$-bit binary counter initialized to 0 that increments whenever a batch of m requests come in. Now, for $0 \leq \ell < d$, whenever the ℓ-th bit flips from 1 to 0, the ℓ-th level of OPRAM_d is destroyed; whenever the ℓ-th bit flips from 0 to 1, the ℓ-th level of OPRAM_d is reconstructed. For the largest level d of OPRAM_d, whenever the d-th (most significant) bit of this binary counter flips from 0 to 1 or from 1 to 0, the $(d+1)$-th level is destroyed and reconstructed. The fact follows in a straightforward manner by observing this binary-counter argument. □

Fact 14. *For every* OPRAM_d *and every* OTM_ℓ *instance at level* $\ell \leq d$*, during the lifetime of the* OTM_ℓ *instance: (a) no two real requests will ask for the same depth-d*

address; and (b) for every request that asks for a real depth-d address, the address must exist in OTM_i.

Proof. We first prove claim (a). Observe that for any OPRAM_d, if some depth-d address $\mathsf{addr}^{\langle d \rangle}$ is fetched from some level $\ell \leq d$, at this moment, $\mathsf{addr}^{\langle d \rangle}$ will either enter a smaller level $\ell' < \ell$; or some level $\ell'' \geq \ell$ will be rebuilt and $\mathsf{addr}^{\langle d \rangle}$ will go into level ℓ'' — in the latter case, level ℓ will be destroyed prior to the rebuilding of level ℓ''. In either of the above cases, due to correctness of the construction, if $\mathsf{addr}^{\langle d \rangle}$ is needed again from OPRAM_d, a correct position label will be provided for $\mathsf{addr}^{\langle d \rangle}$ such that the request will not go to level ℓ (until the level is reconstructed). Moreover, two real requests will not appear in the same request due to the conflict resolution procedure. Finally, claim (b) follows from correctness of the position labels. □

Given the above facts, our construction maintains perfect obliviousness.

Lemma 6 (Obliviousness)**.** *The above OPRAM construction satisfies perfect obliviousness.*

Proof. For every parallel one-time memory instance constructed during the lifetime of the OPRAM, Facts 13 and 14 are satisfied, and thus every one-time memory instance receives a valid request sequence. The lemma then follows in a straightforward fashion by the perfect obliviousness of the parallel one-time memory scheme, and by observing that all other access patterns of the OPRAM construction are deterministic and independent of the input requests. □

4.3.3.2 Asymptotical Complexity

We now analyze the asymptotical efficiency of our OPRAM construction. First, observe that the asymptotical performance of the fetch phase as stated in the following fact.

Fact 15. *The fetch phase can be completed in* $O(m \log^2 N)$ *total work, and in* $O((\log m + \log \log N) \cdot \log N)$ *depth (assuming an unbounded number of CPUs).*

Proof. For total work, it is not difficult to see that one $\log N$ factor arises from the recursion depths, and within each recursion depth it takes $O(m \log N + m \log m)$ work to perform the fetch. where $m \log m$ is the total work incurred by the oblivious routing in between recursion depths and $m \log N$ is the work incurred within a single position-based OPRAM.

For depth, one $\log N$ factor comes from the $\log N$ recursion depths, the other $(\log m + \log \log N)$ factor is due to the depth incurred by each recursion depth as well as due to the routing in between depths: 1) Within each recursion depth, it takes $O(1)$ depth to look up each of the up to $O(\log N)$ hierarchical levels, and then select the correct result in another $O(\log \log N)$ depth; and 2) the routing between adjacent depths can be implemented with the AKS sorting network [10] that takes $O(\log m)$ depth. □

We now proceed to analyze the efficiency of the maintain phase.

Fact 16. *Averaging over a sequence of batch accesses, the maintain phase costs*

$O(m \log^3 N)$ amortized total work (except with negligible in N probability). Further, for each batch of accesses, the maintain phase can always be completed in $O(\log^2 N)$ depth assuming an unbounded number of CPUs.

Proof. For each OPRAM_d, every level $\ell \leq d+1$ must be rebuilt after every 2^ℓ batch of m requests. Due to Fact 11, each rebuilding operation will take $O(2^\ell \cdot m \log(2^\ell \cdot m))$ total work, and has depth $O(\log(2^\ell \cdot m))$, which is at most $O(\log N)$. After the rebuilding, the $\mathsf{Convert}$ algorithm also has the same asymptotic performance. Thus, for each recursion depth, the amortized total work is $O(m \log^2 N)$. Counting all $O(\log N)$ recursion depths, we have the desired result for total work.

For depth, observe that for each recursion depth, the depth incurred by the rebuilding is dominated by the depth of the AKS sorting network which is $O(\log N)$. We then have the depth result by observing that the maintain phase is performed sequentially over $O(\log N)$ recursion depths. □

Lemma 7. *In the above OPRAM construction, the total work blowup is $O(\log^3 N)$, and the depth blowup is $O((\log m + \log \log N) \log N)$.*

Proof. Straightforward from Facts 15 and 16. □

Lemma 8. *The above OPRAM construction has an $O(1)$ space blowup.*

Proof. Our position-based OPRAM_d consists of $d+1$ levels – $(\mathsf{OTM}_j : j = 0, \ldots, d)$ where OTM_j is a one-time oblivious memory of size $O(m \cdot 2^j)$. Thus, the total space consumed by OPRAM_d is given by $\sum_{j=0}^{d}(m \cdot 2^j) = m \cdot 2^{d+1}$ blocks.

Our OPRAM construction consists of $D+1$ OPRAMs – $(\mathsf{OPRAM}_d : d =$

$0, \ldots, \log N - \log m$). Thus, the total space consumed our OPRAM scheme is given by $\sum_{d=0}^{\log N - \log m}(m \cdot 2^{d+1}) = m \cdot 2^{\log N - \log m + 2} = O(N)$ blocks. □

Corollary 17. *The above OPRAM construction incurs $O(\log^3 N)$ simulation overhead when consuming the same number of CPUs as the original PRAM.*

Proof. This corollary is implied directly by Lemma 7. The difference is that Lemma 7 would require more than m CPUs such that the depth of the algorithm may be smaller than the total work blowup, but if we are constrained to exactly m CPUs, the amortized parallel runtime per batch of accesses would be exactly $O(\log^3 N)$. □

Theorem 18. *The above construction is a perfectly secure OPRAM scheme satisfying the following performance overhead:*

- *When consuming the same number of CPUs as the original PRAM, the scheme incurs $O(\log^3 N)$ simulation overhead;*
- *When the OPRAM is allowed to consume an unbounded number of CPUs, the scheme incurs $O(\log^3 N)$ total work blowup and $O((\log m + \log\log N)\log N)$ depth blowup.*

In either case, the space blowup is $O(1)$.

Proof. Straightforward from Lemmas 7, 8, and Corollary 17. □

Note that at this moment, even for the sequential special case, we already achieve asymptotic savings over Damgård et al. [47] in terms of space consumption. Furthermore, Damgård et al. [47]'s construction is sequential in nature and does not immediately give rise to an OPRAM scheme.

4.3.4 Extension: Results for Large Block Sizes

Observe that if the block size is large, then each block in OPRAM_d can store more position identifiers for blocks in OPRAM_{d+1}. Hence, the number D of recursive OPRAMs can be reduced. This can lead to the following improvement.

Corollary 19 (Large Block Size)**.** *Suppose the block size is $\Theta(N^\epsilon)$ bits. Then, the above OPRAM construction can be modified to have $O(\frac{1}{\epsilon}\log^2 N)$ total work blowup and simulation overhead, and $O(\frac{1}{\epsilon}(\log m + \log\log N))$ depth blowup.*

Proof. When the block size is $B := \Theta(N^\epsilon)$ bits, the number of depths of recursive OPRAM's becomes $D := \frac{\log N}{\log \frac{B}{\log N}} = O(\frac{1}{\epsilon})$.

Hence, in every performance metric stated in Lemma 7 and Corollary 17, one factor of $\log N$ is replaced with $O(\frac{1}{\epsilon})$. □

4.4 Related Work

Before we conclude this chapter, we describe the work that is closely related to the result presented. Goldreich and Ostrovsky first showed a computationally secure ORAM scheme with poly-logarithmic simulation overhead. Therefore, one interesting question is whether ORAMs can be constructed without relying on computational assumptions. Ajtai [11] answered this question and showed that statistically secure ORAMs with poly-logarithmic simulation overhead exist. Although Ajtai removed computational assumptions from ORAMs, his construction has a (negligibly small) statistical failure probability, i.e., with some negligibly small probability, the

ORAM construction can leak information. Subsequently, Shi et al. [134] proposed the tree-based paradigm for constructing statistically-secure ORAMs. Tree-based constructions were later improved further in several works [38, 43, 61, 140, 143], and this line of works improve the practical performance of ORAM by several orders of magnitude in comparison with earlier constructions. It was also later understood that the tree-based paradigm can be used to construct computationally secure ORAMs saving yet another $\log \log$ factor in cost in comparison with statistical security [38, 56].

Perfectly secure ORAM [47] was first studied by Damgård et al. Perfect security requires that the (oblivious) program's memory access patterns be *identically distributed* regardless of the inputs to the program; and thus with probability 1, no information can be leaked about the secret inputs to the program. To date, Damgård et al.'s construction [47] remains the only known non-trivial, perfectly secure ORAM scheme. Their scheme achieves $O(\log^3 N)$ simulation overhead and $O(\log N)$ space blowup relative to the original RAM program. As mentioned, even for the sequential special case, our work asymptotically improves Damgård et al.'s result [47] by avoiding the $O(\log N)$ blowup in space; and moreover, our ORAM construction is conceptually simpler than that of Damgård et al.'s.

Oblivious Parallel ORAM (OPRAM) was first proposed in an elegant work by Boyle, Chung, and Pass [28], and subsequently improved in several followup works [35, 36, 38, 39, 117]. All known results on OPRAM focus on the statistically secure or the computationally secure setting. To the best of our knowledge, until this work, we know of no efficient OPRAM scheme that is perfectly secure. Chen,

Lin and Tessaro [39] introduced a generic method to transform any ORAM into an OPRAM at the cost of a $\log N$ blowup — their techniques achieve only *statistical* security too since security (or correctness) is only guaranteed with high probability (specifically, when some queue does not become overloaded in their scheme).

4.5 Conclusion and Future Work

In this chapter, we showed a construction for a perfectly secure OPRAM scheme with $O(\log^3 N)$ total work blowup, $O(\log N(\log m+\log\log N))$ depth blowup, and $O(1)$ space blowup. To the best of our knowledge our scheme is the first perfectly secure (non-trivial) OPRAM scheme, and even for the sequential special case we asymptotically improve the space overhead relative to Damgård et al. [47]. Prior to our work, the only known perfectly secure ORAM scheme is that by Damgård et al. [47], where they achieve $O(\log^3 N)$ simulation overhead and $O(\log N)$ space blowup. No (non-trivial) OPRAM scheme was known prior to our work, and in particular the scheme by Damgård et al. [47] does not appear amenable to parallelization. Finally, in comparison with known statistically secure OPRAMs [38,143], our work removes the dependence (in performance) on the security parameter; thus we in fact asymptotically outperform known statistically secure ORAMs [143] and OPRAMs [38] when (sub-)exponentially small failure probabilities are required.

Exciting questions remain open for future research:

- Can we construct perfectly secure ORAMs/OPRAMs whose total work blowup matches the best known statistically secure ORAMs/OPRAMs assuming neg-

ligible security failures?

- Can we construct perfectly secure ORAM/OPRAM schemes whose concrete performance lends to deployment in real-world systems?

Chapter 5: HOP: Hardware Makes Obfuscation Practical

Program obfuscation [14, 76] is a powerful cryptographic primitive, enabling numerous applications that rely on intellectually-protected programs and the safe distribution of such programs. For example, program obfuscation enables a software company to release software patches without disclosing the vulnerability to an attacker. It could also enable a pharmaceutical company to outsource its proprietary genomic testing algorithms, to an untrusted cloud provider, without compromising its intellectual properties. Here, the pharmaceutical company is referred to as the "sender" whereas the cloud provider is referred to as the "receiver" of the program.

Recently, the cryptography community has had new breakthrough results in understanding and constructing program obfuscation using multilinear maps [58]. However, cryptographic approaches towards program obfuscation have limitations. First, it is well-understood that strong (simulation secure) notions of program obfuscation cannot be realized in general [14] — although they are desired in many applications such as the aforementioned ones. Second, existing cryptographic constructions of obfuscation (that achieve imperfect notions of security, such as indistinguishability obfuscation [59]) incur prohibitive practical overheads, and are infeasible for most interesting application scenarios. For example, it takes ~ 3.3

hours to obfuscate even a very simple program such as an 80-bit point function (a function that is 0 everywhere except at one point) and $\sim$ 3 minutes to evaluate it [98]. Moreover, these cryptographic constructions of program obfuscation rely on new cryptographic assumptions whose security is still being investigated by the community through a build-and-break iterative cycle [40]. Thus, to realize a practical scheme capable of running general programs, it seems necessary to introduce additional assumptions.

In this direction, there has been work by both the cryptography and architecture communities in assuming trusted hardware storing a secret key. However, proposals from the cryptography community to realize obfuscation (and a closely related primitive called functional encryption) have been largely theoretical, focusing on what minimal trusted hardware allows one to circumvent theoretical impossibility and realize simulation-secure obfuscation [42, 52, 74]. Consequently these works have not focused on practical efficiency, and they often require running the program as circuits (instead of as RAM programs) and also utilize expensive cryptographic primitives such as fully homomorphic encryption (FHE) and non-interactive zero knowledge proofs (NIZKs). On the other hand, proposals from the architecture community such as Intel SGX [111], AEGIS [141], XOM [99], Bastion [34], Ascend [55] and GhostRider [102] are more practical, but their designs do not achieve cryptographic definition of obfuscation. In this work, we close this gap by designing and implementing a practical construction of program obfuscation for RAM programs using trusted hardware.

Problem statement. The problem of obfuscation can be described as follows. A

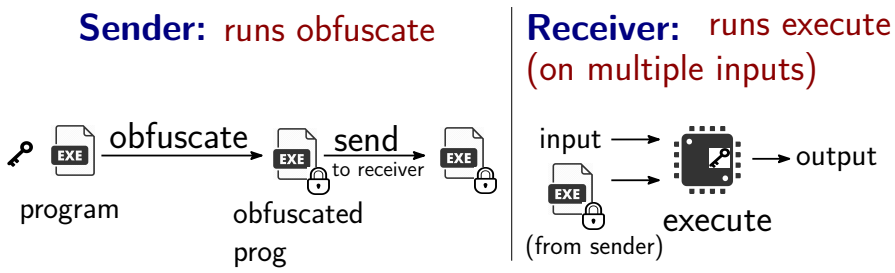


Figure 5.1: **Obfuscation Scenario.** The sender obfuscates programs using the obfuscate procedure. It sends (possibly multiple) obfuscated program(s) to the receiver. The receiver can execute any obfuscated program with any input of its choice.

sender, who owns a program, uses an obfuscate procedure to create an obfuscated program. It then sends this obfuscated program to a receiver who can execute the program on inputs of her choice. The obfuscated program should be functionally identical to the original program. For any given input, the obfuscated program runs for time T (fixed for the program) and returns an output.[1] The receiver only has a black box-like access to the program, i.e., it learns only the program's input/output behavior and the bound on the runtime T. In obfuscation, the inputs/outputs are public (not encrypted).

To make use of a trusted secure processor (which we call a *HOP processor*), our obfuscation model is modified as follows (cf. Figure 5.1). HOP processors are manufactured with a hardwired secret key. The HOP processor (which is trusted) is given to the receiver, and the secret key is given to the sender. Using the secret key, the sender can create multiple obfuscated programs using the obfuscate procedure and send them to the receiver. The receiver then runs the execute procedure (possi-

[1] T is analogous to a bound on circuit size in the cryptographic literature.

bly multiple times) to execute the program with (cleartext) inputs of her choice. As mentioned, the receiver (adversary) learns only the final outputs and nothing else. In other words, we offer virtual blackbox simulation security, where the receiver learns only as much information as if she were interacting with an oracle that computes the obfuscated program. In particular, the receiver should not learn anything from the HOP processor's intermediate behavior such as timing or memory access patterns, or the program's total runtime (since each program always runs for a fixed amount of time set by the sender).

Key distribution with public/private keys. We assume symmetric keys for simplicity. HOP may also use a private/public key distribution scheme common in today's trusted execution technology. The obfuscate and execute operations can be decoupled from the exact setup and key distribution system used to get public/private keys into the HOP processor. A standard setup for key distribution [75, 111] is as follows: First, a trusted manufacturer (e.g., Intel) creates a HOP processor with a unique secret key. Its public key is endorsed/signed by the manufacturer. Second, the HOP processors are distributed to receivers and the certified public keys are distributed to senders (software developers). Note that the key goal of obfuscation is to secure the sender's program and this relies on the secrecy of the private key stored in the processor. Thus, it is imperative that the sender and the manufacturer are either the same entity or the sender trusts the manufacturer to not reveal the secret key to another party.

Non-goals. We do not defend against analog side channels such as measuring power analysis or heat dissipation, we also do not defend against hardware fault

injection [9,25,89]. We assume that the program to be obfuscated is trustworthy and will not leak sensitive information on its own, including through possible software vulnerabilities such as buffer overflows [22]. There exist techniques to mitigate these attacks, and we consider them to be complementary to our work.

Challenges. It may seem that relying on secure hardware as described above easily 'solves' the program obfuscation problem. This is not the case: even with secure hardware, it is still not easy to develop a secure *and* practical obfuscation scheme. The crux of the problem is that many performance optimizations in real systems (and related work in secure processors [55, 102, 124]) hinge on exploiting *program-dependent behavior*. Yet, obfuscation calls for completely hiding all program-dependent behavior. Indeed, we started this project with a strawman processor that gives off the impression of executing any (or every) instruction during each time step – so as to hide the actual instructions being executed. Not surprisingly, this incurs huge ($\sim 10,000\times$; c.f. Section 5.2.2) overheads over an insecure scheme, even after employing a state-of-the-art Oblivious RAM [56, 67] to improve the efficiency of accessing main memory. Moreover, in an obfuscation setting, the receiver can run the same program multiple times for different inputs and outputs. Introducing practical features such as context switching — where the receiver can obtain intermediate program state — enables this level of flexibility but also enables new attacks such as *rewinding* and *mix-and-match* execution. Oblivious RAMs, in particular, are not secure against rewinding and mix-and-match attacks and an important challenge in this work is to protect them against said attacks in the context of the HOP system.

Our Contributions

Given the above challenges, a primary goal of this work is to develop and implement an optimized architecture that is still provably secure by the VBB obfuscation definition. In more detail, we make the following contributions:

1. Theoretical contributions: We provide the first theoretic framework to *efficiently* obfuscate RAM programs directly on secure hardware. One goal here is to avoid implicitly transforming the obfuscated program to its circuit representation (e.g., [52]), as the RAM to circuit transformation can incur a polynomial blowup in runtime [63]. We also wish for our analysis to capture important performance optimizations that matter in an implementation; such as the use of an Oblivious RAM, on-chip memory, instruction scheduling, and context switching. As a byproduct, part of our analysis achieves a new theoretical result (extending [74]): namely, how to provide program obfuscation for RAM programs *directly* assuming only 'stateless' secure hardware.[2] We also show interesting technical subtleties that arise in constructing efficient RAM-model program obfuscation from stateless hardware. In particular, we highlight the different techniques used to overcome all possible forms of *rewinding* and *mix-and-match* attacks (which may be of independent interest). Putting it all together, we provide a formal proof of security under the universally composable (UC) simulation framework [31].

[2]Roughly speaking, a HOP processor which allows the host to arbitrary context switch programs on/off the hardware is equivalent to 'stateless' hardware in the language of prior work [42,74]. This is explained further in Section 5.2.

2. Implementation with trusted hardware: We design and implement a hardware prototype system (called HOP) that attains the definition of program obfuscation and corresponds to our theoretic analysis. To the best of our knowledge, this effort represents the *first* prototype implementation of a provably secure VBB obfuscation scheme in *any* model under *any* assumptions. For performance, our HOP prototype uses a hardware-optimized Oblivious RAM, on-chip memory and instruction scheduling (our implementation does not support context switching). As mentioned earlier, our key differentiator from prior secure processor work is that our performance optimizations maintain *program privacy* and exhibit no program-dependent behavior. With these optimizations, HOP performs $5\times \sim 238\times$ better than the baseline HOP design across simple to sophisticated programs while the overhead over an insecure system is $8\times \sim 76\times$. The program code size overhead for HOP is only an *additive* constant. Our final design will require 72% area when synthesized on a commodity FPGA device. Of independent interest, we prove that our optimized scheme always achieves to within $2\times$ the performance of a scheme that does not protect the main memory timing channel (Section 5.2.3).

5.1 Related Work

We now describe work that is closely related to HOP.

Obfuscation and Oblivious RAMs. To enable running RAM programs directly on secure hardware, we use a hardware implementation of an ORAM [56,57] to hide access patterns to external memory. Using this, we describe a protocol to achieve the

definition of VBB obfuscation. Specifically, under VBB obfuscation, an adversary can execute a program multiple times with inputs of his choice and still not be able to learn the program. Interestingly, ORAMs were also originally introduced to prevent software piracy. Compared to an ORAM, we consider a setting weaker than the one considered by Goldreich and Ostrovsky. We assume a stateless trusted hardware token instead of a stateful token. We improve the resulting protocol while taking into consideration other side channels such as the timing channel attacks. Finally, we implement a prototype hardware that is capable of executing an obfuscated program.

Secure processors. Secure processors such as AEGIS [141], XOM [99], Bastion [34] and Intel SGX [111] encrypt and verify the integrity of main memory. Applications such as VC3 [131] that are built atop Intel SGX can run MapReduce computations [49] in a distributed cloud setting while keeping code and data encrypted. However, these secure processors do not hide memory access patterns. An adversary observing communication patterns between a processor and its memory can still infer significant information about the data [121, 156].

There have been some recent secure processor proposals that do hide memory access patterns [55, 102, 107, 124]. Ascend [55] is a secure processor architecture that protects privacy of data against physical attacks when running arbitrary programs. Phantom [107] similarly achieves memory obliviousness, and has been integrated with GhostRider [102] to perform program analysis and decide whether to use an encrypted RAM or Oblivious RAM for different memory regions. They also employ a scratchpad wherever applicable. Raccoon [124] hides data access patterns on

commodity processors by evaluating all program paths and using an Oblivious RAM in software.

The primary difference between the above schemes and HOP is the following. All of the above schemes focused on protecting input data, while the program is assumed to be public and known to the adversary. GhostRider [102] even utilizes public knowledge of program behavior to improve performance through static analysis. Conversely, obfuscation and HOP protect the program and the input data is controlled by the adversary. We remark, however, that HOP can be extended to *additionally* achieve data privacy simply by adding routines to decrypt the (now private) inputs and encrypt the final outputs before they are sent to the client (now different from the HOP processor owner). Naturally, the enhanced security comes with additional cost. We evaluate this overhead of additionally providing program-privacy by comparing to GhostRider in Section 5.5.5.

Obfuscation. The formal study of virtual black-box (VBB) obfuscation was initiated by Hada [76] and Barak et al. [14]. Unfortunately, Barak et al. showed that it is impossible to achieve program obfuscation for general programs. Barak et al. also defined a weaker notion of indistinguishability obfuscation ($i\mathcal{O}$), which avoids their impossibility results. Garg et al. [59] proposed a construction of $i\mathcal{O}$ for all circuits based on assumptions related to multilinear maps. However, these constructions are not efficient from a practical standpoint. There are constructions for $i\mathcal{O}$ for RAM programs proposed where the size of the obfuscated program is independent of the running time [21,32,90]. However, by definition, these constructions do not achieve VBB obfuscation.

In order to circumvent the impossibility of VBB obfuscation, Goyal et al. [74] considered virtual black-box obfuscators on minimal secure hardware tokens. Goyal et al. show how to achieve VBB obfuscation for all polynomial time computable functions using *stateless* secure hardware tokens that only perform authenticated encryption/decryption and a single NAND operation. In a related line of work, Döttling et al. [52] show a construction for program obfuscation using a single stateless hardware token in universally input-oblivious models of computation. Bitansky et al. [20] show a construction for program obfuscation from "leaky" hardware. Similarly, Chung et al. [42] considered basing the closely related primitive of functional encryption on hardware tokens. Unfortunately, all the above works require the obfuscated program run using a universal circuit (or similar model) to achieve function privacy. They do not support running RAM programs directly. This severely limits the practicality of the above schemes, as we demonstrate in Section 5.5.5.

Heuristic approaches to obfuscation. There are heuristic approaches to code obfuscation for resistance to reverse engineering [85, 130, 156]. These works provide low overheads, but do not offer any cryptographic security.

Terminology: Hardware Tokens. Trusted hardware is widely referred to as hardware *tokens* in the theoretical literature [42, 52, 74, 86]. Secure tokens are typically assumed to be minimal trusted hardware that support limited operations (e.g., a NAND gate in [74]). However, running programs in practice requires full-fledged processors. In this work, we refer to HOP as "secure hardware" or a "secure processor". As a processor, HOP will store a lot more internal state (e.g., a register file, etc.). We note that from a theoretic perspective, both HOP and 'simple' hardware

tokens require a number of gates which is polylogarithmic in memory size.

Terminology: Stateful vs. Stateless tokens. The literature further classifies secure tokens as either stateful tokens or stateless. A stateful token maintains state across invocations. On the other hand, a stateless token, except for a secret key, does not maintain any state across invocations. While HOP maintains state across most invocations for better performance, we will augment HOP to support on-demand context switching — giving the receiver the ability to swap out an obfuscated program for another at any time (Section 5.2.5), which is common in today's systems. In an extreme scenario, the adversary can context switch after every processor cycle. In this case, HOP becomes equivalent to a "stateless" token from a theoretical perspective [42, 74], and our security proof will assume stateless tokens.

5.2 Obfuscation from Trusted Hardware

In this section, we will intuitively describe the HOP architecture. We will start with an overview of a simple (not practical) HOP processor to introduce some key points. Each subsection after that introduces additional optimizations (some expose security issues, which we address) to make the scheme more practical. We give security intuition where applicable, and formally prove security for the fully optimized scheme in Section 5.3.

5.2.1 Execution On-Chip

Let us start with the simplest case where the whole obfuscated program and its data (working set) fit in a processor's on-chip storage. Then, we may architect a HOP processor to be able to run programs whose working sets don't exceed a given size. In the setup phase, first, the sender correctly determines a value T – the amount of time (in processor cycles) that the program, given any input, runs on HOP. Then, the sender encrypts (obfuscates) the program together using an authenticated encryption scheme. T is authenticated along and included with the program but is public. The obfuscated program is sent to the receiver. The receiver then sends the obfuscated program and her own input to the HOP processor. The HOP processor decrypts and runs the program, and returns a result after T processor cycles. The HOP processor makes no external memory requests during its execution since the program and data fit on chip. Security follows trivially.

5.2.2 Adding External Memory

Unfortunately, since on-chip storage is scarce (commercial processors have a few MegaBytes of on-chip storage), the above solution can only run programs with small working sets. To handle this, like any other modern processor, the HOP processor needs to access an external memory, which is possibly controlled by the malicious receiver.

When the HOP processor needs to make an access to this receiver memory, it needs to hide its access patterns. For the purposes of this discussion, the access

pattern indicates the processor's memory operations (reads vs. writes), the memory addresses for each access and the data read/written in each access. We hide access pattern by using an Oblivious RAM (ORAM), which makes a polylogarithmic number of physical memory accesses to serve each logical memory request from the processor [140]. The ORAM appears to HOP as an on-chip memory controller that intercepts memory requests from the HOP processor to the external memory. That is, the ORAM is a hardware block on the processor and is trusted. (More formal definitions for ORAM are given in Section 5.3.1.)

Each ORAM access can take thousands of processor cycles [56]. Executing instructions – once data is present on-chip – is still as fast as an insecure machine (e.g., several cycles). To hide when ORAM accesses are actually needed, HOP must make accesses at a static program-independent frequency (more detail below). As before, HOP runs for T time on all inputs and hence achieves the same privacy as the scheme in Section 5.2.1.

Generating T and security requirements. When accessing receiver-controlled memory, we must change T to represent some amount of work that is independent of the external memory's latency. That is, if T is given in processor cycles, the adversary can learn the true program termination time by running the program multiple times and varying the ORAM access latency each time (causing a different number of logical instructions to complete each time). To prevent this, we change T to mean 'the number of external memory read/writes made with the receiver.'

Integrity. To ensure authenticity of the encrypted program instructions and data during the execution, HOP uses a standard Merkle tree (or one that is integrated

with the ORAM [127]) and stores the root of a Merkle tree internally. The receiver cannot tamper with or rewind the memory without breaking the Merkle tree authentication scheme.

Efficiency. While the above scheme can handle programs with large working sets, it is very inefficient. The problem is that each instruction may trigger multiple ORAM accesses. To give off the impression of running any program, we must provision for this worst case: running each instruction must incur the cost of the worst-case number of ORAM accesses. This can result in $\sim 10,000\times$ slowdown over an insecure processor.[3] The next two subsections discuss two techniques to securely reduce this overhead by over two orders of magnitude. These ideas are based on well-known observations that many programs have more arithmetic instructions than memory instructions, and exhibit locality in memory accesses.

5.2.3 Adding Instruction Scheduling

The key intuition behind our first technique is that many programs execute multiple arithmetic instructions for every memory access. For example, an instruction trace may be the following: 'A A A A M A A M', where A, M refer to arithmetic and memory instructions respectively.

Our optimization is to let the HOP processor follow a fixed and pre-defined schedule: N arithmetic instructions followed by one memory access. In the above

[3] Our ORAM latency from Section 5.5 is 3000 cycles. The RISC-V ISA [33] we adopt can trigger 3 ORAM accesses, one to fetch the instruction, 1 or 2 more to fetch the operand, depending on whether the operand straddles an ORAM block boundary.

example, given a schedule of A^4M, the processor would insert two *dummy* arithmetic instructions to adhere to the schedule. A dummy arithmetic instruction can be implemented by executing a `nop` instruction. The access trace observable to the adversary would then be:

A A A A M A A **A** **A** M

The bold face **A** letters refer to dummy arithmetic instructions introduced by the processor.

Likewise, if another part of the program trace contains a long sequence of arithmetic instructions, the processor will insert dummy ORAM accesses to adhere to the schedule.

Gains. For most programs in practice, there exists a schedule with $N > 1$ that would perform better than our baseline scheme from Section 5.2.2. For $(N + 1)$ instructions, the baseline scheme performs $(N+1)$ arithmetic and memory accesses. With an A^NM schedule, our optimized scheme performs only one memory access which translates to a speedup of $N\times$ in the best case, when the cost of the memory access is much higher than an arithmetic instruction. To translate this into performance on HOP - given that HOP must run for T time - consider the following: If $N > 1$ does improve performance for the given program on all inputs, it means the sender can specify a smaller T for that program, while still having the guarantee that the program will complete given any input. A smaller T means better performance.

Setting N and security intuition. We design all HOP processors to use the same value of N for all programs and all inputs (i.e., N is set at HOP manufacturing time

like the private key). More concretely, we set

$$N = \frac{\text{ORAM latency}}{\text{Arithmetic latency}}$$

In other words, the number of processor cycles spent on arithmetic instructions and memory instructions are the same. For typical parameter settings, $N > 1000$ is expected. While this may sound like it will severely hurt performance given pathological programs, we show that this simple strategy does "well" on arbitrary programs and data, formalized below.

Claim: For *any* program and input, the above N results in $\leq 50\%$ of processor cycles performing dummy work.

Proof. Without loss of generality, we break up a program into a sequence of instruction *epochs*, where each epoch consists of a continuous run of arithmetic instructions followed by a continuous run of memory instructions. Denote the i-th epoch as $A^{n_i}M^{p_i}$. For example, the program

A A A A M A A M M M

has 2 epochs, with $n_1 = 4, p_1 = 1, n_2 = 2, p_2 = 3$.

Without loss of generality, we align the start of each epoch with the beginning of an $A^N M$ schedule. Given our choice of N, we examine the number of processor cycles spent doing dummy operations in each epoch. For the rest of the analysis, we abbreviate $|M| =$ ORAM latency and $|A| =$ Arithmetic latency.

Consider the start of epoch i (i.e., the first A instruction). To progress from the start of the epoch to the first M instruction (excluded) in the epoch, we perform $|A| * N * \lfloor \frac{n_i}{N} \rfloor + |A| * (n_i \mod N)$ real cycles and $|M| * \lfloor \frac{n_i}{N} \rfloor + |A| * (N - (n_i \mod N))$

dummy cycles worth of work. To progress from the first M instruction (including) to the end of the epoch, we perform $|M| * p_i$ real cycles and $|A| * N * (p_i - 1)$ dummy cycles worth of work. Note that by our definitions of epochs, we have that $p_i \geq 1$.

Also note that $|M| = |A| * N$ by our choice of N. Combining these two time periods, we spend $|M| * (\lfloor \frac{n_i}{N} \rfloor + p_i) + |A| * (n_i \mod N)$ real cycles and $|M| * (\lfloor \frac{n_i}{N} \rfloor + p_i - 1) + |A| * (N - (n_i \mod N))$ dummy cycles worth of work. □

The claim implies that in comparison to a solution that *does not* protect the main memory timing channel, our fixed schedule introduces a maximum overhead of 2× given any program – whether they are memory or computation intensive. Said another way, even when more sophisticated heuristics than a fixed schedule are used for different applications, the performance gain from those techniques is a factor of 2 at most.

Security. We note that our instruction scheduling scheme does not impact security because we use a fixed, public N for all programs.

5.2.4 Adding on-chip Scratchpad Memory

Our second optimization adds a scratchpad: a *small* unit of trusted memory (RAM) inside the processor, accesses to which are not observable by the adversary.[4] It is used to temporarily store a portion of the working set for programs that exhibit locality in their access patterns.

Running programs with a scratchpad. We briefly cover how to run programs

[4]We remark that we use a software-managed scratchpad (as opposed to a conventional processor cache) as it is easier to determine T when using a scratchpad.

using a scratchpad here. More (implementation-specific) detail is given in Section 5.4.1. At a high level, data is loaded into the scratchpad from ORAM/unloaded to ORAM using special (new) CPU instructions that are added to the obfuscated program. These instructions statically determine when to load which data to specified offsets in the scratchpad. Now, the scratchpad load/unload instructions are the only instructions that access ORAM (i.e., are the only 'M' instructions). Memory instructions in the original program (e.g., normal loads and stores) merely lookup the scratchpad inside the processor (these are now considered 'A' instructions). We will assume the program is correctly compiled so that whenever a program memory instruction looks up the scratchpad, the data in question has been put there sometime prior by a scratchpad load/unload instruction.

Security intuition. When the program accesses the scratchpad, it is hidden from the adversary since this is done on-chip. As before, the only adversary-visible behavior is when ORAM is accessed and this will be governed by the program-independent schedule from Section 5.2.3.

Program independence. We note that HOP with a scratchpad is still program independent. Multiple programs can be written (and obfuscated) for the same HOP processor. One minor limitation, however, is that once an obfuscated program is compiled, it must be compiled with 'minimum scratchpad size' specified as a new parameter and cannot be run on HOP processors that have a smaller scratchpad. This is necessary because having a smaller scratchpad will increase T by some unknown amount. If the program is run on a HOP processor with a larger scratchpad, it will still function but some scratchpad space won't be used.

Gains. In the absence of a scratchpad, the ratio of arithmetic to memory instructions is on average 5:1 for our workloads. When using a scratchpad, a larger amount of data is stored by the processor, thus decreasing memory accesses. This effectively decreases the execution time T of the program and substantially improves performance for programs with high locality (evaluated in Section 5.5.3).

5.2.5 Adding context switching and stateless tokens

For the solutions discussed until now, once a program is started, it cannot be stopped until it returns a response. But a user may wish to concurrently run multiple obfuscated programs for a practical deployment model. Therefore, we design the HOP processor to support on-demand context switch, i.e., the receiver can invoke a context switch at any point during execution. This, however, introduces security problems that we need to address.

A context switch means that the current program state should be swapped out from the HOP processor and replaced with another program's state. Since such a context switch can potentially happen at every invocation, one can potentially think of the HOP processor as storing no state, i.e., it is a stateless token. In such a scenario, we design it to encrypt all its internal state, and send this encrypted/authenticated state (denoted $\overline{\mathsf{state}}$) to the receiver (i.e., the adversary) on a context switch. Whenever the receiver passes control back to the token, it will pass back the encrypted state as well, such that the token can "recover" its state upon every invocation.

Challenges. Although on the surface, this idea sounds easy to implement, in reality it introduces avenues for new attacks that we now need to defend against. For the rest of the chapter, and in-line with real processors, we assume the only data that remains in HOP is the per-chip secret key. A notable attack is the *rewinding* attack. In this attack, instead of passing to the token the correct and fresh encrypted $\overline{\mathsf{state}}$ as well as fresh values of memory reads, a malicious receiver can pass old values.[5] The receiver can also *mix-and-match* values from entirely different executions of the same program or different programs. The rest of the section outlines how to prevent the above attacks. We remark that while the below have simple fixes, the problems themselves are easy to overlook and underscore the need for a careful formal analysis. Indeed, we discovered several of these issues while working through the security proof itself.

Preventing mix-and-match. To prevent this attack, we enforce that the receiver must submit an encrypted state $\overline{\mathsf{state}}$, corresponding to an execution at some point t, along with a matching read from time t for the same execution. To achieve this, observe that $\overline{\mathsf{state}}$ is encrypted with a IND-CPA + INT-CTXT-secure authenticated

[5] Here is a possible attack by which the adversary can distinguish between two access patterns. Consider the access pattern $\{a, a\}$ i.e., accessing the same block consecutively. If a tree-based ORAM [134] is used, after the first access, the block is remapped to a new path l' and the new path l' would be subsequently accessed. If the adversary rewinds and executes again, the block may be mapped to a different path l''. Thus, for two different executions, two different paths (l' and l'') are accessed for the second access. Note that for another access pattern $\{a, b\}$ for $a \neq b$, the same paths would be accessed even after rewinding, thus enabling the adversary to distinguish between access patterns.

encryption scheme, and that the state carries all necessary information to authenticate the next memory read. The state contains information unique to the specific program, the specific program execution, and to the specific instruction that the token expects.

Preventing rewinding during program execution. An adversary may try to gain more information by rewinding an execution to a previous time step, and replaying it from that point on. To prevent an adversary from learning more information in this way, we make sure that the token simply replays an old answer should rewinding happen — this way, the adversary gains no more information by rewinding. To achieve this, we make sure that any execution for a (program, inp) pair is entirely deterministic no matter how many times you replay it. All randomness required by the token (e.g., those required by the ORAM or memory checker) are generated pseudorandomly based on the tuple $(K,\ H_S,\ H_R)$ where K is a secret key hardwired in the token, H_R is a commitment to the receiver's input and $H_S := \mathsf{digest}(\overline{\mathsf{mem}_0})$ is a Merkle root of the program.

Preventing rewinding during input insertion. In our setting, the obfuscated program's inputs inp are chosen by the receiver. Since inputs can be long, it may not be possible to submit the entire input in one shot. As a result, the receiver has to submit the input word by word. Therefore the malicious receiver may rewind to a point in the middle of the input submission, and change parts of the input in the second execution. Such a rewinding causes two inputs to use the same randomness for some part of the execution.

To prevent such an input rewinding attack, we require that the adversary sub-

mit a Merkle tree commitment $H_R := \mathsf{digest}(\mathsf{inp})$ of its input inp upfront, before submitting a long input word by word. H_R uniquely determines the rest of the execution, such that any rewinding will effectively cause the token to play old answers (as mentioned above), and the adversary learns nothing new through rewinding.

5.3 Formal Scheme

We now give a formal model for the fully optimized HOP processor (i.e., including all subsections in Section 5.2) and prove its security in UC framework. Section 5.3.1 describes the preliminaries. Section 5.3.2 describes the ideal functionality for obfuscation of RAM programs. Sections 5.3.3 and 5.3.4 describe our formal scheme and proof in the UC framework.

5.3.1 Preliminaries

The notations used in this section are summarized in Table 5.1. We denote the assignment-operator with :=, while we use = to denote equality. For succinctness, encryption of data is denoted by an overline, e.g., $\overline{\mathsf{state}} = \mathsf{Enc}_K(\mathsf{state})$, where Enc denotes a IND-CPA + INT-CTXT-secure authenticated encryption scheme and K is the key used for encryption.

Universal Composability framework. The Universal Composability framework [31] considers two worlds – 1. *real world* where the parties execute a protocol π. An adversary $\mathcal{A}$ controls the corrupted parties. 2. *ideal world* where we assume the presence of a trusted third party. The parties interact with a trusted third party

(also called ideal functionality $\mathcal{F}$) with a protocol ϕ. A simulator $\mathcal{S}$ tries to mimic the actions of $\mathcal{A}$. Intuitively, the amount of information revealed by π in the real world should not be more than what is revealed by interacting with the trusted third party in the ideal world. In other words, we have the following: an environment $\mathcal{E}$ observes one of the two worlds and guesses the world. Protocol π UC-realizes ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$, such that an environment $\mathcal{E}$ cannot distinguish (except with negligible probability) whether it is interacting with $\mathcal{S}$ and ϕ or with $\mathcal{A}$ and π.

Remark: ORAM initialization. In this chapter, we assume an ORAM starts out with a memory array where the first N words are non-zero (reflecting the initial unshuffled memory), followed by all zeros. Most ORAM schemes require an initialization procedure to shuffle the initial memory contents. We assume that the ORAM algorithm performs a linear scan of first N memory locations and inserts them into ORAM. This is used by the simulator in our proof to extract the input used for execution of the program. We use the convention that such initialization is performed by the ORAM algorithm upon the first read or write operation — therefore our notation does not make such initialization explicit. This also means that the first ORAM operation will incur a higher overhead than others.

5.3.2 $\mathcal{F}_{obf}^{\mathcal{RAM}}$: Modeling Obfuscation in UC

The ideal functionality for obfuscation $\mathcal{F}_{obf}^{\mathcal{RAM}}$ is described in Figure 5.2. The sender sends the description of a RAM program, $\mathsf{RAM} \in \mathcal{RAM}$ and a program

Table 5.1: Notations

K	Hardwired secret key stored by the token
$\mathsf{mem}_{\text{init}}$	A program as a list of instructions
inp	Input to the program
mem	Memory required for program execution
outp	Program output
$\ell_{\mathsf{in}}, \ell_{\mathsf{out}}, B$	Bit-lengths of input, output, and memory word
N	Number of words in memory
T	Time for program execution
$\mathsf{RAM.params}$	$\{T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, B\}$
$\mathsf{oramstate}$	State stored by ORAM
$\mathsf{sstorestate}$	State stored by sstore
H_R	Digest of receiver's input, i.e., $\mathsf{digest}(\mathsf{inp})$
H_S	Digest of sender's program, i.e., $\mathsf{digest}(\overline{\mathsf{mem}_{\text{init}}})$
H'	Merkle root of the main memory

ID pid, using the "create" query. The functionality stores this program, pid, the sender and receiver. When the receiver invokes "execute" query on an input inp, it evaluates the program on inp, and returns output outp.

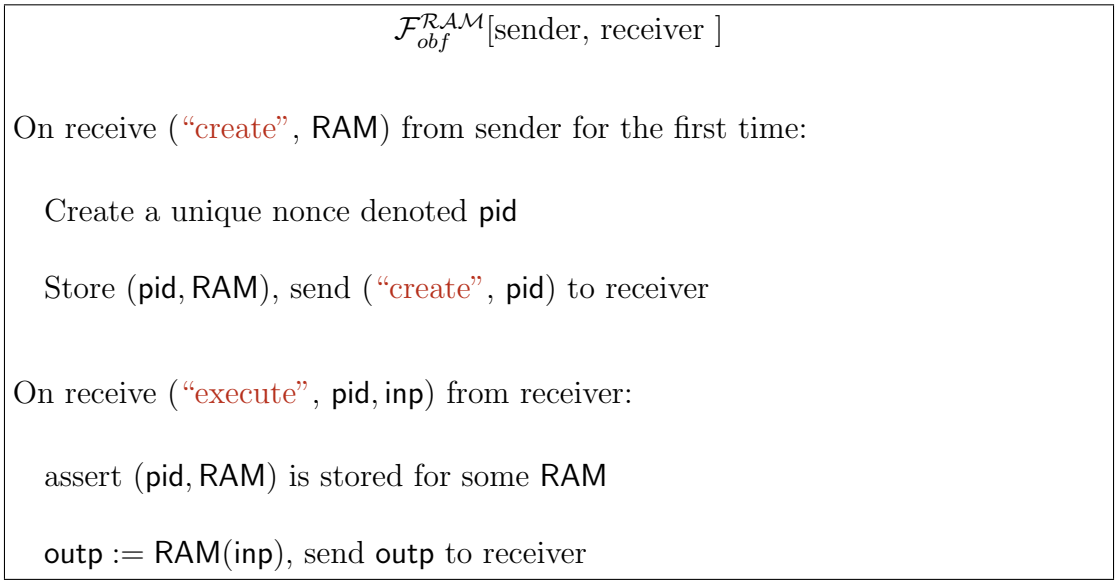


Figure 5.2: **Ideal Functionality** $\mathcal{F}_{obf}^{\mathcal{RAM}}$. Although there can be multiple instances of this ideal functionality, we omit writing the session identifier explicity without risk of ambiguity.

5.3.3 Scheme Description

We now provide the complete description of our scheme. We model the secure hardware token through the $\mathcal{F}_{token}$ functionality (Figures 5.3 and 5.4). Our construction realizes $\mathcal{F}_{obf}^{\mathcal{RAM}}$ in the $\mathcal{F}_{token}$-hybrid model [74] and is described in Figure 5.5.

In order to account for all possible token queries that may be required for an ORAM scheme, $\mathcal{F}_{token}$ relies on an internal, transient instance of $\mathcal{F}_{internal}$ to execute each step of the program evaluation. Each time $\mathcal{F}_{token}$ yields control to the receiver, the entire state of $\mathcal{F}_{internal}$ is destroyed. Whenever the receiver calls back $\mathcal{F}_{token}$ with $\overline{\mathsf{state}}$, $\mathcal{F}_{token}$ once again creates a new, transient instance of $\mathcal{F}_{internal}$, sets its state to the decrypted state, and invokes $\mathcal{F}_{internal}$ to execute next step.

The sender. Let the program to be obfuscated be RAM with an initial CPU state $\mathsf{cpustate}_{\text{init}}$ and a list of program instructions $\mathsf{mem}_{\text{init}}$. The sender first creates the token containing a hardwired secret key K where $K := (K_1, K_2, K_3)$. K_1 is used as the encryption key for encrypting state, K_2 is used as the key to a pseudorandom function used by the ORAM and K_3 is used as the key for a pseudorandom function used by sstore (described later). This is modeled by our functionality using the "store key" query (Figure 5.5 line 1). The sender then encrypts mem_0 (one instruction at a time) to obtain $\overline{\mathsf{mem}_{\text{init}}}$. It creates a Merkle root $H_S := \mathsf{digest}(\overline{\mathsf{mem}_{\text{init}}})$, which is used by $\mathcal{F}_{token}$ during execution to verify integrity of the program. The sender creates an encrypted header $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_{\text{init}}, H_S, \mathsf{RAM.params})$ where $\mathsf{RAM.params} = \{T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, B\}$. The sender sends $\overline{\mathsf{header}}$, $\overline{\mathsf{mem}_{\text{init}}}$, and $\mathsf{RAM.params}$ as the obfuscated program to the receiver. As the obfuscated program consists of only the encrypted program and metadata, for a program of size P bits, the obfuscated program has size $P + O(1)$ bits. In the real world, the sender sends the hardware token with the functionality $\mathcal{F}_{token}$ to the receiver. The receiver can use the same stateless token to execute multiple obfuscated programs sent by the sender.

The receiver. On the receiver's side, the token functionality makes use of an ORAM and a secure store sstore. The token functionality (trusted hardware functionality) is modeled by an augmented RAM machine.

1. **ORAM.** ORAM takes in $[\kappa := \mathsf{PRF}_{K_2}(\mathsf{ssid}), \mathsf{oramstate}]$ (where $\mathsf{ssid} := (H_S, H_R)$) as internal secret state of the algorithm. κ is a session-specific seed

$\mathcal{F}_{token}$ [sender, receiver]

// Store the secret key K in the token

On receive ("store key", K) from sender:

Store the secret key K, ignore future "store key" inputs

Send "done" to sender

// This step commits the receiver to his input through H_R

On receive ("initialize", $\overline{\mathsf{header}}, H_R$) from receiver:

Parse $K := (K_1, K_2, K_3)$

$(\mathsf{cpustate}_0, H_S, \mathsf{RAM.params}) := \mathsf{Dec}_{K_1}(\overline{\mathsf{header}})$; abort if fail

$\mathsf{state} := \{\mathsf{ssid} := (H_S, H_R),\ \mathit{time} := 0,\ \mathsf{rdata} := 0,\ \mathsf{cpustate} := \mathsf{cpustate}_{\text{init}},$

$\mathsf{sstorestate} := (\text{"init"}, H_S, H_R, H' := 0),$

$\mathsf{oramstate} := \text{"init"}, \mathsf{params} := \mathsf{RAM.params}\}$

send $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\mathsf{state})$ to receiver

On receive (_) from $\mathcal{F}_{internal}$: // ORAM queries

$\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\mathcal{F}_{internal}.\mathsf{state})$, Send (_, $\overline{\mathsf{state}}$) to receiver

On receive (_, $\overline{\mathsf{state}}$) from receiver: // ORAM queries

$\mathsf{state} := \mathsf{Dec}_{K_1}(\overline{\mathsf{state}})$, abort if fail

Instantiate $\mathcal{F}_{internal}$, set $\mathcal{F}_{internal}.\mathsf{state} := \mathsf{state}$, and $\mathcal{F}_{internal}.K := K$

Send _ to $\mathcal{F}_{internal}$

Figure 5.3: **Functionality $\mathcal{F}_{token}$.** For succinctness, encryption of some data is represented using an overline on it, e.g., $\overline{\mathsf{state}} = \mathsf{Enc}_{K_1}(\mathsf{state})$, where Enc denotes a IND-CPA + INT-CTXT-secure authenticated encryption scheme. "_" denotes a wildcard field that matches any string.

$\mathcal{F}_{internal}$

Define $\mathcal{F}_{internal}.\mathsf{state} \stackrel{\text{alias}}{:=} (\mathsf{ssid}, time, \mathsf{rdata}, \mathsf{cpustate}, \mathsf{sstorestate}, \mathsf{oramstate}, \mathsf{params})$

// execute program

On receive ("execute one step") from $\mathcal{F}_{token}$:

1: assert $time \leq \mathsf{params}.T$

2: $(\mathsf{cpustate}, \mathsf{op}) \leftarrow \Pi'(\mathsf{cpustate}, \mathsf{rdata})$

3: Send op to $\mathsf{ORAM}[\mathsf{PRF}_{K_2}(\mathsf{ssid}), \mathsf{oramstate}] \Leftrightarrow \mathsf{sstore}[\mathsf{PRF}_{K_3}(\mathsf{ssid}), \mathsf{sstorestate}] \Leftrightarrow \mathcal{F}_{token}$, wait for output from ORAM, abort if sstore aborts; /* instantiate ORAM with state oramstate, instantiate sstore with state sstorestate, connect ORAM's communication tape to sstore's input tape, connect sstore's communication tape to caller $\mathcal{F}_{token}$. This represents a multi-round protocol. */

4: If $\mathsf{op} = (\texttt{read}, \ldots)$, let $\mathsf{rdata} :=$ output

5: $time := time + 1$

6: If $time = \mathsf{params}.T$: send ("okay", rdata) to $\mathcal{F}_{token}$; else send ("okay", $\perp$) to $\mathcal{F}_{token}$

Figure 5.4: **Functionality $\mathcal{F}_{internal}$.**

$\mathbf{Prot}_{obf}$[sender, receiver]

Sender:

On receive ("create", $\mathsf{RAM} = \langle \mathsf{cpustate}_{\text{init}}, \mathsf{mem}_{\text{init}} \rangle$) from env:

1: If not initialized: $K := (K_1, K_2, K_3) \xleftarrow{\$} \{0,1\}^{3\lambda}$, send ("store key", K) to $\mathcal{F}_{token}$, await "done"

2: $\overline{\mathsf{mem}_{\text{init}}} := \{\mathsf{Enc}_{K_1}(\mathsf{mem}_{\text{init}}[i], \mathsf{rand}())\}_{i \in |\mathsf{mem}_{\text{init}}|}$

3: $H_S := \mathsf{digest}(\overline{\mathsf{mem}_{\text{init}}})$ // H_S: program Merkle root

4: $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_{\text{init}} || H_S || \mathsf{RAM.params}, \mathsf{rand}())$

5: Send $(\overline{\mathsf{header}}, \overline{\mathsf{mem}_{\text{init}}}, \mathsf{RAM.params})$ to receiver

Receiver:

On receive ("execute", $\mathsf{pid}, \mathsf{inp}$) from env:

1: Await $(\overline{\mathsf{header}}, \overline{\mathsf{mem}_{\text{init}}}, \mathsf{RAM.params})$ from sender s.t. $\mathsf{RAM.params}.H_S = \mathsf{pid}$ if not received already

2: Initialize $\mathsf{mem} := \overline{\mathsf{mem}_{\text{init}}} || \mathsf{inp} || \vec{0}$

3: Send ("initialize", $\overline{\mathsf{header}}$, $H_R := \mathsf{digest}(\mathsf{inp})$) to $\mathcal{F}_{token}$, await $\overline{\mathsf{state}}$ from $\mathcal{F}_{token}$

4: **for** t in $\{1, \ldots, T\}$:

5: Send ("execute one step", $\overline{\mathsf{state}}$) to $\mathcal{F}_{token}$

6: Await $(\mathsf{oper}, \overline{\mathsf{state}})$ from $\mathcal{F}_{token}$; // $\overline{\mathsf{state}}$ overwritten with the received value

7: Until $\mathsf{oper} = (\text{"okay"}, \_)$, repeat: //multiple requests due to ORAM

8: perform the operation oper on mem and let the response be res

9: forward $(\mathsf{res}, \overline{\mathsf{state}})$ to $\mathcal{F}_{token}$, and await $(\mathsf{oper}, \overline{\mathsf{state}})$ from $\mathcal{F}_{token}$;

10: Parse $\mathsf{oper} := (\text{"okay"}, \mathsf{outp})$, output outp

Figure 5.5: **Protocol $\mathbf{Prot}_{obf}$.** Realizes $\mathcal{F}_{obf}^{\mathcal{RAM}}$ in the $\mathcal{F}_{token}$-hybrid model.

used to generate all pseudorandom numbers needed by the ORAM algorithm — recall that all randomness needed by ORAM is replaced by pseudorandomness to avoid rewinding attacks. As mentioned in Section 5.3.1, we assume that the ORAM initialization is performed during the first read/write operation. At this point, the ORAM reads the first N memory locations to read the program and the input, and inserts them into the ORAM data structure within mem.

2. **Secure store module sstore.** sstore is a stateful deterministic secure storage module that sits in between the ORAM module and the untrusted memory implemented by the receiver. Its job is to provide appropriate memory encryption and authentication. sstore's internal state includes $\kappa := \mathsf{PRF}_{K_3}(\mathsf{ssid})$ and $\mathsf{sstorestate}$. $\mathsf{sstorestate}$ contains a succinct digest of program, input and memory to perform memory authentication. κ is a session-specific seed used to generate all pseudorandom numbers for memory encryption.

   At the beginning of an execution, $\mathsf{sstorestate}$ is initialized to $\mathsf{sstorestate} := (H_S, H_R, H' := 0)$, where H_S denotes the Merkle root of the encrypted program provided by the sender, H_R denotes the Merkle root of the (cleartext) input and H' denotes the Merkle root of the memory mem. By convention, we assume that if a Merkle tree or any subtree's hash is 0, then the entire subtree must be 0. The operational semantics of sstore is as follows: upon every data access request $(\texttt{read}, addr)$ or $(\texttt{write}, addr, \mathsf{wdata})$:

   - If $addr$ is in the $\mathsf{mem}_{\text{init}}$ part of the memory (the sender-provided en-

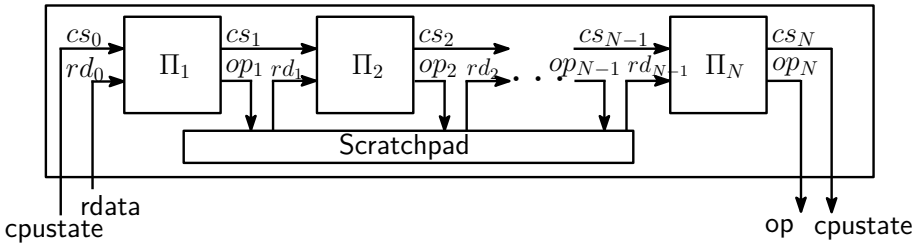


Figure 5.6: **Augmented Random Access Machine.** In this figure, $\mathsf{cpustate}_i$ is denoted by cs_i and rdata_i is denoted by rd_i.

crypted program), interact with mem and use H_S to verify responses. Update H_S appropriately if the request type is `write`.

- If *addr* is in the inp part of the memory (the receiver-provided input), interact with mem and use H_R to verify responses.
- Otherwise, interact with mem and use H' to verify responses. Update H' appropriately.

Upon successful completion, sstore outputs the data fetched for read requests, and outputs 0 or 1 for write requests. Note that the sstore algorithm simply aborts if any of the responses fail verification.

3. **Augmented Random Access Machines.** We now extend the RAM model to support instruction scheduling and a scratchpad (Sections 5.2.3 and 5.2.4). $\mathcal{RAM}$ can be augmented to use a next instruction circuit $\Pi' := \Pi^N$ for a fixed N, with the following modifications:

   (a) Π' is a combinational circuit, which consists of N next-instruction circuits Π_i cascaded as shown in Figure 5.6.

   (b) The Π_i's use an additional shared memory, referred to as scratchpad. Each

Π_i (except Π_1) operates on the output of Π_{i-1} and an operand rdata_{i-1} read from scratchpad. The next instruction circuit Π' outputs op_N to retrieve rdata from mem, which is subsequently used by Π_1.

On input inp, the execution of $\mathsf{RAM}[T, N, \ell_{\mathsf{in}}, \ell_{\mathsf{out}}, B]$ with parameters $\Pi', \mathsf{cpustate}, \mathsf{mem}\rangle$ is similar to what was defined in Section 2.1 but uses Π' as the next instruction circuit. The augmented random access machine $\mathcal{RAM}'$ models a RAM that performs N instructions followed by an ORAM access. If some op_i cannot be served by the scratchpad, subsequent Π_j for $i+1 \leq j \leq N$ do not update $\mathsf{cpustate}_j$ and output $\mathsf{op}_N = \mathsf{op}_i$ to load the required data in scratchpad.

Remark. For augmented random access machines that uses a scratchpad, rdata would typically be larger than a memory word (e.g. 512 bits).

We now explain how the receiver executes the program using the token described in Figure 5.3 and protocol in Figure 5.5.

Program execution. For ease of explanation, let us first assume that the ORAM is initialized and contains the program and input. The execution for any input proceeds in T time steps (Figure 5.5 line 4). At each time step, the receiver interacts with the token with two types of queries. For each type of query, $\mathcal{F}_{token}$ decrypts $\overline{\mathsf{state}}$ (aborts if decryption fails), instantiates $\mathcal{F}_{internal}$ with state and forwards the request to $\mathcal{F}_{internal}$. At the end of query, the $\overline{\mathsf{state}}$ is sent to the receiver along with the query response.

- **Execute one step**: This is shown in Figure 5.3 and Figure 5.5 line 5. When this query is invoked, $\mathcal{F}_{internal}$ executes the next instruction circuit Π' of the

RAM machine to obtain an updated cpustate and an op $\in$ {`read, write`}. Once operation op is performed by the ORAM algorithm, $\mathcal{F}_{internal}$ updates state.*time* to reflect the execution of the instruction (Figure 5.3 line 5). The message "okay" is then sent to the receiver. At $time = T$, $\mathcal{F}_{internal}$ returns the program output to the receiver (Figure 5.3 line 6).

- **ORAM queries**: ORAMs can use a multi-round protocol (with possibly different types of queries) to read/write (Figure 5.3 line 3). It interacts with mem stored at the receiver through $\mathcal{F}_{token}$ (Figure 5.5 lines 7-9). To account for instantiation of any ORAM, $\mathcal{F}_{token}$ is shown to receive any query from receiver (indicated by wildcard (_) in Figures 5.3 and 5.5). These queries are sent to $\mathcal{F}_{internal}$ and vice-versa.

For each interaction with mem, sstore encrypts (resp. decrypts) data sent to (resp. from) the receiver. Moreover, sstore authenticates the data sent by the receiver. This completes the description of execution of the program.

Initialization. To initialize the execution, the receiver first starts by storing the program and input inp in its memory $\mathsf{mem} := \overline{\mathsf{mem}_{\text{init}}}||\mathsf{inp}||\vec{0}$. It commits to its input by invoking "initialize" (Figure 5.5 line 3) and sending a Merkle root of its input ($H_R = \mathsf{digest}(\mathsf{inp})$) along with $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\mathsf{cpustate}_{\text{init}}||H_S||\mathsf{RAM.params})$. $\mathcal{F}_{token}$ initializes the parameters, creates $\overline{\mathsf{state}}$ and sends it to the receiver.

The ORAM and sstore are initialized during the first invocation to "execute one step", i.e., $t = 1$ in Figure 5.5, line 4. The required randomness is generated pseudorandomly based on (K_2, H_S, H_R) for ORAM and (K_3, H_S, H_R) for sstore. As mentioned

in Section 5.3.1, during initialization, ORAM in $\mathcal{F}_{token}$ reads mem_0 word by word (not shown in figure). For each word read, sstore performs Merkle tree verification with $H_s := \mathsf{digest}(\mathsf{mem}_{\text{init}})$. Similarly, when the input is read, sstore verifies it with $H_R := \mathsf{digest}(\mathsf{inp})$. sstorestate and oramstate uniquely determine the initialization state. Hence, if the receiver rewinds, the execution trace remains the same. The commitment H_R ensures that the receiver cannot change his input after invoking "initialize". This completes the formal scheme description of the UC functionality $\mathcal{F}_{token}$.

5.3.4 Proof of Security

Theorem 20. *Assuming that Enc is an INT-CTXT + IND-CPA authenticated encryption scheme,* ORAM *satisfies obliviousness (Section 5.3.1),* sstore *adopts a semantically secure encryption scheme and a collision resistant Merkle hash tree scheme and the security of* PRF*, the protocol described in Figures 5.3 and 5.5 UC realizes $\mathcal{F}_{obf}^{\mathcal{RAM}}$ (Figure 5.2) in the $\mathcal{F}_{token}$-hybrid model.*

Description of the simulator. The ideal world simulator simulates the honest sender and $\mathcal{F}_{token}$.

- The simulator can receive the following three types of valid queries: "create" queries, "initialize" queries, and execution queries. An execution query is either of the format ("execute one step", $\overline{\mathsf{state}}$) or a response to a memory request. Henceforth we assume that all responses to memory requests are of the form ("mem", _, $\overline{\mathsf{state}}$).

- At the beginning, the simulator generates a random key K_1.
- Whenever the simulator receives ("create", pid) from $\mathcal{F}_{obf}^{\mathcal{RAM}}$, it sends $\overline{\mathsf{mem}_{\text{init}}} := \{\mathsf{Enc}_{K_1}(\vec{0})\}_{i \in |\mathsf{mem}_{\text{init}}|}$. $\overline{\mathsf{header}} := \mathsf{Enc}_{K_1}(\vec{0}||H_S := \mathsf{digest}(\overline{\mathsf{mem}_{\text{init}}})||\mathsf{RAM.params})$, and RAM.params to the receiver.
- Whenever the adversary (i.e., receiver) sends ("initialize", $\overline{\mathsf{header}}, H_R$): if the simulator has not sent the adversary $\overline{\mathsf{header}}$ before as a result of a "create" query, abort. At this time, a new subsession identified by ssid $:= (H_S, H_R)$ is created where H_S is contained in $\overline{\mathsf{header}}$, and sstorestate and oramstate for this subsession are initialized honestly.

  The simulator sends $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$ to the adversary.
- For each subsession identified by ssid, the simulator maintains the following:
  - If $\overline{\mathsf{state}}$ was sent to the adversary during a subsession ssid, then the simulator remembers a tuple

    $$(\mathsf{sstorestate}, \mathsf{oramstate})$$

    which denotes the state of the execution when $\overline{\mathsf{state}}$ was sent to the adversary.
  - The simulator forks a new ORAM simulator upon the creation of every new subsession, the term oramstate is the state of the (stateful) ORAM simulator.
  - The simulator forks an honest instance of sstore upon the creation of

every new subsession. The sstore instance is initialized with a randomly generated key, and the term sstorestate is its internal state.

- Whenever the simulator receives any execution query from the adversary, it checks if the $\overline{\mathsf{state}}$ received has been sent to the adversary before. If not, the simulator aborts. Else, continue with the following.
  - If the adversary has sent the same execution query before, the simulator replays the old answer from before with the following exception: for the $\overline{\mathsf{state}}$ contained in the answer, the simulator will re-encrypt $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$.
  - Otherwise, if $\overline{\mathsf{state}}$ was sent to the adversary earlier as part of a memory request, then the simulator must check the correctness of the response returned by the adversary. To achieve this, the simulator retrieves the sstorestate at the time $\overline{\mathsf{state}}$ was sent to the adversary. Now, using this sstorestate, the simulator runs the honest sstore instance corresponding to the current subsession to check the correctness of memory request. If the check fails, the simulator simply aborts.
  - If the simulator has not aborted, the simulator retrieves the oramstate at the time $\overline{\mathsf{state}}$ was sent to the adversary. Now, the simulator calls the ORAM simulator to obtain the next memory request — we assume that the ORAM simulator will return the same answer when invoked with the same oramstate.

    If the next memory request is a write request, then the data for the write

is set to a dummy message $\vec{0}$, and then this message is passed along to the sstore instance (which internally performs memory encryption). The outcome of sstore along with a fresh encryption $\overline{\mathsf{state}} := \mathsf{Enc}_{K_1}(\vec{0})$ is sent to the adversary.

Remark 1. *Notice that in the simulation, multiple ciphertexts* $\overline{\mathsf{state}}$ *can correspond to the same point of execution in a subsession. However, if the adversary rewinds the execution of a subsession, all other parts of the response it obtains will be deterministic (i.e., same as the last time) if the simulation does not abort.*

We now show that the view of the adversary in a real execution is indistinguishable from that in the above described simulated execution. We show this indistinguishability between the real and ideal world by using the following sequence of hybrids:

Hybrid H_0**:** This is the real world execution. The simulator simulates the honest sender and hence, has access to the sender's program. It uses the token $\mathcal{F}_{token}$ to respond to queries by the adversary (receiver).

Hybrid H_1**:** This hybrid is identical to hybrid H_0 except for the following. For integrity verification, instead of using the Merkle tree scheme, the simulator performs an honest memory check. Merkle tree checks are performed for the memory with Merkle root H', program $\mathsf{mem}_{\mathrm{init}}$ with Merkle root H_S and input with Merkle root H_R.

For each subsession ssid, the simulator maintains a table storing the requests sent by the adversary and the responses sent by the simulator. Specifically, it stores a table consisting of decrypted request state ($\mathsf{oramstate}$ and $\mathsf{sstorestate}$, denoting the execution state), decrypted response state and the snapshot of memory mem'. When $\overline{\mathsf{state}}$ is sent as a part of memory request in an execution query, instead of using Merkle root H', the simulator verifies the correctness by running an honest sstore instance for the subsession. Specifically, the simulator looks up the table by response state and compares the response sent by adversary with mem'. The simulator aborts if the comparison fails. Otherwise, it runs a simulated execution of the token and responds to the adversary.

Recall that as mentioned in the scheme, during an initial linear scan for initializing ORAM, both the program and the input are loaded. When program is loaded during this initialization, the simulator looks up the table based on the decrypted request state and authenticates the program by comparing it to $\mathsf{mem}_{\text{init}}$ instead of using the Merkle root H_S. When the adversary initializes execution, the simulator saves the commitment H_R of the adversary's input in ssid. When the adversary sends a ($\overline{\mathsf{state}}$, input value), the simulator looks up the table by decrypted response state to find the request state sent by the adversary. If it finds an entry, the simulator verifies the correctness of the input value using the Merkle root H_R. *This is where the simulator extracts input from the adversary.* It should be noted that H_R is generated by the adversary. By the collision resistance of hash functions used by Merkle trees, the adversary cannot generate two inputs with the same Merkle root H_R.

Thus, the only event in which this hybrid differs from H_0 is if the adversary breaks the collision resistance of hash functions used by Merkle trees. In this case, the simulator aborts. The probability of this bad event is negligible; this can be shown by reducing the security to a Merkle tree game between a challenger and an adversary. In the absence of this bad event, this hybrid is identical to H_0.

Hybrid H_2: This hybrid is identical to the previous one except for the following. The pseudorandom functions (PRF) are replaced with truly random functions, i.e., whenever PRF function (i) with key K_2 is invoked by the ORAM algorithm and (ii) key K_3 is invoked by sstore in hybrid H_2, the simulator samples a random number instead. In the protocol, the use of a PRF while initializing ORAM ensures that the ORAM accesses memory locations in a deterministic manner. Hence, if the adversary rewinds execution, the same memory locations are accessed by the ORAM algorithm. In this hybrid, we replace PRF with truly random function. Based on the decrypted request state, the simulator determines if the adversary has sent the same query before. If yes, the simulator replays the old answer by using the same randomness in the ORAM algorithm. Otherwise, the simulator looks up the table by response state to determine if this state was sent to the adversary as part of a memory request. If yes, the simulator generates new random numbers to be used by the ORAM algorithm and stores the request and response states along with the random numbers in the table. Except for these changes, the simulator computes the query response as in hybrid H_1. The simulator sends the updated response $\overline{\mathsf{state}}$ to the adversary.

This hybrid is computationally indistinguishable from hybrid H_1 by the security of pseudorandom functions. If the adversary can distinguish between this hybrid and hybrid H_1, we can show a reduction to a game where the adversary can distinguish between a pseudorandom function and a truly random *function* with non-negligible probability.

Hybrid H_3: This hybrid is identical to the previous one except for the following. The simulator replaces the authentication scheme used to verify state with an honest check. In order to do so, along with the other information stored in the table in hybrid H_2, the simulator also stores the encrypted request and response $\overline{\mathsf{state}}$. Note that this is when the simulator starts storing multiple ciphertexts $\overline{\mathsf{state}}$ corresponding to the same point of execution in a subsession.

Instead of using an authentication scheme to verify state, it compares the $\overline{\mathsf{state}}$ sent by the adversary with any of the $\overline{\mathsf{state}}$ values that was previously sent by the simulator. If the simulator does not find an entry in the table, it aborts. If the check succeeds, the simulator can determine the exact execution state based on $\mathsf{oramstate}$ and $\mathsf{sstorestate}$. The simulator knows the program mem_0 from the honest sender, extracts the input inp from the adversary (as described in hybrid H_1) and has stored a snapshot of all random numbers that were generated for ORAM. Hence, by simulating an instance of the token, the simulator can compute the exact response state that needs to be returned without decrypting the request sent by the adversary. The simulator encrypts this response state and sends the encrypted $\overline{\mathsf{state}}$ to the adversary. If the execution proceeds without aborting until time T, then at

T-th step, the simulator calls the ideal functionality $\mathcal{F}_{obf}^{\mathcal{RAM}}$ on input inp and sends the output outp to the adversary.

The computational indistinguishability of this hybrid from hybrid H_2 follows from the INT-CTXT security of the authenticated encryption scheme Enc. If the adversary can distinguish between hybrid H_2 and this hybrid with non-negligible probability, we can show a reduction where the adversary can break the security of an INT-CTXT secure authenticated encryption game with non-negligible probability.

Hybrid H_4: This hybrid is identical to the previous one except for the following. The simulator generates a random key K_1. For all $\overline{\mathsf{state}}$ and memory writes by sstore sent to the adversary, the simulator instead sends $Enc_{K_1}(\vec{0})$. Also, the simulator begins execution by sending $\overline{\mathsf{mem}_{\text{init}}} := \{Enc_{K_1}(\vec{0}\}_{i \in |\mathsf{mem}_{\text{init}}|}$, $\overline{\mathsf{header}} := Enc_{K_1}(\vec{0}||H_s := \mathsf{digest}(\overline{\mathsf{mem}_{\text{init}}}||\mathsf{RAM.params})$ and $\mathsf{RAM.params}$ to the adversary.

Given the security of an IND-CPA secure authenticated encryption scheme Enc, this hybrid is identically distributed as the previous hybrid. If the adversary can distinguish between this hybrid and hybrid H_3, we can show a reduction where the adversary can break the security of an IND-CPA secure authenticated encryption scheme.

Hybrid H_5: This hybrid is identical to the previous one except for the following. Instead of the execution of an ORAM algorithm, the simulator invokes an ORAM

simulator. For an execution query, after checking the correctness of the response sent by the adversary, the simulator retrieves the $\mathsf{oramstate}$ and invokes the ORAM simulator with this state. We assume that the ORAM simulator will return the same answer when invoked with the same $\mathsf{oramstate}$. The output of the ORAM simulator is sent to the adversary. Assuming that the ORAM scheme is statistically secure, this hybrid is statistically indistinguishable from hybrid H_4.

In this hybrid, the simulator uses an ORAM simulator for ORAM requests, performs ideal checks for the memory that needs to be stored by the adversary and for the token state sent to the adversary, uses truly random functions, and sends $Enc_{K_1}(\vec{0})$ to the adversary. The simulator knows the program $\mathsf{mem}_{\text{init}}$ from the sender, extracts the program input inp from the adversary as described in hybrid H_1. If the execution proceeds until T steps without aborting, the simulator internally simulates an instance of the ideal functionality $\mathcal{F}_{obf}^{\mathcal{RAM}}$ to obtain the output outp. The simulator in this hybrid behaves exactly as the ideal world simulator described earlier. Hence, this is the ideal world execution.

5.4 Implementation

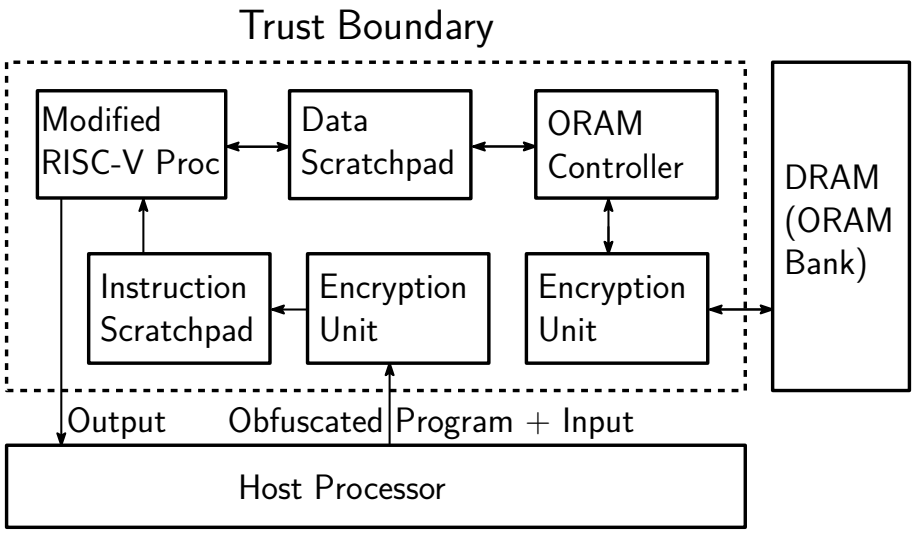


Figure 5.7: HOP Architecture

The final architecture of HOP (with the optimizations from Section 5.2) is shown in Figure 5.7. We now describe implementation-specific details for each major component.

5.4.1 Modified RISC-V Processor and Scratchpad

We built HOP with a RISC-V processor which implements a single stage 32bit integer base user-level ISA developed at UC Berkeley [33]. A RISC-V C cross-compiler is used to compile C programs to be run on the processor. The RISC-V processor is modified to include a 16 KB instruction scratchpad and a 512 KB data scratchpad (Section 5.2.4). The RISC-V processor and the compiler are modified accordingly to accommodate the new scratchpad load/unload instructions

(described below). While HOP uses a single stage RISC-V processor, our system does not preclude additional hardware optimizations in commodity processors such as multi-issue, branch predictor, etc. Our only requirement to support such processor structures is the ability to calculate, for that program over all inputs, a suitably conservative maximum runtime T.

New scratchpad instructions. For our prototype, we load the scratchpad using a new instruction called **spld**, which is specified as follows:

$$\textbf{spld}\ addr, \#mem, spaddr$$

In particular, *addr* is used to specify the starting address of the memory that needs to be loaded in scratchpad. $\#mem$ is the number of memory locations to be loaded on the scratchpad starting at *addr* and *spaddr* is the location in scratchpad to store the loaded data. When the processor intercepts an **spld** instruction, it performs two operations: 1. It writes back the data stored in this scratchpad location to the appropriate address in main memory (ORAM). 2. It reads $\#mem$ memory locations starting at main memory address *addr* into scratchpad locations starting at *spaddr*. Of course, **spld**'s precise design is not fundamental: we need a way to load an on-chip memory such that it is still feasible to statically determine T.

Example scratchpad use.

Figure 5.8 shows an example scenario where **spld** is used. The program shows a part of the code used for decompressing data using the `bzip2` compression algorithm. The algorithm decompresses blocks of compressed data and outputs data of size CSIZE independently. Each block of data may be read and processed multiple

```
1: int decompress(char *chunk) {
2:       int compLen = 0;
3:      // initial processing
4:      burrowsWheeler(chunk, compLen);
5:      // more processing
6:      writeOutput(chunk);
7:       return compLen;
8: }
9: void main() {
10:     char *inp = readInput();
11:     for (i = 0; i < len(inp); i += len) {
12:         spld(inp + i, CSIZE, 0);
13:          len = decompress(inp + i);
14:     }
15: }
```

Figure 5.8: Example program using **spld**: `bzip2`

times during different steps of compression (run-length encoding, Burrows-Wheeler transform, etc.). Hence, each such block is loaded into the scratchpad (line 12) before processing. This ensures that every subsequent access to this data is served by the scratchpad instead of memory (thereby reducing expensive ORAM accesses). After decompressing the block, **spld** is executed for the next block of compressed data.

5.4.2 ORAM Controller

We use a hardware ORAM controller called 'Tiny ORAM' from [56, 57]. The ORAM controller implements an ORAM tree with 25 levels, having 4 blocks per bucket. Each block is 512 bits (64 Bytes) to match modern processor cache line size. This corresponds to a total memory of 4 GB. The ORAM controller uses a stash of size 128 blocks and an on-chip position map of 256 KB. For integrity and freshness, Tiny ORAM uses the PosMap MAC (PMMAC) scheme [56]. We note that PMMAC protects data integrity but does not achieve malicious security. We estimate the cost of malicious security using a hardware Merkle-tree on ORAM in Table 5.2. We disable the PosMap Lookaside Buffer (PLB) in Freecursive ORAM to avoid leakage through the total number of ORAM accesses.

5.4.3 Encryption Units

For all encryption units, we can use tinyaes from OpenCores [5]. The encryption units can communicate with the external DRAM (bandwidth of 64 Bytes/cycle)

Table 5.2: **Resource allocation and utilization of HOP on Xilinx Virtex V7485t FPGA.** For each row, first line indicates the estimate. % utilization is mentioned in parentheses. LUT: Slice LookUp Table, FFs: Flip-flops or slice registers, BRAM: Block RAM.

	LUT	FFs	LUT-Mem	BRAM
Total Estimate	169472	51870	81112	566.5
(% Utilization)	(55.8%)	(8.5%)	(62.0%)	(55.0%)
HOP Estimate	103462	39803	38725	437
(% Utilization)	(34.0%)	(6.6%)	(47.7%)	(42.4%)
(HOP− ORAM) Estimate	21626	6579	1	83
(% Utilization)	(7.1%)	(1.1%)	(∼0%)	(8.1%)
Estimate with Merkle tree	221041	81410	81126	566.5
(% Utilization)	(72.8%)	(13.4%)	(62.0%)	(55.0%)

as well as the host processor. Data is encrypted before writing to the DRAM. Similarly, all data read from the DRAM is decrypted first before processed by the ORAM controller. Another encryption unit can be used to decrypt the obfuscated program before loading it into the instruction scratchpad.

5.5 Evaluation

We now present a detailed evaluation of HOP for some commonly used programs, and compare HOP to prior work.

5.5.1 Methodology

We measure program execution time in processor cycles, and compare with our own baseline scheme (to show the effectiveness of our optimizations), an insecure processor as well as related prior work. For each program, we choose parameters so that our baseline scheme requires about 100 million cycles to execute. We also report processor idle time, the time spent on dummy arithmetic instructions and dummy memory accesses to adhere to an $A^N M$ schedule (Section 5.2.3).

For the programs we evaluate (except *bzip2*; c.f., Section 5.5.4), we calculate T manually. We remark that the average input completion time and worst case time are very similar for these programs. To find T for larger programs, one may use established techniques in determining worst case execution time (e.g., a tool from [142]).

In our prototype, evaluating an arithmetic instruction takes 1 cycle while reading/writing a word from the scratchpad takes 3 cycles. Given the parameters in Section 5.4.2, an ORAM access takes 3000 cycles. For our HOP configurations with a scratchpad, we require both scratchpad read/writes and arithmetic instructions to take 3 cycles in order to hide which is occurring. Following Section 5.2.3, we set $N = 3000$ when not using a scratchpad; with a scratchpad, we use $N = 1000$. For

our evaluation, we consider programs ranging from those with high locality (e.g., `bwt-rle`) to those that show no locality (e.g., `binsearch`).

5.5.2 Area Results

We synthesized, placed and routed (a slightly modified version of) HOP on a Xilinx Virtex V7485t FPGA for parameters described in Section 5.4. HOP operates at 79.3 MHz on this FPGA. The resource allocation and utilization figures are mentioned in Table 5.2. The first three rows represent the total estimate, estimate for HOP (i.e. excluding RISC-Vprocessor, and the scratchpad) and an estimate for HOP that does not account for ORAM. The last row shows the total overhead including an estimate for a Merkle tree scheme. Excluding the processor, scratchpad and ORAM, HOP consumes $< 9\%$ of the FPGA resources. We see that the total area overhead of HOP is small and can be built on a single FPGA chip.

5.5.3 Main Results

Figure 5.9 shows the execution time of HOP variants relative to an insecure processor. For each program, there are three bars shown. The first bar is for the baseline HOP scheme (i.e., Section 5.2.2 only); the second bar only uses an $A^N M$ schedule without a scratchpad (adds Section 5.2.3); and the third bar is our final scheme that uses a scratchpad and the $A^N M$ schedule (adds Section 5.2.4). All schemes are relative to an insecure processor that does not use ORAM or hide what instruction it is executing. We assume this processor uses a scratchpad that has the

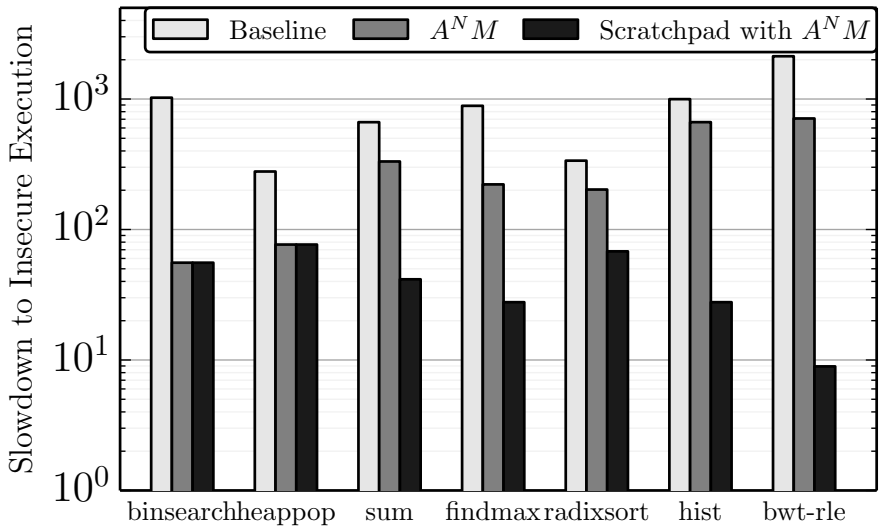


Figure 5.9: Execution time for HOP for different programs with (i) baseline scheme, (ii) $A^N M$ schedule and (iii) Scratchpad + $A^N M$.

same capacity as HOP in Section 5.4.1. The time required to insert the program and data is not shown.

Comparison of HOP variants. As can been seen in the figure, the $A^N M$ schedule without a scratchpad gives a 1.5× $\sim$ 18× improvement. Adhering to an $A^N M$ schedule requires some dummy arithmetic or memory instructions during which the processor is essentially idle. We observe that for our programs, the idle time ranges between 43% and 49.9% of the execution time, consistent with the claim in Section 5.2.3.

Effect of a scratchpad. The effect of a scratchpad largely depends on program locality. We thus classify programs in our evaluation into four classes:

1. Programs such as `binsearch, heappop` do not show locality. Thus, a scratchpad does not improve performance.

2. Programs such as `sum, findmax` stream (linear scan) over the input data. Given that an ORAM block is larger than a word size (512 bits vs 32 bits in our case), a scratchpad in these streaming applications can serve the next few (7 with our parameters) memory accesses after **spld**. A larger ORAM block size can slightly benefit these applications while severely penalize programs with no locality, and therefore is not a good trade-off.
3. Programs that maintain a small working set at all times will greatly benefit from a scratchpad. We evaluate one such program `bwt-rle`, which performs Burrows-Wheeler transform and run length encoding, and is used in compression algorithms.
4. Lastly, some programs are a mix of the above cases — some data structures can be entirely loaded into the scratchpad whereas some cannot (e.g. a Radix sort program).

Comparison to insecure processor. The remaining performance overhead of the optimized HOP (the third bar) comes from several sources. First, the performance of ORAM: The number of cycles to perform a memory access using ORAM is much higher than a regular DRAM. In HOP, an ORAM access is 40× more expensive than an insecure access. Second, dummy accesses to adhere to a schedule: As shown in Section 5.2.3, the performance overhead due to dummy accesses $\leq 2\times$. For programs such as `bwt-rle`, HOP has a slowdown as low as 8×. This is primarily due to the reduction in ORAM accesses by maintaining a small working set in the scratchpad.

5.5.4 Case Study: bzip2

To show readers how our system performs on a realistic and complex benchmark, we evaluate HOP on the open-source algorithm `bzip2` (re-written for a scratchpad, cf. Figure 5.8). We evaluate the decompression algorithm only, as the decompression algorithm's performance does not heavily depend on the input if one fixes the input size [1]. This allows us to run an average case input and use its performance to approximate the effect of running other inputs. To give a better sense for how the optimizations are impacted by different inputs, we don't terminate at a worst-case time T but rather terminate as soon as the program completes.

We run tests on two inputs, both highly compressible strings. For the first input, HOP achieves 106× speedup over the baseline scheme and 17× slowdown over the insecure version. For the second input, HOP achieves 234× speedup over the baseline and 8× slowdown over the insecure version. Thus, the gains and slowdowns we see from the prior studies extend to this more sophisticated benchmark.

5.5.5 Comparison with GhostRider [102]

Recall from Section 5.1 that GhostRider protects input data to the program *but not the program*. Since our privacy guarantee is strictly greater than GhostRider, we now compare to that work to show the cost of extra security. Note: we compare to the GhostRider compiler and not the implementation in [102] which uses a different parameterization for the ORAM scheme. This comparison shows the additional cost that is incurred by HOP to hide the program. We don't show the

full comparison for lack of space, but point out the following extreme points: For programs with unpredictable access patterns (`binsearch, heappop`), GhostRider outperforms HOP by $\sim 2\times$. HOP's additional overhead is from executing dummy instructions to adhere to a particular schedule. For programs with predictable access patterns (`sum, findmax, hist`), GhostRider's performance is similar to that of an insecure processor.

5.5.6 Time for Context Switch

Since it was not required for our performance evaluation, we have not yet implemented context switching (Section 5.2.5) in our prototype. Recall, context switching means the receiver interrupts the processor, which encrypts and writes out all the processor state (including CPU state, instruction scratchpad, data scratchpad, ORAM position map and stash) to DRAM. We estimate the time of a context switch as follows. The total amount of data stored by our token is $\sim$ 800 KB (Section 5.4). Assuming a DRAM bandwidth of 10 GB/s and a matching encryption bandwidth, it would take $\sim 160\mu$s to perform a context switch to run another program. Note that this assumes all data for a swapped-out context is stored in DRAM (i.e., the ORAM data already in the DRAM need not be moved). If it must be swapped out to disk because the DRAM must make room for the new context, the context switch time grows proportional to the ORAM size.

5.6 Conclusion

This chapter makes two main contributions. First, we construct an optimized hardware architecture - called HOP - for running obfuscated RAM programs. We give a matching theoretic model for our optimized architecture and prove it secure. A by-product of our analysis shows the first obfuscation scheme for RAM programs using 'stateless' tokens. Second, we present a complete implementation of our optimized architecture and evaluate it on real-world programs. The complete design would require an estimated 72% the area of a V7485t Field Programmable Gate Array (FPGA) chip. Run on a variety of benchmarks, HOP achieves an average overhead of $8\times \sim 76\times$ relative to an insecure system. To the best of our knowledge, this effort represents the *first* implementation of a provably secure VBB obfuscation scheme in any model under any assumptions.

Chapter 6: GraphSC: Parallel Secure Computation Made Easy

Through their interactions with many web services, and numerous apps, users leave behind a dizzying array of data across the web ecosystem. The privacy threats due to the creation and spread of personal data are by now well known. The proliferation of data across the ecosystem is so complex and daunting to users, that encrypting data at all times appears as an attractive approach to privacy. However, this hinders all benefits derived from mining user data, both by online companies and the society at large (e.g., through opinion statistics, ad campaigns, road traffic and disease monitoring, etc). Secure computation allows two or more parties to evaluate any desirable polynomial-time function over their private data, while revealing only the answer and nothing else about each party's data. Although it was first proposed about three decades ago [150], it is only in the last few years that the research community has made enormous progress at improving the efficiency of secure computation [82, 93, 103, 119, 120]. As such, secure computation offers a better alternative, as it enables data mining while simultaneously protecting user privacy.

The need to analyze data on a massive scale has led to modern architectures that support parallelism, as well as higher level programming abstractions to

take advantage of the underlying architecture. Examples include MapReduce [49], Pregel [108], GraphLab [105], and Spark [151]. These provide software developers interfaces handling inputs and parallel data-flow in a relatively intuitive and expressive way. These programming paradigms are also extremely powerful, encompassing a broad class of machine learning, data mining and graph algorithms. While these paradigms enable developers to efficiently write and execute complex parallel tasks on very large datasets, securely computing on private data is not an objective for any of these frameworks. Our goal is to bring secure computation to such frameworks in a way that does not require programmers to have cryptographic expertise.

The benefits of integrating secure computation into such frameworks are numerous. The potential to carry out data-analysis tasks while simultaneously not leaking private data could change the privacy landscape. Consider a few examples. A very common use of MapReduce is to compute histograms that summarize data. This has been done for all kinds of data, such as counting word frequencies in documents, or summarizing online browsing behavior, or YouTube viewing behavior to name just a few. Another common use of the graph parallelization models (e.g., GraphLab) is to compute influence in a social graph through, for example, the PageRank algorithm. Today, joint influence over multiple social graphs belonging to different companies (such as Facebook and LinkedIn), cannot be computed because companies do not share such data. For this to be feasible, the companies need to be able to perform an oblivious secure computation on their joint graph in a highly efficient way that supports their massive datasets and completes in a reasonable time. A privacy requirement for such an application is to ensure that the graph

structure, and any associated data, is not leaked; the performance requirements for scalability and efficiency demand the application to be highly parallelizable. A third example application is recommender systems based on the matrix factorization (MF) algorithm. It was shown that it is possible to carry out secure MF, enabling users to receive recommendations without ever revealing records of past behavior (e.g., movies watched or rated) in the clear to the recommender system [119]. But this previous work did not gracefully incorporate parallelism to scale to millions of records.

This chapter addresses the following key question: *can we build an efficient secure computation framework that uses familiar parallelization programming paradigms?* By creating such a framework, we can bring secure computation to the practical realm for modern massive datasets. Furthermore, we can make it accessible to a wide audience of developers that are already familiar with modern parallel programming paradigms, and are not necessarily cryptography experts.

One naïve approach to obtain high parallelization is the following: (a) programmers write programs using a programming language specifically designed for (sequential) secure computation such as the SCVM source language [103] or the ObliVM source language [104]; (b) apply an existing program-to-circuits compiler[1]; and (c) exploit parallelism that occurs at the circuit level – in particular, all the gates within the same layer (circuit depth) can be evaluated in parallel. Henceforth,

[1]RAM-model compilers such as SCVM [103] and ObliVM [104] effectively compile a program to a sequence of circuits as well. In particular, dynamic memory accesses are compiled into ORAM circuits.

we use the term circuit-level parallelism to refer to this baseline approach.

While intuitive, this baseline approach is far from ideal. The circuit derived by a sequential program-to-circuits compiler can also be sequential in nature, and many opportunities to extract parallelism may remain undiscovered. We know from experience, in the insecure environment, that generally trying to produce parallel algorithms requires careful attention. Two approaches have been intensely pursued (for the case of non-secure computation): (a) *Design of parallel algorithms:* an entire field of research has focused on designing parallel versions of specific algorithms that seek to express computation tasks with shallow depth and without significantly increasing the total amount of work in comparison with the sequential setting; and (b) *Programming abstractions for parallel computation:* the alternative to finding point solutions for particular problems, is to develop programming frameworks that help programmers to easily extract and express parallelism. The frameworks mentioned above fall into this category. These two approaches can also be followed for solutions in secure computation; examples of point solutions include [119, 120]. In this work, we follow the second approach to enable parallel oblivious versions for a range of data mining algorithms.

There are two fundamental challenges to solve our problem. The first is the need to provide a solution that is data oblivious, in order to prevent any information leakage and to prevent unnecessary circuit explosion. The second is that of migrating secure computation models to the parallel environment in an efficient way. Because our solution focuses on graph-based parallel algorithms, we need to ensure that the graph structure itself is not revealed.

In this chapter, we focus on 2-party computation in the semi-honest model. Our two parties could be two non-colluding cloud providers (such as Google and Amazon) where both parties have parallel computing architectures (multiple machines with multiple cores). In this case, the data is outsourced to the cloud providers, and within each cloud the secret data could be distributed across multiple machines. In a second scenario, a single cloud provider splits up the data to achieve resilience against insider attacks or compromise. To realize these, we make the following novel contributions.

Our Contributions

We design and implement a parallel secure computation framework called GraphSC. With GraphSC, developers can write programs using programming abstractions similar to Pregel and GraphLab [68, 105, 108]. GraphSC executes the program with a parallel secure computation backend. Adopting this programming abstraction allows GraphSC to naturally support a broad class of data mining algorithms.

New parallel oblivious algorithms. To the best of our knowledge, our work is the *first to design non-trivial parallel oblivious algorithms that outperform generic Oblivious Parallel RAM* [28, 35, 38]. The OPRAM constructions are of a theoretical nature, with computational costs that would be prohibitive in a practical implementation. Analogously, in the sequential literature, a line of research focuses on designing efficient oblivious algorithms that outperform generic ORAM [23, 54, 72, 152].

Many of these works focus on specific functionalities of interest. However, such a one-at-a-time approach is unlikely to gain traction in practice, since real-life programmers likely do not possess the expertise to design customized oblivious algorithms for each task at hand; moreover, they *should not* be entrusted to carry out cryptographic design tasks.

While we focus on designing efficient parallel oblivious algorithms, *we take a departure from such a one-at-a-time design approach. Specifically, we design parallel oblivious algorithms for GraphSC's programming abstractions, which in turn captures a broad class of interesting data mining and machine learning tasks.* We will demonstrate this capability for four such algorithms. Moreover, our parallel oblivious algorithms can also be made accessible to non-expert programmers. Our parallel oblivious algorithms achieve logarithmic total-work and depth blowup in comparison with the poly-logarithmic blowup of generic OPRAM [28]. In particular, *for a graph containing* $|\mathsf{E}|$ *edges and* $|\mathsf{V}|$ *vertices, GraphSC just has an overhead of* $O(\log |\mathsf{V}|)$ *when compared with the parallel insecure version.*

System implementation. ObliVM-GC (`http://www.oblivm.com`) is a programming language that allows a programmer to write a program that can be compiled into a garbled circuit, so that the programmer need not worry about the underlying cryptographic framework. In this chapter, we architect and implement GraphSC, a parallel secure computation framework that supports graph-parallel programming abstractions resembling GraphLab [105]. Such graph-parallel abstractions are expressive and easy-to-program, and have been a popular approach for developing parallel data mining and machine learning algorithms. GraphSC is suitable for both

multi-core and cluster-based computing architectures. The source code of GraphSC is available at `http://www.oblivm.com`.

Evaluation. To evaluate the performance of our design, we implement four classic data analysis algorithms: (1) a histogram function assuming an underlying MapReduce paradigm; (2) PageRank for large graphs; and two versions of matrix factorization (MF), namely, (3) MF using gradient descent, and (4) MF using alternating least squares (ALS). We study numerous metrics, such as the effect of parallelism, i.e., the change in execution time with increasing number of processors, as well as the amount of communication between processors. We deploy our experiments in a realistic setting, both on a controlled testbed and on Amazon Web Services (AWS). We show that we can achieve practical speeds for our 4 example algorithms, and that the performance scales gracefully with input size and the number of processors. We achieve these gains with minimal communication overhead, and an insignificant impact on accuracy. For example, we were able to run matrix factorization on MovieLens dataset (6000 users, 4000 movies) consisting of 1 million ratings in less than 13 hours on a small 7-machine cluster. As far as we know, this is the first application of a complicated secure computation algorithm on a large real-world dataset; previous work [119] managed to complete a similar task on only 17K ratings, with no ability to scale beyond a single machine. This demonstrates that our work can bring secure computation into the realm of practical large-scale parallel applications.

The rest of the chapter is structured as follows. Following the related work, in Section 6.2 we present GraphSC, our framework for parallel computation on

large-scale graphs. In Section 6.3 we detail how GraphSC can support parallel *data oblivious* algorithms. Then, in Section 6.4, we discuss how such parallel oblivious algorithms can be converted into parallel *secure* algorithms. Section 6.5 discusses the implementation of GraphSC and detailed evaluation of its performance on several real-world applications. We conclude the chapter in Section 6.6.

Model and Terminology

Our main deployment scenario is the following parallel secure two-party computation setting. The two parties are two non-colluding, semi-honest cloud providers (potentially executing some outsourced computation). Since we adopt Yao's Garbled Circuits [149], one cloud provider acts as the garbler, and the other acts as the evaluator. Each cloud provider can have multiple processors performing the garbling or evaluation.

In this chapter, we focus on a specific class of graph computations. For a graph with V vertices and E edges, we assume that the size information $|\mathsf{V}| + |\mathsf{E}|$ is public. To keep terminology simple, our main algorithms in Section 6.3.3 refer to *parallel oblivious algorithms* – assuming a model where multiple processors have a shared random-access memory. It turns out that once we derive parallel oblivious algorithms, it is easy to translate them into parallel secure computation protocols. Section 6.4 and Figure 6.4 later in the chapter will elaborate on the details of our models and terminology.

6.1 Related Work

Secure computation has been studied for decades, starting from theory [73, 87, 132, 133, 149] to implementations [24, 79, 81, 82, 92, 93, 103, 109, 119, 125, 155].

Parallel secure computation frameworks. Most existing implementations are sequential. However, parallel secure computation has naturally attracted attention due to the wide adoption of multi-core processors and cloud-based compute clusters. Note that in Yao's Garbled Circuits [149], the garbler's garbling operations are trivially parallelizable. However, evaluation of the garbled circuit must be done layer by layer, and therefore, the depth of the circuit(s) determine the degree to which evaluation can be parallelized.

Most research on parallel secure computation just exploits the natural parallelism within each circuit or between circuits (e.g., for performing cut-and-choose in the malicious model). For example, Husted et al. [83] propose using a GPU-based backend for parallelizing garbled circuit generation and evaluation. Their work exploits the natural circuit-level parallelism – however, in cases where the program is inherently sequential (e.g., a narrow and deep circuit), their technique will not be able to exploit massive degrees of parallelism for evaluation. Assuming the computation on a single vertex/edge is a low-depth circuit, our design ensures that GraphSC primitives are implemented as low-depth circuits. Though our design currently works on a multi-core processor architecture or a compute cluster, the same programming abstraction and parallel oblivious algorithms can be directly ported to a GPU-based backend; our work thus is complementary to Husted et al. [83].

Kreuter et al. [93] exploit parallelism to parallelize cut-and-choose in malicious-model secure computation. In particular, cut-and-choose techniques require the garbled evaluation of multiple circuits, such that one can assign each circuit to a different processor. In comparison, we focus on parallelization in the semi-honest model. If we were to move to the malicious model, we would also benefit from the additional parallelism natural in cut-and-choose. Our approach is closest to, and inspired by, the privacy-preserving matrix factorization (MF) framework by Nikolaenko et al. [119] that implements gradient-descent MF as a garbled circuit. As in our design, the authors rely on oblivious sorting that, as they note, is parallelizable. Though Nikolaenko et al. exploit this to parallelize parts of their MF computation, their overall design is not parallelizable: it results in a $\Omega(|V| + |E|)$-depth circuit, containing serial passes over the data. In fact, the algorithm in [119] is equivalent to the serial algorithm presented in Algorithm 2, restricted to MF. Crucially, beyond extending our implementation to any algorithm expressed by GraphSC (not just gradient-descent MF), our design also parallelizes these serial passes (cf. Figure 6.3), leading to a circuit of logarithmic depth. Finally, as discussed in Section 6.5, the garbled circuit implementation in [119] can only be run on a single machine, contrary to GraphSC.

Automated frameworks for sequential secure computation. In the sequential setting, numerous automated frameworks for secure computation have been explored, some of which [81, 155] build on (a subset of) a standard language such as C; others define customized languages [24, 79, 92, 103]. As mentioned earlier, the circuits generated by these sequential compilers may not necessarily have low depth.

For general-purpose secure computation backends, several protocols have been investigated and implemented, including those based on garbled circuits [149, 150], GMW [66], somewhat or fully homomorphic encryption [60], and others [17, 46]. In this chapter, we focus on a garbled circuits backend for the semi-honest setting, but our framework and programming abstractions can readily be extended to other backends as well.

ORAM and oblivious algorithms. While ORAMs obliviously simulate any RAM program, oblivious algorithms have also been studied to obliviously simulate specific algorithms [23, 54, 72, 113, 145, 152]. These solutions provide point solutions that outperform ORAMs. As recent works point out [103], ORAM and oblivious algorithms are key to transforming programs into compact circuits[2] – and circuits represent the computation model for almost all known secure computation protocols. Broadly speaking, any data oblivious algorithm admits an efficient circuit implementation whose size is proportional to the algorithm's runtime. Generic RAM programs can be compiled into an oblivious counterpart with polylogarithmic blowup [67, 70, 96, 134, 143].

In a similar manner, Oblivious Parallel RAMs (OPRAM) [28,35,38], essentially transform PRAM programs into low-depth circuits, also incurring a polylogarithmic blowup. As mentioned earlier, these works are more of a theoretical nature and expensive in practice. In comparison, our work proposes efficient oblivious algorithms

[2]For secure computation, a program is translated into a sequence of circuits whose inputs can be oblivious memory accesses. Note that this is different from transforming a program into a single circuit – for the latter, the best known asymptotical result incurs quadratic overhead [129].

for a restricted (but sufficiently broad) class of PRAM algorithms, as captured by our GraphSC programming abstractions.

Parallel programming paradigms. The past decade has given rise to parallelization techniques that are suitable to cheap modern hardware architecture. MapReduce [49] is a seminal work that presented a simple programming model for processing massive datasets on large cluster of commodity computers. This model resulted on a plethora of system-level implementations [135] and improvements [151]. A second advancement was made with Pregel [108], a simple programming model for developing efficient parallel algorithms on large-scale graphs. This also resulted in several implementations, including GraphLab [68, 105] and Giraph [13]. The simplicity of interfaces exposed by these paradigms (like the scatter, gather, and apply operations of Pregel) led to their widespread adoption, as well as to the proliferation of algorithms implemented in these frameworks. We introduce similar programming paradigms to secure computation, in the hope that it can revolutionize the field like it did to non-secure parallel programming models, thus making secure computation easily accessible to non-experts, and easily deployable over large, cheap clusters.

6.2 GraphSC

In this section, we formally describe GraphSC, our framework for parallel computation. GraphSC is inspired by the scatter-gather operations in GraphLab and Pregel. Several important parallel data mining and machine learning algorithms can be cast in this framework (some of these are discussed in Section 6.5.1); a brief

example (namely, the PageRank algorithm) can also be found below. We conclude this section by highlighting the challenges behind implementing GraphSC in a secure fashion.

6.2.1 Programming Abstraction

Data-augmented graphs. The GraphSC framework operates on data-augmented directed graphs. A data-augmented directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{D})$ consists of a directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$, as well as user-defined data on each vertex and each edge denoted $\mathsf{D} \in (\{0,1\}^*)^{|\mathsf{V}|+|\mathsf{E}|}$. We use the notation $\mathsf{v.data} \in \{0,1\}^*$ and $\mathsf{e.data} \in \{0,1\}^*$ to denote the data associated with a vertex $\mathsf{v} \in \mathsf{V}$ and an edge $\mathsf{e} \in \mathsf{E}$ respectively.

Programming abstractions. GraphSC follows the Pregel/GraphLab programming paradigm, allowing computations that are "graph-parallel" in nature, i.e., each vertex performs computations on its own data as well as data collected from its neighbors. In broad terms, this is achieved through the following three primitives, which can be thought of as interfaces exposed by the GraphSC abstraction:

1. Scatter: A vertex propagates data to its adjacent edges and updates the edge's data. More specifically, Scatter takes a user-defined function $f_S : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, and a bit $b \in \{\text{“}\mathsf{in}\text{”}, \text{“}\mathsf{out}\text{”}\}$, and updates each directed edge $\mathsf{e} = (\mathsf{u}, \mathsf{v})$ as follows:

$$\mathsf{e.data} := \begin{cases} f_S(\mathsf{e.data}, \mathsf{v.data}) & \text{if } b = \text{“}\mathsf{in}\text{”}, \\ f_S(\mathsf{e.data}, \mathsf{u.data}) & \text{if } b = \text{“}\mathsf{out}\text{”}. \end{cases}$$

   Note that the bit b indicates whether the update operation is to occur over

incoming or outgoing edges of each vertex.

2. Gather: Through this operation, a vertex aggregates the data from adjacent edges and updates its own data. More specifically, Gather takes as input a binary aggregation operator $\oplus : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and a bit $b \in \{$ "in", "out" $\}$ and updates the data on each vertex $\mathsf{v} \in \mathsf{V}$ as follows:

$$\mathsf{v.data} := \begin{cases} \mathsf{v.data} \;||\; \bigoplus_{\forall \mathsf{e} \in \mathsf{neigh(v,in)},} \mathsf{e.data} & \text{if } b = \text{"in"}, \\ \mathsf{v.data} \;||\; \bigoplus_{\forall \mathsf{e} \in \mathsf{neigh(v,out)},} \mathsf{e.data} & \text{if } b = \text{"out"}, \end{cases}$$

where $||$ indicates concatenation, and $\bigoplus$ is the iterated binary operation defined by $\oplus$, and $\mathsf{neigh(v,in)}$ and $\mathsf{neigh(v,out)}$ represent the incoming and outgoing edges of v respectively. Hence, at the conclusion of the operation, the vertex stores both its previous value, as well as the output of the aggregation through $\oplus$.

3. Apply: Vertices perform some local computation on their data. More specifically, Apply takes a user-defined function $f_A : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, and updates every vertex's data as follows:

$$\mathsf{v.data} := f_A(\mathsf{v.data}).$$

A program abiding by the GraphSC abstraction can thus make arbitrary calls to such Scatter, Gather and Apply operations. Beyond determining this sequence, each invocation of Scatter, Gather, and Apply must also supply the corresponding user-defined functions f_S, f_A, and aggregation operator $\oplus$. Note that the graph structure G does not change during the execution of any of the three GraphSC primitives.

Throughout our analysis, we assume the time complexity of f_S, f_A, and the binary operator $\oplus$ (applied to only 2 arguments) is constant, i.e., it does not depend on the size of G. This is true when, e.g., both vertex and edge data take values in a finite subset of $\{0,1\}^*$, which is the case for all applications we consider[3].

Requirements for the aggregation operator $\oplus$. During the Gather operation, a vertex aggregates data from multiple adjacent edges through a binary aggregation operator $\oplus$. GraphSC requires that this aggregation operator is *commutative* and *associative*, i.e.,

- **Commutative**: For any $a, b \in \mathcal{D}$, $a \oplus b = b \oplus a$.
- **Associative**: For any $a, b, c \in \mathcal{D}$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

Here, $\mathcal{D}$ represents the set of values that can be assumed by vertex and edge data in the graph. Roughly speaking, commutativity and associativity guarantee that the result of the aggregation is insensitive to the ordering of the edges.

6.2.2 Expressiveness

At a high level, GraphSC borrows its structure from Pregel/GraphLab [68,105, 108], which is also defined by the three conceptual primitives called Gather, Apply and Scatter. There are however a few differences that are not included in GraphSC,

[3]Note that, due to the concatenation operation $||$, the memory size of the data at a vertex can in theory increase after repeated consecutive Gather operations. However, in the Pregel/GraphLab paradigm, a Gather is always followed by an Apply, that merges the aggregated edge data with the vertex data through an appropriate user-defined merge operation f_A. Thus, after each iteration completes the size of vertex data remains constant.

as they break obliviousness. For instance, Pregel allows arbitrary message exchanges between vertices, which is not supported by GraphSC. Pregel also supports modification of the graph structure during computation, whereas GraphSC does not allow such modifications. Finally, GraphLab supports asynchronous parallel computation of the primitives, whereas GraphSC, and its data oblivious implementation we describe in Section 6.3, are both synchronous.

Except for these differences that are necessary to maintain obliviousness, all other programs that can be expressed in Pregel/GraphLab can be expressed in GraphSC. GraphSC encompasses classic graph algorithms like Bellman-Ford, bipartite matching, connected component identification, graph coloring, etc., as well as several important data mining and machine learning operations including PageRank [123], matrix factorization using gradient descent and alternating least squares [91], training neural networks through back propagation [128] or parallel empirical risk minimization through the alternating direction method of multipliers (ADMM) [27]. We review some of these examples in more detail in Section 6.5.1.

6.2.3 Example: PageRank

Let us try to understand these primitives using the PageRank algorithm [123] as an example. Recall that PageRank computes a ranking score PR for each vertex u of a graph G through a repeated iteration of the following assignment:

$$\forall \mathsf{u} \in \mathsf{V}, \quad \mathsf{PR}(\mathsf{u}) = \frac{0.15}{|\mathsf{V}|} + 0.85 \times \sum_{(\mathsf{v},\mathsf{u}) \in \mathsf{E}} \frac{\mathsf{PR}(\mathsf{v})}{\mathsf{L}(\mathsf{v})},$$

Algorithm 1 PageRank example

1: **function** computePageRank($\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{D})$)

2: $f_S(\mathsf{e.data}, \mathsf{u.data})$: $\mathsf{e.data} := \frac{\mathsf{u.data.PR}}{\mathsf{u.data.L}}$

3: $\oplus(\mathsf{e}_1\mathsf{.data}, \mathsf{e}_2\mathsf{.data})$: $\mathsf{e}_1\mathsf{.data} + \mathsf{e}_2\mathsf{.data}$

4: $f_A(\mathsf{v.data})$: $\mathsf{v.data.PR} := \frac{0.15}{|\mathsf{V}|} + 0.85 \times \mathsf{v.data.agg}$

5: **for** $i := 1$ **to** K **do** *// K: number of iterations until convergence*

6: Scatter(G, f_S, "out")

7: Gather($\mathsf{G}, \oplus$, "in")

8: Apply(G, f_A)

9: // Every vertex v stores its PageRank PR

where $\mathsf{L}(\mathsf{v})$ is the number of outgoing edges. Initially, all vertices are assigned a PageRank of $\frac{1}{|\mathsf{V}|}$.

PageRank can be expressed in GraphSC as shown in Algorithm 1. The data of every vertex v comprises two real values, one for the PageRank (PR) of the vertex and the other for the number of its outgoing edges ($\mathsf{L}(\mathsf{v})$). The data of every edge $\mathsf{e} = (\mathsf{u}, \mathsf{v})$ comprises a single real value corresponding to the weighted contribution of PageRank of the outgoing vertex u.

For simplicity, we assume that each vertex v has pre-computed and stored $\mathsf{L}(\mathsf{v})$ at the beginning of the algorithm's execution. The algorithm then consists of several iterations, each evoking a Scatter, Gather and Apply operation. The Scatter operation updates the edge data $\mathsf{e} = (\mathsf{u}, \mathsf{v})$ by the weighted PageRank of the

outgoing vertex u, i.e., $b =$ "out" and

$$f_S(\mathsf{e.data}, \mathsf{u.data}) : \mathsf{e.data} := \frac{\mathsf{u.data.PR}}{\mathsf{u.data.L}}.$$

In the Gather operation, every vertex v adds up the weighted PageRank over incoming edges e(u, v) and concatenates the result with the existing vertex data, by storing it in the variable v.data.agg. That is, $b =$ "in", and $\oplus$ is given by

$$\oplus(\mathsf{e}_1.\mathsf{data}, \mathsf{e}_2.\mathsf{data}) : \mathsf{e}_1.\mathsf{data} + \mathsf{e}_2.\mathsf{data}.$$

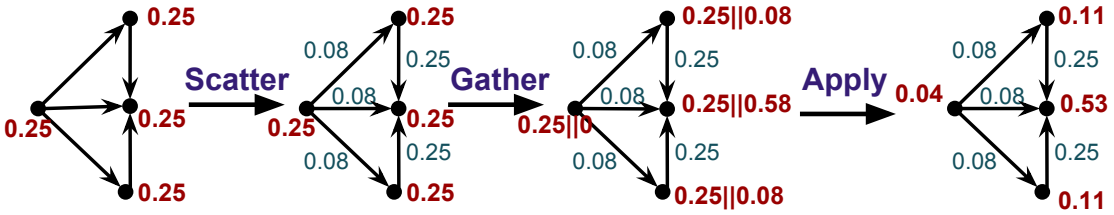


Figure 6.1: **One iteration of PageRank computation.** 1. Every page starts with PR = 0.25. 2. During Scatter, outgoing edges are updated with the weighted PageRank of vertices. 3. Vertices then aggregate the data on incoming edges in a Gather operation and store it along with their own data. 4. Finally, vertices update their PageRank in an Apply operation.

The Apply operation computes the new PageRank of vertex v using v.data.agg.

$$f_A(\mathsf{v.data}) : \mathsf{v.data.PR} := \frac{0.15}{|\mathsf{V}|} + 0.85 \times \mathsf{v.data.agg}.$$

An example iteration is shown in Figure 6.1.

6.2.4 Parallelization and Challenges in Secure Implementation

Under our standing assumption that f_S, f_A, and $\oplus$ have $O(1)$ time complexity, all three primitives are linear in the input, i.e., can be computed in $O(|\mathsf{V}| + |\mathsf{E}|)$ time. Moreover, like Pregel/GraphLab operations, Scatter, Gather and Apply can be easily parallelized, by assigning each vertex in graph G to a different processor. Each vertex also maintains a list of all incoming edges and outgoing edges, along with their associated data. Scatter operations involve transmissions: e.g., in a Scatter "out" operation, a vertex sends its data to all its outgoing neighbors, who update their corresponding incoming edges. Gather operations on the other hand are local: e.g., in a Gather "in" operation, a vertex simply aggregates the data in its incoming edges and appends it to its own data. Both Scatter and Gather operations can thus be executed in parallel across different processors storing the vertices. Finally, in such a configuration, Apply operations are also trivially parallelizable across vertices. Note that, in the presence of $P < |\mathsf{V}|$ processors, to avoid a single overloaded processor becoming a bottleneck, the partitioning of the graph should balance computation and communication across processors.

In this chapter, we wish to design a secure computation framework implementing GraphSC operations in a privacy-preserving fashion, while maintaining its parallelizability. In particular, our design should be such that, only the final output of the program is revealed; the input, i.e., the directed data-augmented graph $\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{D})$ should *not* be leaked during the execution of the program. We note that there are several applications in which hiding the data *as well as* the graph structure

of G is important. For example, in PageRank, the entire input is described by the graph structure G. As noted in [119], in the case of matrix factorization, the graph structure leaks which items a user has rated, which can again be very revealing. To highlight the difficulties that arise in implementing GraphSC in a secure fashion, we note that naïve parallelization, as described above, leaks a lot of information. In particular:

1. The amount of data stored by vertices, based on the above partitioning of the graph, reveals information about its neighborhood.
2. The number of times a vertex is accessed during a scatter phase reveals the number of outgoing neighbors.
3. Finally, the neighbors with which each vertex communicates during a Scatter phase reveals the entire graph G.

6.3 GraphSC Primitives as Efficient Parallel Oblivious Algorithms

In this section, we discuss how the three primitives exposed by the GraphSC abstraction can be expressed as *parallel data oblivious* algorithms. A parallel oblivious algorithm can be converted to a parallel secure algorithm using standard techniques; we describe such a conversion in more detail in Section 6.4, focusing here on data-obliviousness and parallelizability.

6.3.1 Parallel Oblivious Algorithms: Definitions and Metrics

A parallel algorithm $\mathsf{ALG} \in \mathcal{ALG}$ is said to be *perfectly oblivious*, iff for any two inputs inp_0 and inp_1 to the algorithm such that $|\mathsf{inp}_0| = |\mathsf{inp}_1|$ and such that $\mathcal{ALG}$ is known to the adversary, it holds that

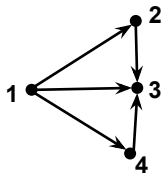
$$\mathsf{Addresses}[\mathsf{ALG}](\mathsf{inp}_0) \equiv \mathsf{Addresses}[\mathsf{ALG}](\mathsf{inp}_0)$$

In this chapter, our parallel oblivious algorithms are all deterministic and thus, the traces are identical. Also, our inputs are data-augmented graphs. Thus, for any two graphs, $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{D})$ and $\mathsf{G}' = (\mathsf{V}', \mathsf{E}', \mathsf{D}')$ such that $|\mathsf{V}| + |\mathsf{E}| = |\mathsf{V}'| + |\mathsf{E}'|$ and $|\mathsf{d}| = |\mathsf{d}'|$ for $\mathsf{d} \in \mathsf{D}$ and $\mathsf{d}' \in \mathsf{D}'$, we require that,

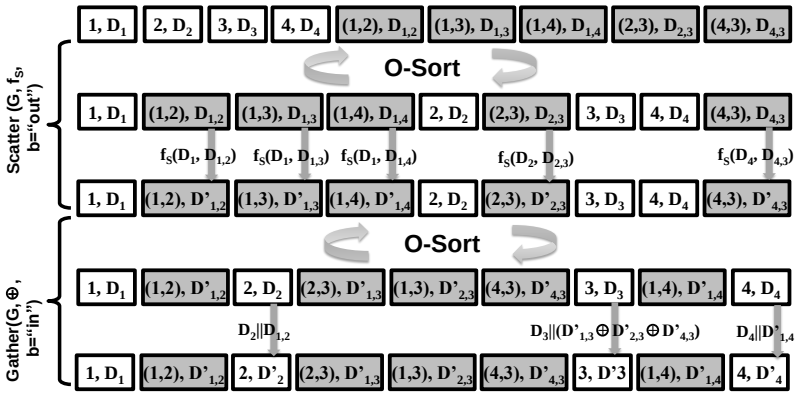
$$\mathsf{Addresses}[\mathsf{ALG}](\mathsf{G}) \equiv \mathsf{Addresses}[\mathsf{ALG}](\mathsf{G}')$$

Note that, by the above definition, a parallel oblivious algorithm *hides both the graph structure and the data on the graph's vertices and edges*. Only the "size" of the graph $|\mathsf{V}| + |\mathsf{E}|$ is revealed. Moreover, if such an algorithm is represented as a circuit of depth $\Theta(T)$, then it comprises of $\Theta(T)$ layers and each such layer represents the state of the shared memory at time t.

For our parallel oblivious algorithms, we are interested in the total work blowup and the depth blowup. The total work of a parallel oblivious algorithm may increase due to two reasons – First, due to the *cost of parallelism*: the most efficient (insecure) parallel algorithm may incur a blowup in terms of total work. Second, due to the *cost of obliviousness*: requiring that the algorithm is oblivious may also incur additional blowup in total work. We refer the reader to Section 2.2



(a) Graph G.



(b) Transformations of list representing graph G.

Figure 6.2: **Oblivious Scatter and Gather on a single processor.** We apply a Scatter followed by a Gather. *Scatter*: Graph tuples are sorted so that edges are grouped together after the outgoing vertex. e.g. $D_{1,2}, D_{1,3}, D_{1,4}$ are grouped after D_1. Then, in a single pass, all edges are updated. e.g. $D_{1,3}$ is updated as $f_S(D_1, D_{1,3})$. *Gather*: Graph tuples are sorted so that edges are grouped together before the incoming vertex. e.g. $D'_{1,3}, D'_{2,3}, D'_{4,3}$ are grouped before D_3. Then, in a single pass, all vertices compute the aggregate. e.g. $D'_3 = D_3||D'_{1,3} \oplus D'_{2,3} \oplus D'_{4,3}$.

in Chapter 2 for the specific definitions of total work blowup and depth blowup.

6.3.2 Single-Processor Oblivious Algorithm

Before presenting our fully-parallel solution, we describe how to implement each of the three primitives in a data-oblivious way on a single processor (i.e., when $P = 1$). One key challenge is how to *hide the graph structure* G during computation.

Alternative graph representation: Our oblivious algorithms require an alternative representation of graphs, that does not differentiate between edges and vertices.

Both vertices and edges are represented as tuples of the form $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$. In particular, each vertex u is represented by the tuple: $\langle \mathsf{u}, \mathsf{u}, 1, \mathsf{data} \rangle$; and each edge (u, v) is represented by the tuple: $\langle \mathsf{u}, \mathsf{v}, 0, \mathsf{data} \rangle$. We represent a graph as a *list of tuples*, i.e., $\mathsf{G} := (\mathsf{t}_i)_{i \in [|\mathsf{V}|+|\mathsf{E}|]}$ where each t_i is of the form $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$.

Algorithm description. We now describe the single-processor oblivious implementation of GraphSC primitives. The formal description of the implementation is provided in Algorithm 2. We also provide an example of the Scatter and Gather operations in Figure 6.2b, for a very simple graph shown in Figure 6.2a.

Apply. The Apply operation is straightforward to make oblivious under our new graph representation. Essentially, we make a linear scan over the list G. During this scan, we apply the function f_A to each vertex tuple in the list, and a dummy operation to each edge tuple.

Scatter. Without loss of generality, we use $b =$ "out" as an example. The algorithm for $b =$ "in" is similar. The Scatter operation then proceeds in two steps, illustrated in the first three lines of Figure 6.2b.

Step 1: Oblivious sort: First, perform an oblivious sort on G, so that tuples with the same source vertex are grouped together. Moreover, each vertex should appear before all the edges originating from that vertex.

Step 2: Propagate: Next, in a single linear scan, update the value of each edge with the nearest preceding vertex by applying the f_S function.

Gather. Again, without loss of generality, we will use $b =$ "in" as an example. The algorithm for $b =$ "out" is similar. Gather proceeds in a fashion similar to

Algorithm 2 Oblivious GraphSC on a Single Processor

G: list of tuples $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$, $N = |\mathsf{V}| + |\mathsf{E}|$

1: **function** Scatter(G, f_S, $b =$ "out")

/ $b =$ "in" is similar and omitted */*

2: **sort** G by $(\mathsf{u}, -\mathsf{isVertex})$

3: **for** $i := 1$ **to** N **do** */* Propagate */*

4: **if** G$[i]$.isVertex **then**

5: val := G$[i]$.data

6: **else**

7: G$[i]$.data := f_S(G$[i]$.data, val)

1: **function** Gather(G, $\oplus$, $b =$ "in")

/ $b =$ "out" is similar and omitted */*

2: **sort** G by $(\mathsf{v}, \mathsf{isVertex})$

3: **var** agg := $1_\oplus$ // *identity w.r.t.* $\oplus$

4: **for** $i := 1$ **to** N **do** */* Aggregate */*

5: **if** G$[i]$.isVertex **then**

6: G$[i]$.data := G$[i]$.data||agg

7: agg := $1_\oplus$

8: **else**

9: agg := agg $\oplus$ G$[i]$.data

1: **function** Apply(G, f_A)

2: **for** $i := 1$ **to** N **do**

3: G$[i]$.data := f_A(G$[i]$.data)

Scatter in two steps, illustrated in the last three lines of Figure 6.2b.

Step 1: Oblivious sort: First, perform an oblivious sort on G, so that tuples with the same destination vertex appear adjacent to each other. Further, each vertex should appear after the list of edges ending at that vertex.

Step 2: Aggregate: Next, in a single linear scan, update the value of each vertex with the $\oplus$-sum of the longest preceding sequence of edges. In other words, values on all edges ending at some vertex v are now aggregated into the vertex v.

Efficiency. Let $N := |\mathsf{V}| + |\mathsf{E}|$ denote the total number of tuples. Assume that the data on each vertex and edge is $O(1)$ length, and hence each f_S, f_A, and $\oplus$ operator is of $O(1)$ cost. Clearly, an Apply operation can be performed in $O(N)$ time. Oblivious sort can be performed in $O(N \log N)$ time using [44, 88] while propagate and aggregate take $O(N)$ time. Therefore, a Scatter and a Gather operation each runs in time $O(N \log N)$.

6.3.3 Parallel Oblivious Algorithms for GraphSC

We now describe how to parallelize the sequential oblivious primitives Scatter, Gather, and Apply described in Section 6.3.2. We will describe our parallel algorithms assuming there are a sufficient number of processors, namely $|\mathsf{V}| + |\mathsf{E}|$ processors. Later in Section 6.3.4, we describe some practical optimizations when the number of processors is smaller than $|\mathsf{V}| + |\mathsf{E}|$.

First, observe that the Apply operation can be parallelized trivially. We now demonstrate how to parallelize the Scatter and Gather operations. Recall that both

Scatter and Gather start with an oblivious sort, followed by either an *aggregate* or a *propagate* operation as described in Section 6.3.2. The oblivious sort is, in principle, a $\log(|\mathsf{V}| + |\mathsf{E}|)$-depth circuit [10]. In practice, this sorting circuit is inefficient and thus, we can use a Bitonic sort [15] which has a $\log^2(|\mathsf{V}|+|\mathsf{E}|)$-depth circuit. However, both these circuits are trivially parallelizable at the circuit level.

It thus suffices to show how to execute the *aggregate* and *propagate* operations in parallel. To highlight the difficulty behind the parallelization of these operations, recall that in a data-oblivious execution, a processor needs to, e.g., aggregate values by accessing the list representing the graph at fixed locations, which do not depend on the data. However, as seen in Figure 6.2b, the positions of edges (i.e., black cells) whose values are to be aggregated and stored in vertices (i.e., white cells) clearly depend on the input (namely, the graph G).

Parallelizing the aggregate operation. Recall that an aggregate operation updates the value of each vertex with values of the longest sequence of edges preceding it. For ease of exposition, we first present a few definitions before presenting our parallel *aggregate* algorithm.

Definition 1. Longest Edge Prefix: For $j \in \{1, 2, \ldots, |\mathsf{V}|+|\mathsf{E}|\}$, the longest edge prefix before j, denoted $\mathsf{LEP}[1, j)$, is defined to be the longest consecutive sequence of edges before j, not including j.

Similarly, let $1 \leq i < j \leq |\mathsf{V}| + |\mathsf{E}|$, we use the notation $\mathsf{LEP}[i, j)$ to denote the longest consecutive sequence of edges before j, constrained to the subarray $\mathsf{G}[i \ldots j)$ (index i being inclusive, and index j being exclusive).

Definition 2. Longest Prefix Sum: Let $1 \leq i < j \leq |\mathsf{V}|+|\mathsf{E}|$, we use the notation $\mathsf{LPS}[i, j)$ to denote the "sum" (with respect to the $\oplus$ operator), of $\mathsf{LEP}[i, j)$.

Abusing notation, we treat $\mathsf{LPS}[i, j)$ as an alias for $\mathsf{LPS}[1, j)$ if $i < 1$. The parallel aggregate algorithm is described in Figure 6.3. The algorithm proceeds in a total of $\log(|\mathsf{V}| + |\mathsf{E}|)$ steps. In each intermediate step τ, a processor $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ computes $\mathsf{LPS}[j - 2^\tau, j)$. As a result, at the conclusion of these $\log(|\mathsf{V}| + |\mathsf{E}|)$ steps, each processor j has computed $\mathsf{LPS}[1, j)$.

This way, by time τ, all processors compute the LPS values for all segments of length 2^τ. Now, observe that $\mathsf{LPS}[j - 2^\tau, j)$ can be computed by combining $\mathsf{LPS}[j - 2^\tau, j - 2^{\tau-1})$ and $\mathsf{LPS}[j - 2^{\tau-1}, j)$ in a slightly subtle (but natural) manner as described in Figure 6.3. Intuitively, at each τ, a segment is aggregated with the immediately preceding segment of equal size *only if* a vertex has not be encountered so far.

At the end of $\log(|\mathsf{V}|+|\mathsf{E}|)$ steps, each processor j working on a vertex, appends its data to the aggregation result $\mathsf{LPS}[1, j)$ – this part is omitted from Figure 6.3 for simplicity.

Parallelizing the propagate operation. Recall that, in a *propagate* operation, each edge updates its data with the data of the nearest preceding vertex. The *propagate* operation can be parallelized in a manner similar to *aggregate*. In fact, we can even express a *propagate* operation as a special *aggregate* operation as follows: Initially, every edge stores *(i)* the value of the preceding vertex if a vertex precedes; and *(ii)* $-\infty$ otherwise. Next, we perform an *aggregate* operation where the $\oplus$

Parallel Aggregate:

/ For convenience, assume that for $i \leq 0$, $\mathsf{G}[i]$ is a vertex; and similarly for $i \leq 0$, $\mathsf{LPS}[i,j)$ as an alias for $\mathsf{LPS}[1,j)$ */.*

Initialize: Every processor j computes:

- $\mathsf{LPS}[j-1,j) := \begin{cases} \mathsf{G}[j-1].\mathsf{data} & \text{if } \mathsf{G}[j-1] \text{ is an edge} \\ \mathbf{1}_{\oplus} & \text{o.w.} \end{cases}$
- $\mathsf{existsvert}[j-1,j) := \begin{cases} \mathsf{False} & \text{if } \mathsf{G}[j-1] \text{ is an edge} \\ \mathsf{True} & \text{o.w.} \end{cases}$

Main algorithm: For each time step $\tau := 1$ to $\log(|\mathsf{V}| + |\mathsf{E}|) - 1$: each processor j computes

- if $\mathsf{existsvert}[j-2^{\tau-1},j) = \mathsf{False}$

  $\mathsf{LPS}[j-2^{\tau},j) := \mathsf{LPS}[j-2^{\tau},j-2^{\tau-1}) \oplus \mathsf{LPS}[j-2^{\tau-1},j)$

  else $\mathsf{LPS}[j-2^{\tau},j) := \mathsf{LPS}[j-2^{\tau-1},j)$

- $\mathsf{existsvert}[j-2^{\tau},j) := \mathsf{existsvert}[j-2^{\tau},j-2^{\tau-1})$ or $\mathsf{existsvert}[j-2^{\tau-1})$

Figure 6.3: Performing the aggregate operation (Step 2 of Gather) in parallel, assuming sufficient number of processors with a shared memory to store the variables.

operator is defined to be the max operator. At the end of $\log |\mathsf{V}| + |\mathsf{E}|$ time steps, each processor has computed $\mathsf{LPS}[1, j)$, i.e., the value of the nearest vertex preceding j. Now if cell $\mathsf{G}[j]$ is an edge, we can overwrite its data entry with $\mathsf{LPS}[1, j)$.

Cost analysis. Recall our standing assumption that the maximum data length on each tuple is $O(1)$. It is not hard to see that the parallel runtime of both the *aggregate* and *propagate* operations is $O(\log(|\mathsf{V}| + |\mathsf{E}|))$. The total amount of work for both *aggregate* and *propagate* is $O((|\mathsf{V}| + |\mathsf{E}|) \cdot \log(|\mathsf{V}| + |\mathsf{E}|))$.

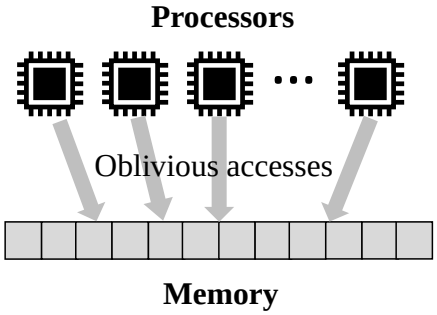
Based on this, we can see that Scatter and Gather each takes $O(\log(|\mathsf{V}| + |\mathsf{E}|))$ parallel time and $O((|\mathsf{V}| + |\mathsf{E}|) \cdot \log(|\mathsf{V}| + |\mathsf{E}|))$ total amount of work. Obviously, Apply takes $O(1)$ parallel time and $O(|\mathsf{V}| + |\mathsf{E}|)$ total work.

Table 6.1 illustrates the performance of our parallel oblivious algorithms for the common case when $|\mathsf{E}| = \Omega(|\mathsf{V}|)$, and the blowup in comparison with a parallel insecure version. Notice that in the insecure world, there exists a trivial $O(1)$ parallel-time algorithm to evaluate Scatter and Apply operations. However, in the insecure world, Gather would take $O(\log(|\mathsf{E}| + |\mathsf{V}|))$ parallel time to evaluate the $\oplus$-sum over $|\mathsf{E}| + |\mathsf{V}|$ variables. Notice also that the $|\mathsf{V}|$ term in the asymptotic bound is absorbed by the $|\mathsf{E}|$ term when $|\mathsf{E}| = \Omega(|\mathsf{V}|)$. The above performance characterization is summarized by the following theorem:

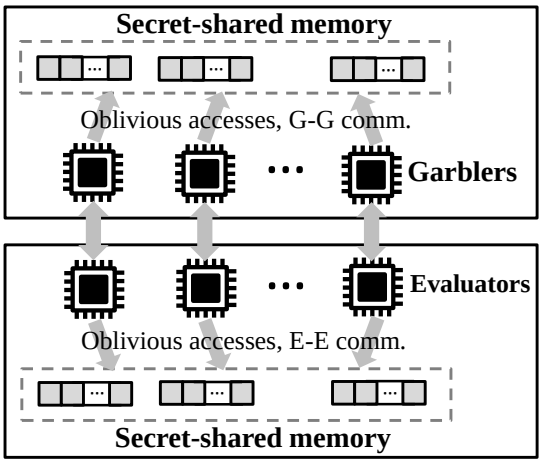
Theorem 21 (Parallel oblivious algorithm for GraphSC)**.** *Let $M := |\mathsf{V}| + |\mathsf{E}|$ denote the graph size. There exists a parallel oblivious algorithm for programs in the GraphSC model, where each Scatter or Gather operation requires $O(\log M)$ parallel time and $O(M \log M)$ total work; and each* Apply *operation requires $O(1)$ parallel*

Op	Total work			Parallel time/Depth		
	Par. insec.	Par. obliv.	Blowup	Par. insec.	Par. obliv.	Blowup
Scatter	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\| \log \|\mathsf{V}\|)$	$O(\log \|\mathsf{V}\|)$	$O(1)$	$O(\log \|\mathsf{V}\|)$	$O(\log \|\mathsf{V}\|)$
Gather	$O(\|\mathsf{E}\| \log d)$	$O(\|\mathsf{E}\| \log \|\mathsf{V}\|)$	$O(\log_d \|\mathsf{V}\|)$	$O(\log d)$	$O(\log \|\mathsf{V}\|)$	$O(\log_d \|\mathsf{V}\|)$
Apply	$O(\|\mathsf{V}\|)$	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\|/\|\mathsf{V}\|)$	$O(1)$	$O(1)$	$O(1)$

Table 6.1: **Complexity of our parallel oblivious algorithms assuming** $|\mathsf{E}| = \Omega(|\mathsf{V}|)$**.** $|V|$ denotes the number of vertices, and $|\mathsf{E}|$ denotes the number of edges. d denotes the maximum degree of a vertex in the graph. *Blowup* is defined as the ratio of the parallel oblivious algorithm with respect to the best known parallel insecure algorithm. We assume that the data length on each vertex/edge is upper-bounded by a known bound D, and for simplicity we omit a multiplicative factor of D from our asymptotical bounds. In comparison with Theorem 21, in this table, some $|\mathsf{V}|$ terms are absorbed by the $|\mathsf{E}|$ term since $|\mathsf{E}| = \Omega(|\mathsf{V}|)$.



(a) Architecture for parallel oblivious algorithms.



(b) Architecture for parallel secure computation.

Figure 6.4: From parallel oblivious algorithms to parallel secure computation.

time and $O(M)$ *total amount of work.*

6.3.4 Practical Optimizations for Fixed Number of Processors

The parallel algorithm described in Figure 6.3 requires $M = |\mathsf{V}| + |\mathsf{E}|$ processors. In practice, however, for large datasets, the number of processors P may be smaller than M. Without loss of generality, suppose that M is a multiple of P. In this case, a naïve approach is for each processor to simulate $\frac{M}{P}$ processors, resulting in $\frac{M \log M}{P}$ parallel time, and $M \log M$ total amount of work. We propose the following practical optimization that can reduce the total parallel time to $O(\frac{M}{P} + \log P)$, and reduce the total amount of work to $O(P \log P + M)$.

We assign to each processor a consecutive range of cells. Suppose that processor j gets range $[s_j, t_j]$ where $s_j = (j-1) \cdot \frac{M}{P} + 1$ and $t_j = j \cdot \frac{M}{P}$. In our algorithm,

each processor will compute $\mathsf{LPS}[1, s_j)$, and afterwards, in $O(M/P)$ time-steps, it can (sequentially) compute $\mathsf{LPS}[1, i)$ for every $s_j \leq i \leq t_j$. Every processor then computes $\mathsf{LPS}[1, s_j)$ as follows

- First, every processor sequentially computes $\mathsf{LPS}[s_j, t_j{+}1)$ and $\mathsf{existswhite}[s_j, t_j{+}1)$.
- Now, assume that every processor started with a single value $\mathsf{LPS}[s_j, t_j + 1)$ and a single value $\mathsf{existswhite}[s_j, t_j + 1)$. Perform the parallel aggregate algorithm on this array of length P.

Sparsity of communication. In a distributed memory setting where memory is split across the processors, the conceptual shared memory is in reality implemented by inter-process communication. An additional advantage of our algorithm is that each processor needs to communicate with at most $O(\log P)$ other processors – this applies to both the oblivious sort step, and the *aggregate* or *propagate* steps. In fact, it is not hard to see that the communication graph forms a hypercube [112].

Let $M := |\mathsf{V}|+|\mathsf{E}|$ and recall that the maximum amount of data on each vertex or edge is $O(1)$. The following corollary summarizes the above observations:

Corollary 22 (Bounded processors, distributed memory.)**.** *When $P < M$, there exists a parallel oblivious algorithm for programs in the GraphSC model, where (a) each processor stores $O(M/P)$ amount of data; (b) each Scatter or Gather operation requires $O(M/P{+}\log P)$ parallel time and $O(P \log P{+}M)$ total work; (c) each Apply operation requires $O(1)$ parallel time and $O(|\mathsf{E}| + |\mathsf{V}|)$ total amount of work; and (d) each processor sends messages to only $O(\log P)$ other processors.*

Security analysis. The oblivious nature of our algorithms is not hard to see: in every time step, the shared memory locations accessed by each processor is fixed and independent of the sensitive input. This can be seen from Figure 6.3, and the description of practical optimizations in this section.

6.4 From Parallel Oblivious Algorithms to Parallel Secure Computation

So far, we have discussed how GraphSC primitives can be implemented as efficient parallel oblivious algorithms, we now turn our attention to how the latter translate to parallel *secure* computation. In this section, we outline the reduction between the two, focusing on a garbled-circuit backend [150] for secure computation.

System Setting. Recall that our focus in this chapter is on secure 2-party computation. As an example, Figure 6.4b depicts two *non-colluding* cloud service providers (e.g., Facebook and Amazon) – henceforth referred to as the two parties. The sensitive data (e.g., user preference data, sensitive social graphs) can be secret-shared between these two parties. Each party has P processors in total – thus there are in total P pairs of processors. The two parties wish to run a parallel secure computation protocol computing a function (e.g., matrix factorization), over the secret-shared data.

While in general, other secure 2-party computation protocols can also be employed, this chapter focuses on a garbled circuit backend [150]. Our focus is on the semi-honest model, although this can be extended with existing techniques [93, 100].

Using this secure model, the oblivious algorithm is represented as a binary circuit. One party then acts as the *garbler* and the other acts as the *evaluator*, as illustrated in Figure 6.4b. To exploit parallelization, each of the two parties parallelize the computational task (garbling and evaluating the circuit, respectively) across its processors. There is a one-to-one mapping between garbler and evaluator processors: each garbler processor sends the tables it garbles to the corresponding corresponding evaluator processor, that evaluates them. We refer to such communication as *garbler-to-evaluator* (GE) communication.

Note that there is a natural correspondence between a parallel oblivious algorithm and a parallel secure computation protocol: First, each processor in the former becomes a (garbler, evaluator) pair in the latter. Second, memory in the former becomes secret-shared memory amongst the two parties. Finally, in each time step, each processor's computation in the former becomes a secure evaluation protocol between a (garbler, evaluator) pair in the latter.

Architectural choices for realizing parallelism. There are various choices for instantiating the parallel computing architecture of each party in Figure 6.4b.

- *Multi-core processor architecture.* At each party, each processor can be implemented by a core in a multi-core processor architecture. These processors share a common memory array.
- *Compute cluster.* At each party, each processor can be a machine in a compute cluster. In this case, accesses to the "shared memory" are actually implemented with *garbler-to-garbler* communication or *evaluator-to-evaluator* com-

munication. In other words, the memory is conceptually shared but physically distributed.

- *Hybrid.* The architecture can be a hybrid of the above, with a compute cluster where each machine is a multi-core architecture.

While our design applies to all three architectures, we used a hybrid architecture in our implementation, exploiting both multi-core *and* multi-machine parallelism. Note that, in the case of a hybrid or cluster architecture with P machines, Corollary 22 implies that each garbler (evaluator) communicates with only $O(\log P)$ other garblers (evaluators) throughout the entire execution. In particular, both garblers and evaluators connect through a hypercube topology. This is another desirable property of GraphSC.

Metrics. Using the above natural correspondence between a parallel oblivious algorithm and a parallel secure computation protocol, there is also a natural correspondence between the primary performance metrics in these two settings: First, the total work of the former directly characterizes (a) the total work *and* (b) the total garbler-to-evaluator (GE) communication in the latter. Second, the parallel runtime of the former directly characterizes the parallel runtime of the latter. We note that, in theory, the garbler is infinitely parallelizable, as each gate can be garbled independently. However, the parallelization of the evaluator (and, thus, of the entire system) is confined by the sequential order defined by the circuit. Thus, parallel runtime is determined by the circuit depth.

In the cluster and hybrid cases, where memory is conceptually shared but

physically distributed, two additional metrics may be of interest, namely, the *garbler-to-garbler* (GG) communication and *evaluator-to-evaluator* (EE) communication. These directly relate to the parallel runtime, since in each parallel time step, each processor makes only one memory access; hence, each processor communicates with at most one other processor at each time-step.

6.5 Evaluation

In this section we present a detailed evaluation of our systems for a few well-known applications that are commonly used for evaluating highly-parallelizable frameworks.

6.5.1 Application Scenarios

In all scenarios, we assume that the data is secret-shared across two non-colluding cloud providers, as motivated in Section 6.4. In all cases, we refer to the total number of vertices and edges in the corresponding GraphSC graph as *input size*.

Histogram. A canonical use case of MapReduce is a word-count (or histogram) of words across multiple documents. Assuming a (large) corpus of documents, each comprising a set of words, the algorithm counts word occurrences across all documents. The MapReduce algorithm maps each word as a key with the value of 1, and the reducer sums up the values of all keys, resulting in the count of appearances of each word. In the secure version, we want to compute the word frequency histogram

while hiding the text in each document. In GraphSC, this is a simple instance of edge counting over a bipartite graph G, where edges connect keys to words. We represent keys and words as 16-bit integers, while accumulators (i.e., key vertex data) are stored using 20-bit integers.

Simplified PageRank. A canonical use case of graph parallelization models is the PageRank algorithm. We consider a scenario in which multiple social network companies, e.g., Facebook, Twitter and LinkedIn, would like to compute the "real" social influence of users on a social graph that is the aggregate of each company's graph (assume users are uniquely identified across networks by their email address). In the secure version, each company is not willing to reveal user data and their social graph with the other network. Vertices are identified using 16-bit integers, and 1bit for isVertex (see Section 6.3.2). The PageRank value of each vertex is stored using a 40-bit fixed-point representation, with 20-bit for the fractional part.

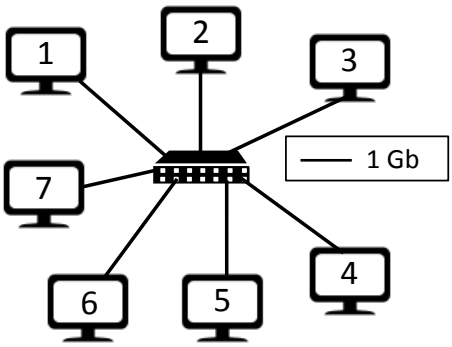
Matrix Factorization (MF). Matrix Factorization [91] splits a large sparse low-rank matrix into two dense low-dimension matrices that, when multiplied, closely approximate the original matrix. Following the Netflix prize competition [18], matrix factorization is widely used in recommender systems. In the secure version, we want to factorize the matrix and learn the user or item feature vectors (learning both can reveal the original input), while hiding both the ratings and items each user has rated. MF can be expressed in GraphSC using a bi-partite graph with vertices representing users and items, and edges connecting each user to the items they rated, carrying the ratings as data. In addition, data at each vertex also contains a feature vector, corresponding to its respective row in the user/item factor matrix. We study

two methods for matrix factorization – gradient descent and alternative least-squares (ALS) (see, e.g., [91]). In gradient descent, the gradient is computed for each rating separately, and then accumulated for each user and each item feature vectors, thus it is highly parallelizable. In ALS we alternate the computation between user feature vectors (assuming fixed item feature vectors) and item feature vectors (assuming fixed user feature vectors). For each step, each vector solves (in parallel) a linear regression using the data from its neighbors. Similar to PageRank, we use 16-bit for vertex id and 1-bit for isVertex. The user and item feature vectors are with dimension 10, with each element stored as a 40-bit fixed-point real.

The secure implementation of matrix factorization using gradient descent has been studied by Nikolaenko et al. [119] who, as discussed in Section 6.1, constructed circuits of linear depth. The authors used a multi-core machine to exploit parallelization during sorting, and relied on shared memory across threads. This limits the ability to scale beyond a single machine, both in terms of the number of parallel processors (32 processors) as well as, crucially, input size (they considered no more than 17K ratings, over a 128 GB RAM server).

6.5.2 Implementation

We implemented GraphSC atop ObliVM-GC, the Java-based garbled circuit implementation that comprises the back end of the GraphSC secure computation framework [7, 104]. ObliVM-GC provides easy-to-use Java classes for composing circuit libraries. We extend ObliVM-GC with a simple MPI-like interface where pro-



(a) Evaluation setup, all machines are connected in a star topology with 1Gbps links.

Machine	#Proc	Memory	CPU Freq
1	24	128 GB	1.9 GHz
2	24	128 GB	1.9 GHz
3	24	64 GB	1.9 GHz
4	24	64 GB	1.9 GHz
5	24	64 GB	1.9 GHz
6	32	128 GB	2.1 GHz
7	32	256 GB	2.6 GHz

(b) Servers' hardware used for our evaluation. The processor used for machine 6 is AMD Opteron 6272. For all other machines, AMD Opteron 6282 SE is used.

Figure 6.5: Experimental setup for our evaluation.

cesses can additionally call non-blocking `send` and blocking `receive` operations. Processes in ObliVM-GC are identified by their unique identifiers.

Finally, we implement oblivious sorting using the bitonic sort protocol [88] which sorts in $O(N \log^2 N)$ time. Asymptotically faster protocols such as the $O(N \log N)$ AKS sort [10] and the recent ZigZag sort [69] are much slower in practice for practical ranges of data sizes.

6.5.3 Setup

We conduct experiments on both a testbed that uses a LAN, and on a realistic Amazon AWS deployment. We first describe our main experiments conducted using a compute cluster connected by a Local Area Network. Later, in Section 6.5.8, we will describe results from the AWS deployment.

Testbed Setup on Local Area Network: Our experimental testbed consists of 7 servers with the configurations detailed in Table 6.5b. These servers are interconnected using a star topology with 1Gbps Ethernet links as shown in Figure 6.5a. All experiments (except the large-scale experiment reported in Section 6.5.6 that uses all of them) are performed using a pair of servers from the seven machines. These servers were dedicated to the experiments during our measurements, not running processes by other users.

To verify that our results are robust, we repeated the experiments several times, and made sure that the standard deviation is small. For example, we ran PageRank 10 times using 16 processors for an input length of 32K. The resulting mean execution time was 390 seconds, with a standard deviation of 14.8 seconds; we therefore report evaluations from single runs.

6.5.4 Evaluation Metrics

We study the gains and overheads that result from our parallelization techniques and implementation. Specifically, we study the following key metrics:

Total Work. We measure the total work using the overall number of AND gates

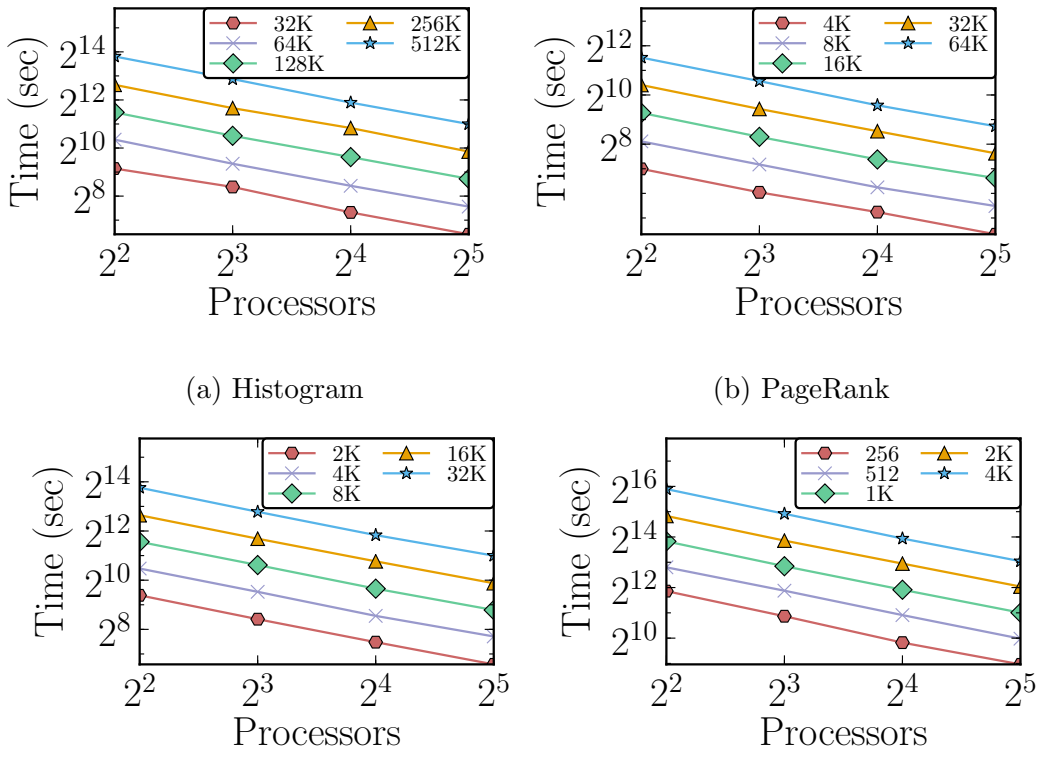
for each application. As mentioned earlier in Section 6.3.4, the total work grows logarithmically with respect to the number of processors P in theory – and in practice, since we employ bitonic sort, the actual growth is log-squared.

Actual runtimes. We report our actual runtimes and compare the overhead with a cleartext baseline running over GraphLab [2, 68, 105]. We stress that while our circuit size metrics are *platform independent*, actual runtime is a platform dependent metric. For example, we expect a factor of 20 speedup if the backend garbled circuit implementation adopts a JustGarble-like approach (using hardware AES-NI) – assuming roughly 2700 Mbps bandwidth provisioned between each garbler and evaluator pair.

Speedup. The obvious first metric to study is the speedup in the time to run each application as a result of adding more processors. In our applications, computation is the main bottleneck. Therefore, in the ideal case, we should observe a factor of x speedup with x factor more processors.

Communication. Parallelization introduces communication overhead between garblers and between evaluators. We study this overhead and compare it to the communication between garblers and evaluators.

Accuracy. Although not directly related to parallelization, for completeness we study the loss in accuracy obtained as a result of implementing the secure version of the applications, both when using fixed-point representation and floating-point representation of the reals.



(a) Histogram

(b) PageRank

(c) Gradient Descent

(d) ALS

Figure 6.6: Computation time for increasing number of processors, showing an almost linear decrease with the number of processors. The lines correspond to different input lengths. For PageRank, gradient descent and ALS, the computation time refers to the time required for one iteration.

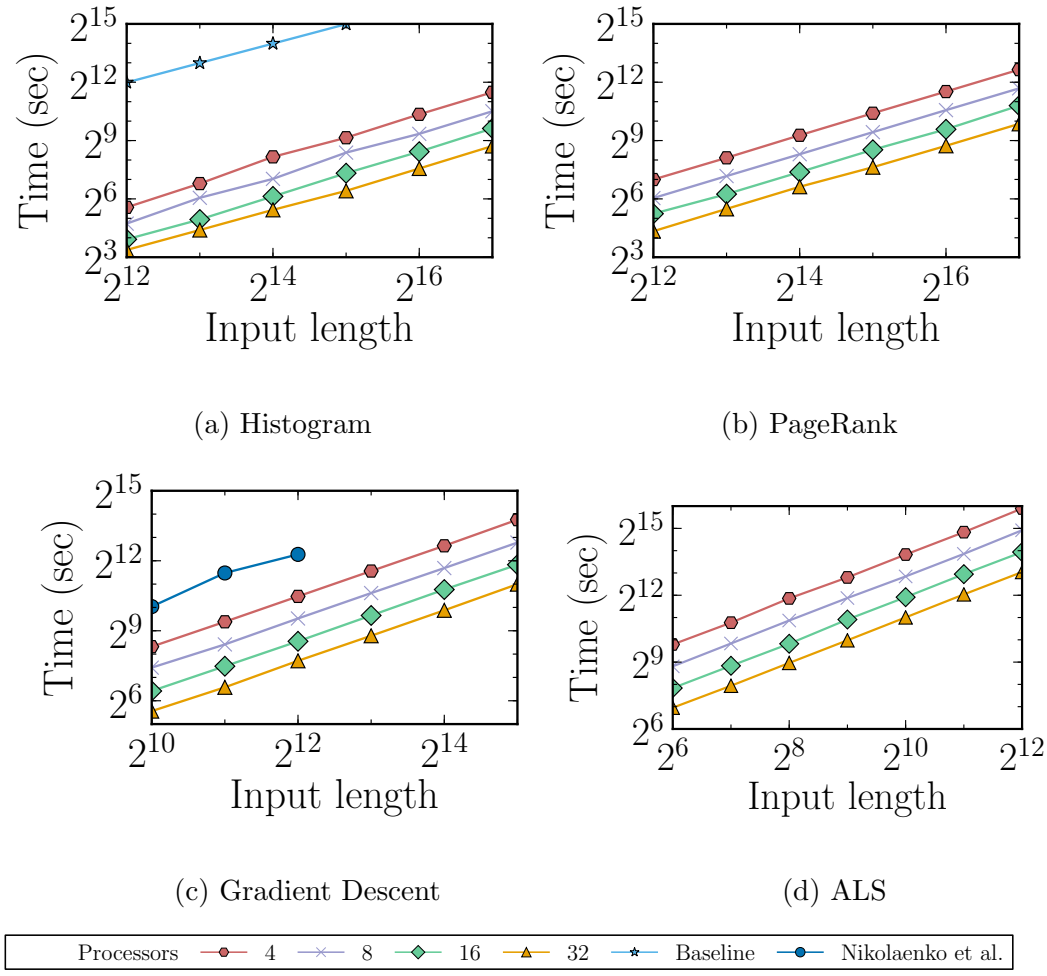


Figure 6.7: Computation time for increasing input size, showing an almost-linear increase with the input size, with a small $\log^2$ factor incurred by the bitonic sort. The lines correspond to different input lengths. For PageRank, gradient descent and ALS, the computation time refers to the time required for one iteration. In Figure 6.7a, the baseline is a sequential ORAM-based baseline using Circuit ORAM [143]. Figure 6.7c compares our performance with the performance of Nikolaenko et al. [119] who implemented the circuit using FastGC [82] and parallelized at the circuit level using 32 processors.

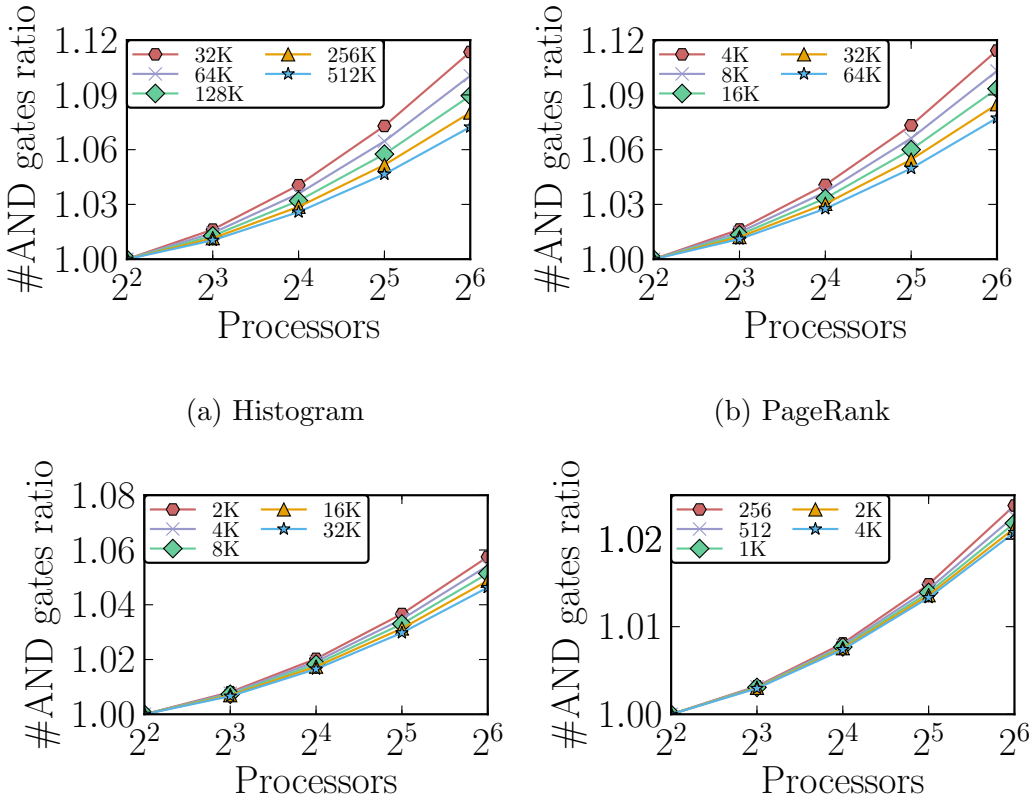
6.5.5 Main Results

Speedup. Figure 6.6 shows the total computation time across the different applications. For all applications except histogram we show the time of a single iteration (consecutive iterations are independent). Since in our experimental setup computation is the bottleneck, the figures show an almost ideal linear speedup as the number of processors grow. Figure 6.7 shows that our method is highly scalable with the input size, with an almost linear increase (a factor of $O(P/\log^2 P)$). Figure 6.7a provides the time to compute a histogram using an oblivious RAM implementation. We use the state-of-the-art Circuit ORAM [143] for this purpose. As the figure shows, the baseline is 2 orders of magnitude slower compared to the parallel version using two garblers and two evaluators.

Figure 6.7c provides the timing presented in Nikolaenko et al. [120] using 32 processors. As the figure shows, using a similar hardware architecture, we manage to achieve a speedup of roughly $\times 16$ compared to their results. Most of the performance gains comes from the usage of GraphSC architecture – whereas Nikolaenko et al. used a multi-threaded version of FastGC [82] as the secure computation backend.

Total Work. Figure 6.8 shows that the total amount of work grows very slowly with respect to the number of processors, indicating that we indeed achieved a very low overhead in the total work (and overall circuit size).

Communication. Figure 6.9a and Figure 6.9b show the amount of total communication and per processor communication, respectively, for running gradient descent. Each plot shows both the communication between garblers and evalua-



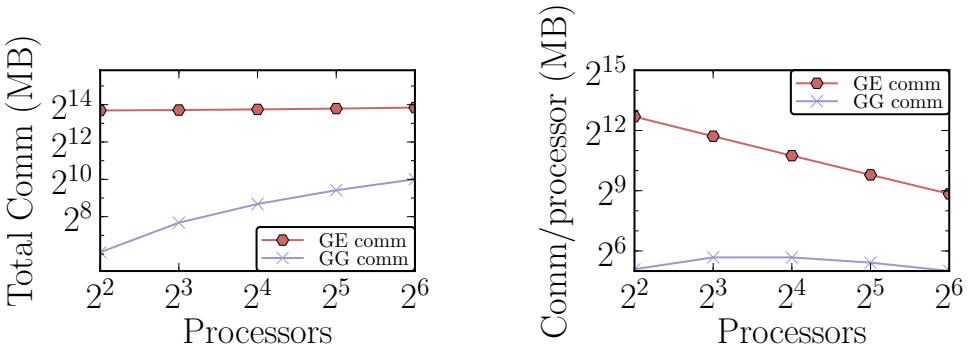
(a) Histogram

(b) PageRank

(c) Gradient Descent

(d) ALS

Figure 6.8: Total work in terms of # AND gates, normalized such that the 4 processor case is 1×. The different curves correspond to different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors P. Recall that our theoretical analysis suggests that the total amount of work is $O(P \log P + M)$, where $M := |\mathsf{V}| + |\mathsf{E}|$ is the graph size. In practice, since we use bitonic sort, the actual total work is $O(P \log^2 P + M)$.



(a) Total Communication

(b) Communication per proc.

Figure 6.9: Communication of garbler-evaluator (GE) and garbler-garbler (GG) for gradient descent (input length 2048).

tors, and the overhead introduced by the communication between garblers (communication between evaluators is identical). Figure 6.9a shows that the total communication between garblers and evaluators remains constant as we increase the number of processors, showing that parallelization does not introduce overhead to the garblers-to-evaluator communication. Furthermore, the garbler-to-garbler (GG) communication is significantly lower than the garblers-to-evaluator communication, showing that the communication overhead due to parallelization is low. As expected, adding more processors increases the total communication between garblers, following $\log^2 P$ (where P is the number of processors), due to the bitonic sort. Figure 6.9b shows the communications per-processor (dividing the results of Figure 6.9a by P). This helps understand overheads in our setting, where, for example, a cloud provider that provides secure computation services (garbling or evaluating) is interested in the communication costs of its facility rather than the total costs. As the number of processors increase, the "out-going" communication (e.g., a provider running

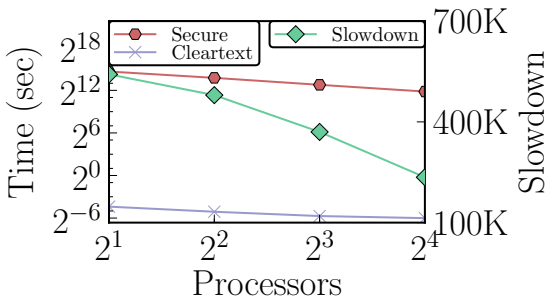


Figure 6.10: Comparison with cleartext implementation on GraphLab for gradient descent (input length 32K)

garblers see the communication with evaluators as "out-going" communication) decreases. The GG communication (or EE communication) remains roughly the same (following $\log^2 P/P$), and significantly lower than the "out-going" communication.

Comparison with a Cleartext Baseline. To better understand the overhead that is incurred from cryptography, we compared GraphSC's execution time with GraphLab [2, 68, 105], a state-of-the-art framework for running graph-parallel algorithms on clear text. We compute the slowdown relative to an insecure baseline, assuming that the same number of processors is employed for GraphLab and GraphSC. Using both frameworks, we ran Matrix Factorization using gradient descent with input length of 32K. For the cleartext experiments, we ran 1000 iterations of gradient descent 3 times, and computed the average time for a single iteration.

Figure 6.10 shows that GraphSC is about 200K - 500K times slower than GraphLab when run on 2 to 16 processors. Since GraphLab is highly optimized and extremely fast, such a large discrepancy is expected. Nevertheless, we note that increasing parallelism decreases this slowdown, as overheads and communication

Table 6.2: Summary of machines used in large-scale experiment, performing matrix factorization over the MovieLens 1M ratings dataset.

Machine	Processors	Type	JVM Memory Size	Num Ratings
1	16	Garbler	64 GB	256K
2	16	Evaluator	60.8 GB	256K
3	6	Garbler	24 GB	96K
3	6	Evaluator	24 GB	96K
4	15	Garbler	58.5 GB	240K
5	15	Evaluator	58.5 GB	240K
6	27	Garbler	113.4 GB	432K
7	27	Evaluator	121.5 GB	432K
Total	128		524.7 GB	1M

costs impact both systems.

6.5.6 Running at Scale

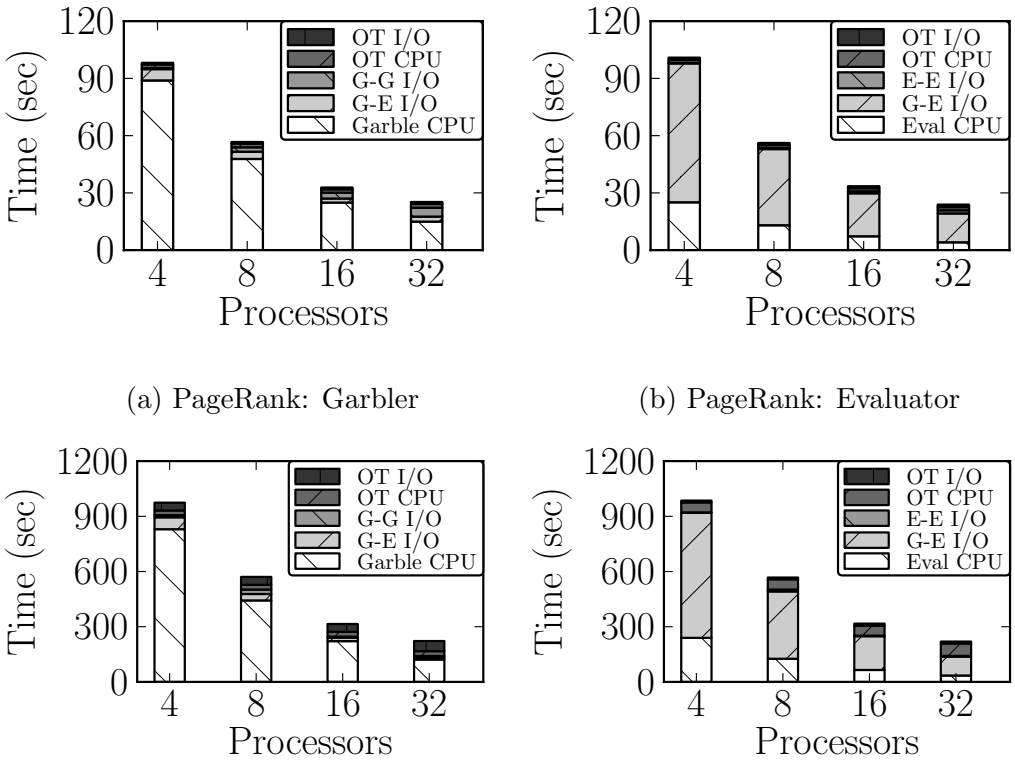
In order to have a full-scale experiment of our system, we ran matrix factorization using gradient descent on the real-world MovieLens dataset that contains 1 million ratings provided by 6040 users to 3883 movies [4]. We factorized the matrix to users and movie feature vectors, each vector with a dimension of 10. We used 40-bit fixed-point representation for reals, with 20 bits reserved for the fractional part. We ran the experiment on an heterogeneous set of machines that we have in the lab. Table 6.2 summarizes the machines and the allocation of data across them.

A single iteration of gradient descent took roughly 13 hours to run on 7 machines with 128 processors, at ˜104 MB data size (i.e., 1M entries). As prior machine learning literature reports [19,84], about 20 iterations are necessary for convergence for the same MovieLens dataset – which would take about 11 days with 128 processors. In practice, this means that the recommendation system can be retrained every 11 days. As mentioned earlier, about 20× speedup is immediately attainable by switching to a JustGarble-like back end implementation with hardware AES-NI, and assuming 2700 Mbps bandwidth between each garbler-evaluator pair. One can also speed up the execution by provisioning more processors.

In comparison, as far as we know, the closest large-scale experiment in running secure matrix factorization was recently performed by Nikolaenko et al. [119]. The authors used 16K ratings and 32 processors to factorize a matrix (on a machine similar to machine 7 in Table 6.2), taking almost 3 hours to complete. The authors could not scale further because their framework runs on a single machine.

6.5.7 Performance Profiling

Finally, we perform micro-benchmarks to better understand the time the applications spend in the different parts of the computation and network transmissions. Figure 6.11 shows the breakdown of the overall execution between various operations for PageRank and gradient descent. Figure 6.12 shows a similar breakdown for different input sizes. As the plots show, the garbler is computation-intensive whereas the evaluator spends a considerable amount of time waiting for the gar-



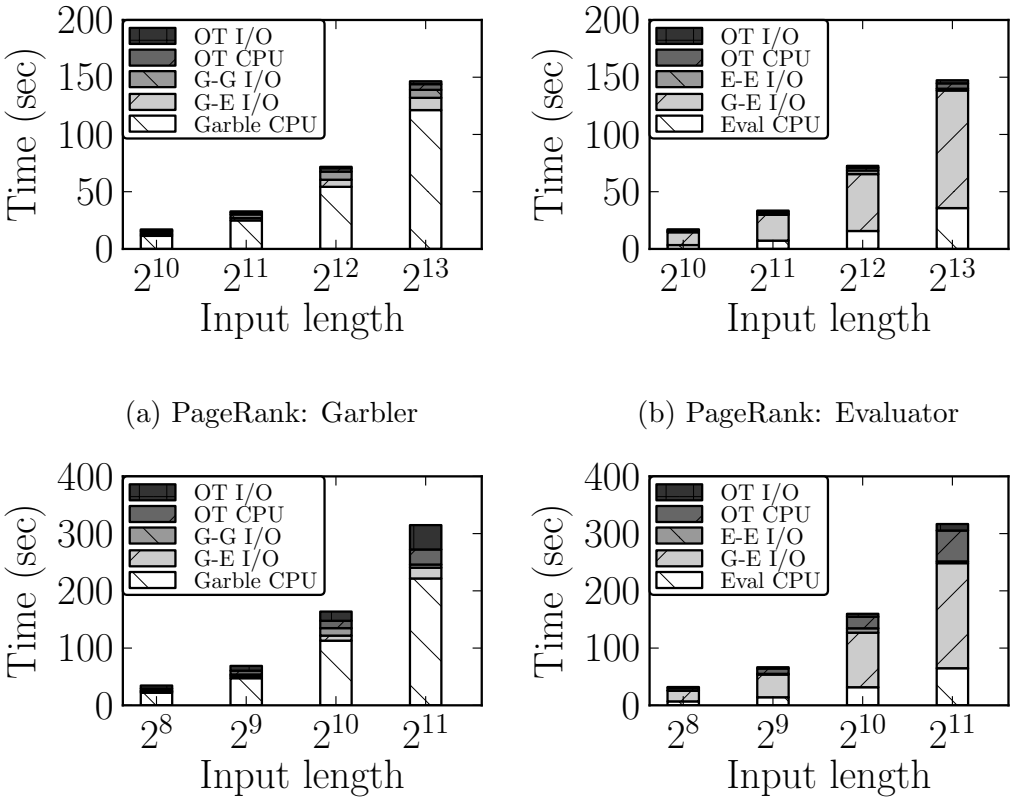
(a) PageRank: Garbler

(b) PageRank: Evaluator

(c) Gradient Descent: Garbler

(d) Gradient Descent: Evaluator

Figure 6.11: A breakdown of the execution times of the garbler and evaluator running one iteration of PageRank and gradient descent for an input size of 2048 entries Here I/O overhead means the time a processor spends blocking on I/O. The remaining time is reported as CPU time.



(a) PageRank: Garbler

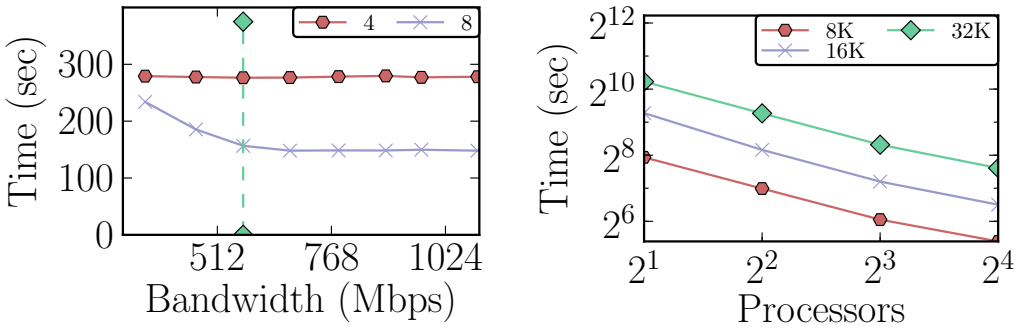
(b) PageRank: Evaluator

(c) Gradient Descent: Garbler

(d) Gradient Descent: Evaluator

Figure 6.12: A breakdown of the execution times of the garbler and evaluator running one iteration of PageRank and gradient descent for an increasing input size using 8 processors for garblers and 8 for evaluators.

bled tables (receive is a blocking operation). In our implementation, the garbler computes 4 hashes to garble each gate, and the evaluator computes only 1 hash for evaluation. This explains why the evaluation time is smaller than the garbling time. Since the computation tasks under consideration are superlinear in the size of the inputs, we see that the time spent on oblivious transfer (both communication and computation) is insignificant in comparison to the time for garbling/evaluating. Our current implementation is built atop Java, and we do not make use of hardware



(a) Varying bandwidths.

(b) Across data centers

Figure 6.13: Performance of PageRank. Figure 6.13a shows performance for 4 and 8 processors at varying bandwidths. The dotted vertical line indicates the inflexion point for 8 processors, below which the bandwidth becomes a bottleneck, resulting in reduced performance. Figure 6.13b shows the performance of PageRank running on geographically distant data centers (Oregon and North Virginia).

AES-NI instructions. We expect that the garbling and evaluation CPU will reduce noticeably if hardware AES-NI were employed [16]. We leave it for future work to port GraphSC to a C-based implementation capable of employing hardware AES-NI features.

6.5.8 Amazon AWS Experiments

We conduct two experiments on Amazon AWS machines. First, we study the performance of the system under different bandwidths on the same AWS data center (Figure 6.13a). Second, to test the performance on a more realistic deployment, where the garbler and evaluator are not co-located, we also conduct experiments by deploying GraphSC on a pair of AWS virtual machines located in different geo-

graphical regions (Figure 6.13b).

The time reported for these experiments should not be compared to the earlier experiments as different machines were used.

Setup. For the experiments with varying bandwidths, both garblers and evaluators were located in the same data center (Oregon - US West). For the experiment across data centers, the garblers were located in Oregon (US West) and the evaluators were located in N. Virginia (US East). We ran our experiments on shared instances running on Intel Xeon CPU E5-2666 v3 processors clocked at 2.9 GHz. Each of our virtual machines consisted of 16 cores and 30 GB of RAM.

Results for Varying Bandwidths. Since communication between garblers and evaluators is a key component in system performance, we further study the bandwidth requirements of the system on a real-world deployment.

We measure the time for a single PageRank iteration with input length of 16K entries. We vary the bandwidth using `tc` [6], a tool for bandwidth manipulation, and then measure the exact bandwidth between machines using `iperf` [3].

Figure 6.13a shows the execution time for two setups, one with 4 processors (2 garblers and 2 evaluators) and the second with 8 processors. Using 4 processors the required bandwidth is always lower than the capacity of the link, thus the execution time remains the same throughout the experiment. However, when using 8 processors the total bandwidth required is higher, and when the available bandwidth is below 570 Mbps the link becomes saturated. The saturation point indicates that each garbler-evaluator pair requires a bandwidth of $570/4 \approx 142$ Mbps. GraphSC has an effective throughput of ˜ 0.58M gates/sec between a pair of processors on our

Amazon AWS instances. Each gate has a size of 240 bits. Hence, the theoretical bandwidth required is $0.58 \times 240 \times 10^6/2^{20} \approx 133$ Mbps. Considering GraphSC is implemented in Java, garbage collection happens intermittently due to which the communication link is not used effectively. Hence, the implementation requires slightly more bandwidth than the theoretical calculation.

Given such bandwidth requirements, the available bandwidth in our AWS setup, i.e., 2 Gbps between the machines, will saturate beyond roughly 14 garbler-evaluator pairs (28 processors). At this point, the linear speedup trend w.r.t. the number of processors (as shown in Figure 6.6) will stop, unless larger bandwidth becomes available. In a real deployment scenario, the total bandwidth can be increased by having multiple machines for garbling and evaluating, hence supporting more processors without affecting the speedup.

Results for Cross-Data-Center Experiments. For this experiment, the garblers are hosted in the AWS Oregon data center and the evaluators are hosted in the AWS North Virginia data center. We measure the execution time of a single iteration of PageRank for different input lengths. As in the previous experiment, we used machines with 2Gbps network links, however, measuring the TCP throughput with `iperf` resulted in ˜50 Mbps per TCP connection. By increasing the receiver TCP buffer size we managed to increase the effective throughput for each TCP connection to ˜400 Mbps.

Figure 6.13b shows that this realistic deployment manages to sustain a linear speedup when increasing the number of processors. Moreover, even 16 processors do not saturate the 2 Gbps link, meaning that the geographical distance does not

Table 6.3: Summary of key evaluation results (1 iteration).

Experiment	Input size	Time (32 processors)
Histogram	1K - 0.5M	4 sec - 34 min
PageRank	4K - 128K	20 sec - 15.5 min
Gradient Descent	1K - 32K	47 sec - 34 min
ALS	64 - 4K	2 min - 2.35 hours
Gradient Descent large scale)	1M ratings	13 hours (128 processors)

impact the speedup resulting from adding additional processors. We note that if more than 14 garbler-evaluator pairs are needed (to further reduce execution time), AWS provides higher capacity links (e.g., 10 Gbps), thereby allowing even higher degrees of parallelism.

During the computation, the garbler garbles gates and sends it to the evaluator. As there are no round trips involved (i.e. garbler does not wait to receive data from the evaluator), the time required for computation across data centers is the same as in the LAN setting.

6.5.9 Summary of Main Results

To summarize, Table 6.3 highlights some of the results, and we present the main findings:

- As mandated from "big-data" algorithms, GraphSC provides high scalability

with the input size, exhibiting an almost linear increase with the input size (up to poly-log factor).

- Parallelization provides an almost ideal linear improvement in execution time with small communication overhead (especially on computation-intensive tasks), both in a LAN based setting and across data centers.
- GraphSC can work on real workloads. We ran a first-of-its-kind large-scale secure matrix factorization experiment, factorizing a matrix comprised of the MovieLens 1M ratings dataset within 13 hours on a heterogeneous set of 7 machines with a total of 128 processors.
- GraphSC supports fixed-point and floating-point reals representation, yielding an overall low rounding errors (provided sufficient fraction bits) compared to execution in the clear.

6.6 Conclusion

This chapter introduced GraphSC, a parallel data-oblivious and secure framework for efficient implementation and execution of algorithms on large datasets. GraphSC seamlessly integrates modern parallel programming paradigms that are familiar to a wide range of developers into an secure data-oblivious framework.

Chapter 7: Conclusion

In this dissertation, we have shown four contributions to advance the understanding of oblivious computation, both theoretically and in practice. Specifically,

- We show an ORAM construction which achieved a sub-logarithmic bandwidth blowup while requiring the servers to perform an inexpensive XOR computation.

- We show the *first* perfectly-secure OPRAM construction, achieving $O(\log^3 N)$ simulation overhead and $O(\log N(\log m + \log\log N))$ depth blowup when the PRAM has m CPUs and stores N blocks of data.

- We described two systems – HOP and GraphSC – to address the problem of performing graph-parallel computations on private data and the distribution of proprietary programs.

Bibliography

[1] bzip2 man pages. http://www.bzip.org/1.0.5/bzip2.txt.

[2] Graphlab powergraph tutorials. https://github.com/graphlab-code/graphlab.

[3] Iperf. https://iperf.fr/.

[4] Movielens dataset. http://grouplens.org/datasets/movielens/.

[5] Open cores. http://opencores.org/.

[6] Tc man page. http://manpages.ubuntu.com/manpages//karmic/man8/tc.8.html.

[7] http://www.oblivm.com.

[8] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *IACR International Workshop on Public Key Cryptography*, 2017.

[9] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM side–channel(s). In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2002.

[10] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(N \log N)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, 1983.

[11] Miklós Ajtai. Oblivious RAMs without cryptographic assumptions. In *Proceedings of the forty-second ACM symposium on Theory of computing*, 2010.

[12] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, 2014.

[13] Ching Avery. Giraph: Large-scale graph processing infrastruction on hadoop. *Hadoop Summit.*, 2011.

[14] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, 2001.

[15] K. E. Batcher. Sorting Networks and Their Applications. AFIPS '68 (Spring), 1968.

[16] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[17] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 1988.

[18] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, 2007.

[19] Smriti Bhagat, Udi Weinsberg, Stratis Ioannidis, and Nina Taft. Recommending with an agenda: Active learning of private attributes using matrix factorization. In *RecSys '14*. ACM.

[20] Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N Rothblum. Program obfuscation with leaky hardware. In *Advances in Cryptology–ASIACRYPT 2011*. 2011.

[21] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, 2015.

[22] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *IEEE S&P*, 2014.

[23] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[24] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*, 2008.

[25] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 1997.

[26] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC*, 2011.

[27] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011.

[28] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography Conference*, 2016.

[29] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, 2016.

[30] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'99, 1999.

[31] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[32] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 429–437. ACM, 2015.

[33] Christopher Celio and Eric Love. The sodor processor collection. `http://riscv.org/download.html#tab_sodor`.

[34] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[35] T-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In *Asiacrypt*, 2017.

[36] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Asiacrypt*, 2017.

[37] T-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. Cryptology ePrint Archive, Report 2018/364, 2018.

[38] T-H. Hubert Chan and Elaine Shi. Circuit OPRAM: A unifying framework for computationally and statistically secure ORAMs and OPRAMs. In *TCC*, 2017.

[39] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography - 13th International Conference, TCC 2016-A*, 2016.

[40] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *EUROCRYPT*. 2015.

[41] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[42] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT*, 2013.

[43] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2014.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 428–436. MIT Press, third edition, 2009.

[45] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

[46] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits. In *Computer Security–ESORICS 2013*. 2013.

[47] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography Conference (TCC)*, pages 144–163. Springer, 2011.

[48] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, 2004.

[50] Ioannis Demertzis, Dimitris Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO*, 2018.

[51] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016-A*, 2016.

[52] Nico Döttling, Thilo Mie, Jörn Müller-Quade, and Tobias Nilges. Basing obfuscation on simple tamper-proof hardware assumptions. *IACR Cryptology ePrint Archive*, 2011.

[53] Zeev Dvir and Sivakanth Gopi. 2-server PIR with sub-polynomial communication. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC*, 2015.

[54] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *SIGSPATIAL*, 2010.

[55] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, 2012.

[56] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, 2015.

[57] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious RAM controller. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, 2015.

[58] Sanjam Garg. Program obfuscation via multilinear maps. In *Security and Cryptography for Networks*. 2014.

[59] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2013.

[60] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM symposium on Theory of computing (STOC)*, 2009.

[61] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[62] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.

[63] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2014.

[64] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2005.

[65] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM symposium on Theory of computing (STOC)*, 1987.

[66] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[67] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[68] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Operating System Design and Implementation (OSDI)*, 2012.

[69] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(N \log N)$ time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, 2014.

[70] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*, 2011.

[71] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, 2012.

[72] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[73] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[74] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, 2010.

[75] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[76] Satoshi Hada. Zero-knowledge and code obfuscation. In *ASIACRYPT*, 2000.

[77] Torben Hagerup. Fast and optimal simulations between CRCW PRAMs. In *STACS, 9th Annual Symposium on Theoretical Aspects of Computer Science*, 1992.

[78] Torben Hagerup. The log-star revolution. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, 1992.

[79] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, 2010.

[80] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3ORAM: A computation-efficient and constant client bandwidth blowup ORAM with shamir secret sharing. In *Conference on Computer and Communications Security (CCS)*, 2017.

[81] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[82] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Usenix Security Symposium*, 2011.

[83] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Annual Computer Security Applications Conference*, 2013.

[84] Stratis Ioannidis, Andrea Montanari, Udi Weinsberg, Smriti Bhagat, Nadia Fawaz, and Nina Taft. Privacy tradeoffs in predictive analytics. In *SIGMETRICS'14*. ACM, 2014.

[85] Meha Kainth, Lekshmi Krishnan, Chaitra Narayana, Sandesh Gubbi Virupaksha, and Russell Tessier. Hardware-assisted code obfuscation for FPGA soft microprocessors. In *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[86] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2007.

[87] Florian Kerschbaum. Automatically optimizing secure computation. In *Computer and Communication Security Conference (CCS)*, 2011.

[88] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[89] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO'99*, 1999.

[90] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, 2015.

[91] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[92] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security*, 2013.

[93] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security symposium*, 2012.

[94] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, 1997.

[95] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[96] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[97] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. *CoRR*, abs/1802.05145, 2018.

[98] Kevin Lewi, Alex J Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5Gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[99] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 2000.

[100] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*. 2007.

[101] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, 2005.

[102] Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[103] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-model Secure Computation. In *IEEE Security and Privacy (S & P)*, 2014.

[104] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[105] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, 2010.

[106] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference (TCC)*, 2013.

[107] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[108] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[109] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *USENIX Security*, 2004.

[110] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*. Citeseer, 2014.

[111] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[112] Russ Miller and Laurence Boxer. *Algorithms sequential & parallel: A unified approach*. Cengage Learning, 2012.

[113] John C. Mitchell and Joe Zimmerman. Data-Oblivious Data Structures. In *Theoretical Aspects of Computer Science (STACS)*, 2014.

[114] Tarik Moataz, Erik-Oliver Blass, and Travis Mayberry. CHf-ORAM: a constant communication ORAM without homomorphic encryption. Cryptology ePrint Archive, Report 2015/1116, 2015.

[115] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[116] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. HOP: Hardware makes obfuscation practical. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[117] Kartik Nayak and Jonathan Katz. An oblivious parallel RAM with $O(\log^2 N)$ parallel runtime blowup. Cryptology ePrint Archive, Report 2016/1141, 2016. `http://eprint.iacr.org/2016/1141`.

[118] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[119] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[120] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S & P*, 2013.

[121] Olga Ohrimenko, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In *ACM CCS*, 2015.

[122] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[123] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, 1998.

[124] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, 2015.

[125] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE S & P*, 2014.

[126] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, 2015.

[127] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ACM SIGARCH Computer Architecture News*, 2013.

[128] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[129] John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[130] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, 2011.

[131] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud. In *IEEE S& P*, 2015.

[132] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, 2011.

[133] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In *CCS*, 2013.

[134] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O(\log^3 N)$ worst-case cost. In *ASIACRYPT*, 2011.

[135] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST)*, 2010.

[136] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[137] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013.

[138] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2011.

[139] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. Cryptology ePrint Archive, Report 2013/280 v3, 2013. `http://eprint.iacr.org/2013/280`.

[140] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[141] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Conference on Supercomputing*, 2003.

[142] L. Tan. The worst case execution time tool challenge 2006: The external test. In *Leveraging Applications of Formal Methods, Verification and Validation*, 2006.

[143] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[144] Xiao Wang, Dov Gordon, and Jonathan Katz. Simple and efficient two-server oram. Cryptology ePrint Archive, Report 2018/005, 2018. https://eprint.iacr.org/2018/005.

[145] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[146] Peter Williams and Radu Sion. SR-ORAM: Single round-trip oblivious RAM. *ACNS, Industrial Track*, 2012.

[147] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security*, 2012.

[148] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Security and Privacy (SP)*, 2015.

[149] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *IEEE symposium on Foundations of Computer Science (FOCS)*, 1982.

[150] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[151] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[152] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *S & P*, 2013.

[153] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. KT-ORAM: A bandwidth-efficient oram built on k-ary tree of PIR nodes. 2014.

[154] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. MSKT-ORAM: A constant bandwidth ORAM without homomorphic encryption. IACR Cryptology ePrint Archive, Report 2016/882, 2016.

[155] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In *Computer and Communication Security Conference (CCS)*, 2013.

[156] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ACM SIGPLAN Notices*, 2004.