A Survey and Analysis of Solutions to the Oblivious Memory Access Problem

by

Erin Elizabeth Chapman

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Thomas Shrimpton, Chair
Melanie Mitchell
Bryant York

Portland State University
2012

ABSTRACT

Despite the use of strong encryption schemes, one can still learn information about encrypted data using side channel attacks [2]. Watching what physical memory is being accessed can be such a side channel. One can hide this information by using *oblivious simulation* – hiding the true access pattern of a program. In this paper we will review the model behind oblivious simulation, attempt to formalize the problem and define a security game. We will review the major solutions proposed so far, the square root and hierarchical solutions, as well as propose a new variation on the square root solution. Additionally, we will show a new formalization for providing software protection by using an encryption scheme and oblivious simulation.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

For centuries, there has been a need to hide the content of messages. Early uses include the Caesar cipher, a simple substitution cipher where each letter is replaced by another, to hide military orders. As knowledge of cryptography increased, the methods used to hide messages have become increasingly complex. Today, we use many mathematical concepts to hide the content of messages. Sometimes hiding the content of messages is not sufficient however. If an attacker is able to gather enough information outside of the message content, this may lead them to learn information hidden in the message itself. For example, if the attacker notices that every time the target accesses a specific file on the Internet, they send a shipment of parts the next day, the attacker is able to learn information about the target that should potentially be kept secret. In this case, simply hiding the message content does not provide enough security and such side channel attacks are possible. To provide complete security, the target also needs to hide what files they are accessing.

Chen et. al. showed that such side channel attacks have become a viable threat with today's infrastructure [2]. Most people have become used to using Internet services to get health information or file their taxes. Such server transactions are typically encrypted using HTTPS. Despite using encryption, these researchers were able to show that detail information about the user could still be learned, such as the illness of a person and the medications they are taking or their family

income. They were able to learn this information simply by watching the amount of data being sent back and forth between the client and the server. For example, in an online health application, when the user selected a health condition from a list, the browser would send the user's selection back to the server. The server in turn would send back an updated web page with information on that illness. By learning the size of the page for each health condition, they could determine which conditions a user had without seeing any unencrypted data. Such vulnerabilities were found on several major websites, telling us that simply watching what we access, encrypted or not, can leak key information that we do not want public.

Hiding what file the target wants to access is referred to as oblivious simulation, a mechanism for hiding the real accesses of the target. Imagine a program that does not try to hide what locations are being accessed. The list of locations the program goes to is referred to as an access pattern. To provide oblivious simulation, the program changes the access pattern to hide which locations it really cares about. This is typically done by accessing many locations in addition to the one location the program really wants.

There are many applications where oblivious simulation is advantageous. In recent years, cloud computing has become increasingly popular. More frequently, companies are using cloud data services to store large amounts of data. The trust relationship with the outside world becomes a key consideration when storing and accessing data on the cloud. Simply encrypting the data might not provide enough security; if an outside party watches which sections of data are accessed and which actions follow, they may be able to discern information. Oblivious simulation can provide an added layer of security here, allowing companies to take advantage of the many services offered by the cloud.

Cloud computing is not the only application for oblivious simulation; it also can be used in designing hardware components. Increasingly secure hardware components are being used in various computing applications. For example, physically

secure CPUs may be used to perform sensitive actions that cannot be done on a regular desktop computer. Typically more than a CPU is required for most applications though. All of the needed components would need to be included in the secure hardware unit without oblivious simulation. With oblivious simulation and an encryption scheme, the CPU could use untrusted memory on a desktop computer to store information while it works.

Currently, there are two main solutions to the oblivious simulation problem: the square root solution and the hierarchical solution [3, 8]. Both solutions have some basic parts in common, but diverge from each other in key ways. The square root solution is based on the use of permutations. As the program runs, the memory is permuted at regular intervals to keep the attacker from knowing what really exists at each location. The hierarchical solution and its variations are based on a series of hash tables. As the program runs, the information is moved between the various tables, again preventing the attacker from knowing what really exists in each hash table.

Both solutions to the problem of oblivious simulation are in the relatively early years of their development. In this paper, we will build the background necessary for understanding both solutions, then analyze each solution in depth. We will provide formal proofs that both methods provide oblivious simulation, as well as a new variation on the square root solution. Additionally, we will discuss variations on the hierarchical solution already proposed.

1.1 Our Contributions

We will expand upon the existing proofs for both the square root solution and the hierarchical solution, adding more formality and depth. For the square root solution, we will also propose a new variation that significantly decreases the time overhead of simulation. Finally, we will give a formal security reduction showing that software protection can be given using a semantically secure encryption

scheme and oblivious simulation.

Chapter 2

PRELIMINARIES

We will be using several constructs from cryptography in this paper, ranging from types of security to random oracles. Before discussing the oblivious simulation problem, one must have an understanding of these topics. Additionally, we will review the Batcher sorting network as it is used heavily in this paper.

2.1 Encryption Scheme

An *encryption scheme* is a three-tuple of algorithms $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. $\mathcal{K}$ is the key generation algorithm, which as its name suggests, is used to create keys for using with the system. The key generation algorithm is randomized and we denote $K \xleftarrow{\$} \mathcal{K}$ for running the key generation algorithm which returns a key K. The set of all keys is denoted by Keys(Π). $\mathcal{E}$ is the encryption algorithm, which takes a specific key $K \in$ Keys(Π), along with a *plaintext* message $M \in \{0,1\}^*$ and produces a *ciphertext* $C \in \{0,1\}^* \cup \{\perp\}$. To denote running $\mathcal{E}$ with key K and message M, we write $C \xleftarrow{\$} \mathcal{E}_K(M)$. The ciphertext hides the original message so it is safe to send on an unsecured line. The final piece is the decryption algorithm $\mathcal{D}$, which like the encryption algorithm, takes a key $K \in$ Keys(Π) and a ciphertext $C \in \{0,1\}^*$. When the ciphertext is received, the decryption algorithm is applied to the key and the ciphertext. If decryption is successful, the original plaintext message $M \in \{0,1\}^*$ is returned, otherwise if decryption fails $\perp$ is returned. We denote running this algorithm as $M \leftarrow \mathcal{D}_K(C)$. In this paper we will be using *symmetric encryption schemes*, where both the sender and the receiver have the

same key which is used for both encryption and decryption.

2.2 Adversaries

An *adversary* is a theoretical entity that represents all of the various threats against an encryption scheme. Typically, this means learning any information about the messages being sent between the sender and receiver. The adversary can be restricted in certain ways or given certain abilities, depending on the type of security being tested. For example, the adversary may be computationally bounded and only allowed to run for so long. Or the adversary may be allowed to tamper with the messages being sent between the sender and receiver, instead of just watching the messages. The adversary may be allowed to adapt its behavior as the attack continues instead of sticking to a set plan of attack.

2.3 Oracles

In security proofs, adversaries and encryption schemes are commonly given access to an *oracle*. An oracle provides black-box access to some algorithm or function. The users of the oracle do not know what is done inside the oracle, but can query it and receive results from its computations. Oracle accesses are considered to take a single unit of time in most models of computation.

2.3.1 Random Oracles

The *random oracle* is an important theoretical construct in cryptography, used to generate random data. When the random oracle is queried with an input that it has not seen before, the output is selected uniformly at random from all possible outputs the oracle may return. If it sees that same input a second time, it will return the same output as before instead of choosing a new output. The random oracle can be used whenever one needs to generate truly random bits as part of a

security proof.

2.4 Indistinguishability Under Chosen-Plaintext Attacks

In the indistinguishability under chosen-plaintext attack security game (IND-CPA), the adversary is given access to a left-right encryption oracle (see Figure 2.2). The adversary is allowed to query the encryption oracle as many times as it wants, sending two messages, M_0 and M_1. If both messages are the same length, the oracle will encrypt one of the messages and return the ciphertext to the adversary. If the adversary is in World 0, M_0 will be encrypted; otherwise if the adversary is in World 1 then M_1 will be encrypted each time. This game is testing whether or not the adversary can learn any information about the plaintext given a ciphertext. If the adversary cannot do much better than guessing which world it is in, the encryption scheme informally is said to be indistinguishable under a chosen-plaintext attack.

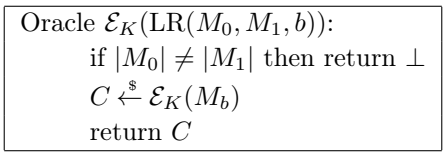

Figure 2.1: The IND-CPA Left-Right Encryption Oracle

More formally, the indistinguishability game for an encryption scheme $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ works as shown in Figure 2.2. The left-right function, denoted by $\mathrm{LR}(M_0, M_1, b)$, returns $\perp$ if M_0 and M_1 are not the same length. Otherwise if the bit b is 0, it returns $\mathcal{E}_K(M_0)$ and $\mathcal{E}_K(M_1)$ if b is 1. A bit b is chosen uniformly at random, as is a key K according to $\mathcal{K}$. Both are given to the oracle $\mathcal{E}_K(\mathrm{LR}(M_0, M_1, b))$ (see Figure 2.1). The oracle operates as follows when queries with (M_0, M_1): if both messages are not of the same length, the oracle returns $\perp$, otherwise the oracle returns $\mathcal{E}_K(M_b)$. If the adversary is in World 0, it is

given access to the oracle $\mathcal{E}_K(\mathrm{LR}(M_0, M_1, 0))$; if it is in World 1, it gets access to $\mathcal{E}_K(\mathrm{LR}(M_0, M_1, 1))$. The adversary may query the oracle as many times as it likes before guessing which world it is in. If it correctly guesses the world, it wins the game.

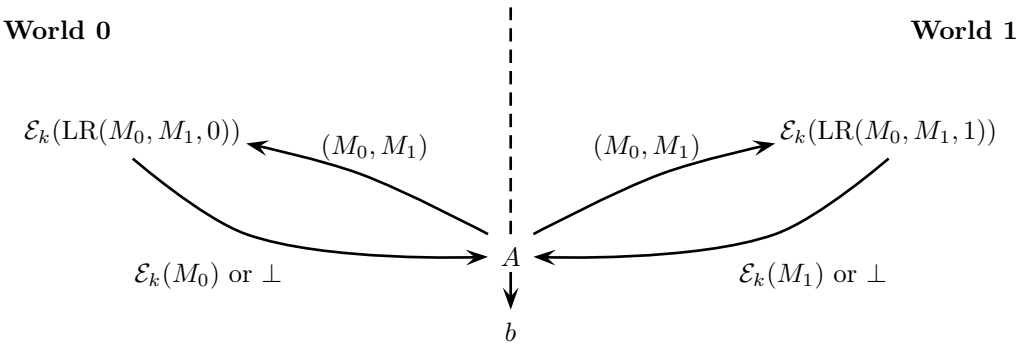

Figure 2.2: The IND-CPA Security Game

If we are in World 0, the instance of the IND-CPA game with adversary A and encryption scheme Π is called the World 0 experiment or $\mathbf{Exp}_{\Pi}^{\text{ind-cpa-0}}(A)$ (see Figure 2.3). In World 1, it is referred to as $\mathbf{Exp}_{\Pi}^{\text{ind-cpa-1}}(A)$ (see Figure 2.4).

Experiment $\mathbf{Exp}_{\Pi}^{\text{ind-cpa-0}}(A)$:
$K \xleftarrow{\$} \mathcal{K}$
$b \xleftarrow{\$} A^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,0))}$
return b

Figure 2.3: The IND-CPA Experiment in World 0

Experiment $\mathbf{Exp}_{\Pi}^{\text{ind-cpa-1}}(A)$:
$K \xleftarrow{\$} \mathcal{K}$
$b \xleftarrow{\$} A^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1))}$
return b

Figure 2.4: The IND-CPA Experiment in World 1

The advantage of the adversary (a measure of how well it does) is defined as the probability that it actually guesses it is in World 1 when it is in World 1 minus

the probability that it guesses it is in World 1 when it is really in World 0:

$$\mathbf{Adv}_{\Pi}^{\text{ind-cpa}}(A) = \Pr\left(\mathbf{Exp}_{\Pi}^{\text{ind-cpa-1}}(A) = 1\right) - \Pr\left(\mathbf{Exp}_{\Pi}^{\text{ind-cpa-0}}(A) = 1\right) \quad (2.1)$$

If the advantage is close to zero, that means the adversary is wrong about half the time about which world it is in, which is not much better than randomly guessing the world. However, if the advantage is close to one, that means it is almost always right about which world it is in and is a good adversary. For an encryption scheme to be considered IND-CPA secure, it should have a small advantage.

2.5 The Batcher Sort

A Batcher sorting network [1, 6] is a type of merge sort. Unlike a sorting algorithm, the steps taken by a sorting network are only determined by the length of the input. The Batcher sorting network begins by splitting the input into two parts recursively until only two values need to be compared, sorting those elements and then performing even-odd merges until the entire list is sorted. A program would perform the sort using the following procedures:

Procedure Sort(low**,** $high$**,** $list$**):**
    If $high - low \geq 1$
        $mid = low + \frac{high-low}{2}$
        Sort(low , mid, $list$)
        Sort($mid + 1$, $high$, $list$)
        Merge(low, $high$, 1, $list$)

Procedure Merge(low**,** $high$**,** $step$**,** $list$**):**
    $newStep = 2 * step$
    If $newStep < high - low$
        Merge(low, $high$, $newStep$, $list$)
        Merge($low + step$, $high$, $newStep$, $list$)
        For $i = \{low + step, low + step + newStep, low + step + 2 * newStep, \ldots,$
        $high - step\}$:
            Order(i, $i + step$, $list$)
    Else
        Order(i, $i + step$, $list$)

Procedure Order(x, y, $list$)**:**

```
If list[x] > list[y]
        tmp = list[y]
        list[y] = list[x]
        list[x] = tmp
Else
        list[x] = list[x]
        list[y] = list[y]
```

As you can see from the pseudocode, the only time the value is considered is when **Order** is called, but the same addresses are accessed regardless of the values. Since the comparisons do not depend on the values, the same steps are done each time the algorithm is called on lists of the same length.[1]

Now that we have constructed the basic cryptographic principles, we can begin to develop the necessary notions for the oblivious RAM problem and its associated concepts.

[1]For the correctness of the algorithm, refer to [1].

Chapter 3

THE OBLIVIOUS RAM PROBLEM

Before analyzing the various solutions for the oblivious RAM problem, it is important to have a firm grasp of exactly what the problem entails. We will begin this chapter by defining the RAM model and how it operates. Ostrovsky takes the standard RAM model and redefines it to be two interconnected Turing machines [8]. Once that is understood, oblivious simulation (also called the oblivious RAM problem) will be defined as well as its associated security games.

3.1 The RAM Model

A *random access machine* (RAM) is the combination of two Turing machines. Specifically, we will be using *interactive Turing machines* (ITM). An ITM is a five-tape Turing machine, as depicted in Figure 3.1: a read-only input tape, a write-only output tape, a read/write work tape, a read-only communication tape and a write-only communication tape. ITMs are parametrized by the length of the work tape and the block size of the communication tape. We denote this as $ITM_{(c,w)}$ where c is the block size of the communication tape and w is the length of the work tape. To start the ITM on some input x, which we denote by $ITM_{(c,w)}(x)$, the input is written to the read-only input tape of the ITM. The ITM copies the input to the first $|x|$ positions on its work tape (or halts if the input is longer than the work tape) and then works in rounds. The round begins by reading the next block from the read-only communication tape. The ITM does any computations needed on the work tape, then writes a block to the write-only

communication tape. When the ITM has finished execution, it copies the final results to the output tape. Execution can be halted by one of two ways; either a special symbol telling the ITM to halt was read off of the work tape or the computations performed by the ITM signaled that it should halt.

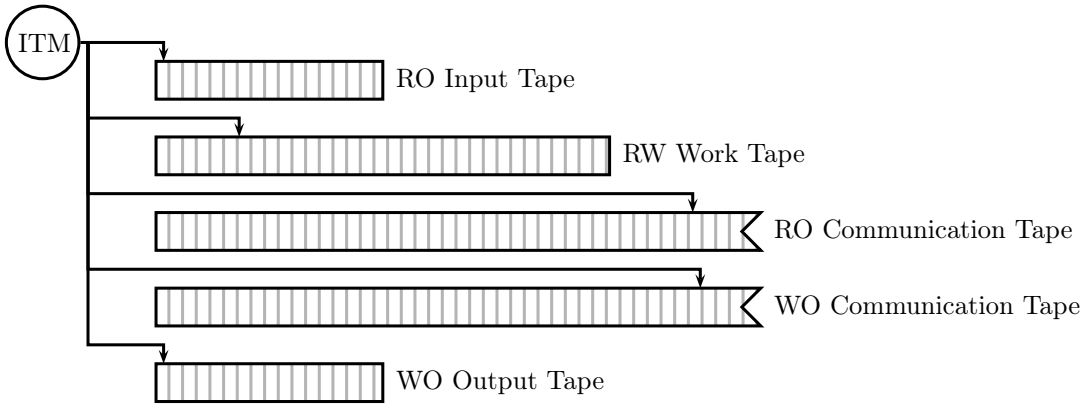

Figure 3.1: The ITM Model

The RAM combines two ITMs, one referred to as the CPU and the other as MEM. The CPU has a limited amount of storage, similar to the registers on a real CPU. The MEM on the other hand is like the memory in a computer; compared to the amount storage on the CPU there is quite a bit. In the RAM, the read-only communication tape of the CPU is connected to the write-only communication tape of the RAM and vice versa (Figure 3.2).

Let us begin by defining how the MEM ITM works. We denote MEM_k for $ITM_{(k,2^k k)}$, $k \in \mathbb{N}$, with the work tape divided into 2^k words of length k. The first word on the work tape is associated with an address of one, the second word is addressed with two and so on for all 2^k words. When MEM_k is started on some input x, denoted by $MEM_k(x)$, the input is copied to its work tape like an ITM normally does. From then on the MEM is message driven. Messages are of the form (a, i, v), where $a \in \{0,1\}^k$ is an address on the work tape, i is an instruction and $v \in \{0,1\}^k$ is a value. Three different instructions can be sent to the MEM:

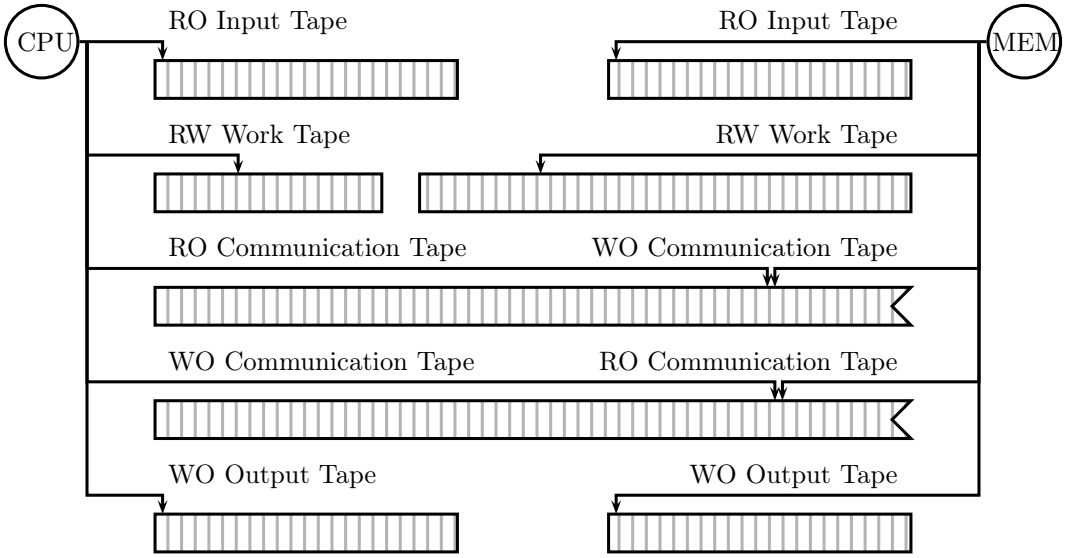

Figure 3.2: The RAM Model

read, write and halt. If the read instruction is sent, the value sent is ignored and the MEM writes the value currently stored at address a on its work tape to the communication tape. Otherwise, if the write instruction is sent, the value v sent by the CPU is written to word a on the MEM's work tape and to the communication tape. The final instruction is halt. When the MEM receives this instruction it copies its work tape to its output tape until a special symbol is hit.

The CPU ITM is where all of the work is done in the RAM model. Here we denote CPU_k for $ITM_{(k,k)}$, $k \in \mathbb{N}$. When the CPU_k is started on some input x, again denoted by $CPU_k(x)$, the input is copied from the input tape to the work tape. The CPU performs some calculations based on this input and then sends a message of the same form that the MEM receives: (a, i, v), where $a \in \{0,1\}^k$ is an address, $i \in \{read, write, halt\}$ is an instruction and $v \in \{0,1\}^k$ is a value. Like the MEM, the CPU is driven only by messages from this point on. After sending a message, with the exception of sending the halt instruction, the CPU waits to receive a message back which will be a value $v \in \{0,1\}^k$. The message is copied to the work tape and the CPU performs another set of calculations and sends another

message. When the CPU sends the halt message, it halts itself with no output.

Formally, we define a family of types $RAM_k = (CPU_k, MEM_k)$, for $k \in \mathbb{N}$, where $CPU_k = ITM_{(k,k)}$ and $MEM_k = ITM_{(k,2^k k)}$. When operating an instance of RAM_k, it is started on some input (s, y) where s is the input for CPU_k and y is the input to the MEM_k. The CPU_k and MEM_k alternate rounds with the CPU_k running first until the CPU_k halts. The output from RAM_k is the output of $MEM_k(y)$ when interacting with $CPU_k(s)$.

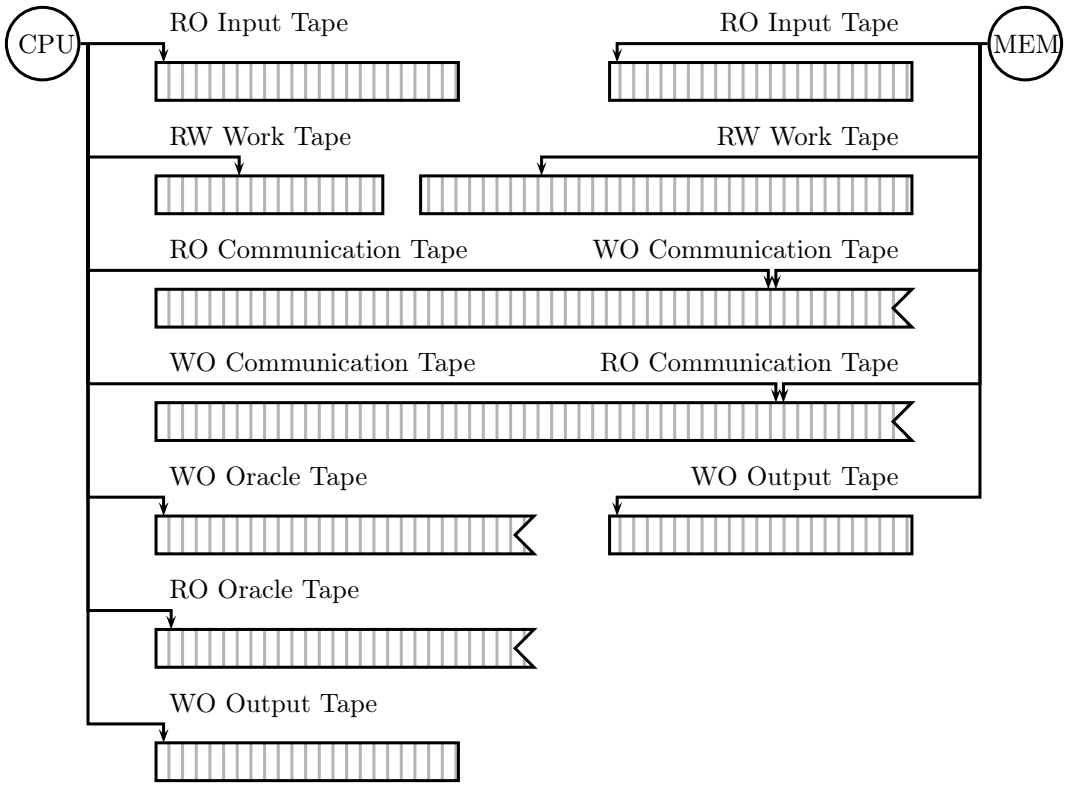

Figure 3.3: The Oracle-RAM Model

A simple variation of the RAM model is the oracle-RAM (Figure 3.3). Here, the CPU is given access to two extra tapes which are used to access a random oracle: a read-only oracle tape and a write-only oracle tape. When the CPU writes a query to the write-only oracle tape, the CPU is moved into a special oracle invocation state. In a single step the oracle writes the output corresponding to the input

queried to the read-only oracle tape. This model is commonly used to simulate a RAM that is given access to an encryption scheme.

3.2 Oblivious RAMs

Some RAMs may have the property of being *oblivious*; if an adversary is watching the messages between the CPU and the MEM, it cannot tell what memory the CPU is really trying to access. This is formalized using the concept of *access patterns*. An access pattern for a deterministic RAM_k, $k \in \mathbb{N}$, on input (s, y) is the list of memory locations accessed by the CPU, denoted by $A^k(s, y) = (a_1, a_2, \ldots, a_i, \ldots)$ where the i^{th} message from the CPU_k to the MEM_k is $(\cdot, a_i, \cdot)$. For an oracle-RAM_k on input (s, y) we use the random variable $\tilde{A}^k(s, y)$ to represent the access pattern. $\tilde{A}^k(s, y)$ assumes a specific value for RAM_k given a uniformly selected random oracle on input (s, y). We will use the function RO to refer to this instance of the oracle. An oracle-RAM_k is said to be oblivious if for all inputs (s_1, y_1) and (s_2, y_2), if $|\tilde{A}^k(s_1, y_1)|$ and $|\tilde{A}^k(s_2, y_2)|$ are equally distributed then so are $\tilde{A}^k(s_1, y_1)$ and $\tilde{A}^k(s_2, y_2)$. In other words, if the length of the access patterns are equally distributed, the access patterns are indistinguishable (this is similar to the concept of indistinguishability against chosen plaintext attacks for encryption schemes).

The oblivious RAM problem only worries about hiding the access pattern. Using an encryption scheme that is IND-CPA secure to encrypt the values being stored before sending them to the MEM prevents the values being stored from leaking information (see Section A). Since we know the values cannot leak information, we only need to concern ourselves with checking if the access pattern leaks information when evaluating solutions to the oblivious RAM problem.

An oracle-RAM is said to *obliviously simulate* a program if no information about the access pattern of the program running on a deterministic RAM can be learned from the access pattern when it runs on the oracle-RAM. More formally, given an oracle-RAM_k and a deterministic $RAM'_{k'}$, RAM_k obliviously simulates

$RAM'_{k'}$ if these three conditions hold:

- RAM_k is oblivious
- $\forall(s, y)$, $\forall RO$, the output of $RAM_k(s, y)$ when given access to RO in the form of a random oracle is the same as $RAM'_{k'}(s, y)$
- The running time of $RAM_k(s, y)$ is determined by $RAM'_{k'}(s, y)$

3.2.1 Oblivious RAM Security Games

Constructing a formal security game for the oblivious RAM problem is made difficult by one key requirement in the problem: the two access patterns in question, $\tilde{A}^k(s_0, y_0)$ and $\tilde{A}^k(s_1, y_1)$, must have lengths that are equally distributed. However, the length of the access patterns is a random variable and there may not be a way to know if the access patterns generated by two different inputs are equally distributed. In the security game, the adversary must at minimum be given access to an access pattern oracle in a similar manner to the encryption oracle in the IND-CPA security game (Section 2.4). The access pattern oracle would be presented with two inputs to the RAM_k, (s_0, y_0) and (s_1, y_1), and using the left-right function $LR(\cdot, \cdot, \cdot)$, choose one of the inputs and return the access pattern for that input if the lengths are equally distributed. Otherwise, if the lengths of the access patterns are not equally distributed, $\perp$ is returned. The question is how does the adversary and the access pattern oracle determine that the lengths are equally distributed.

The naive way is to have the access pattern oracle run the RAM on both inputs. If the access patterns generated are not of the same length, the oracle would return $\perp$. However, the adversary would have no way of knowing in advance if the access patterns are of the same length and this would unfairly penalize it (unlike the IND-CPA game where the adversary decides whether or not the messages are of the same length).

The Naive Oblivious RAM Security Game

The naive oblivious RAM security game tests whether or not an adversary can distinguish between access patterns (Figure 3.5). To set up the ORAM game, a function RO is chosen uniformly at random and is given to RAM_k as a random oracle. Additionally, a bit b is chosen uniformly at random. The adversary A sends pairs of inputs $((s_0, y_0), (s_1, y_1))$ to the access pattern oracle AP (see Figure 3.4). If $|\tilde{A}^k(s_0, y_0)| \neq |\tilde{A}^k(s_1, y_1)|$, $\perp$ is returned. Otherwise $\tilde{A}^k(s_b, y_b)$ is returned. A is allowed to query AP as many times as it wants but is only allowed to run for $2^{O(k)}$ time. When A halts, it outputs a bit b'. If $b' = b$, then A wins.

```
Oracle AP(RAM_k, LR((s_0, y_0), (s_1, y_1), b)):
    if |Ã^k(s_0, y_0)| ≠ |Ã^k(s_1, y_1)|
        return ⊥
    run RAM_k(s_b, y_b)
    return Ã^k(s_b, y_b)
```

Figure 3.4: The Access Pattern Oracle

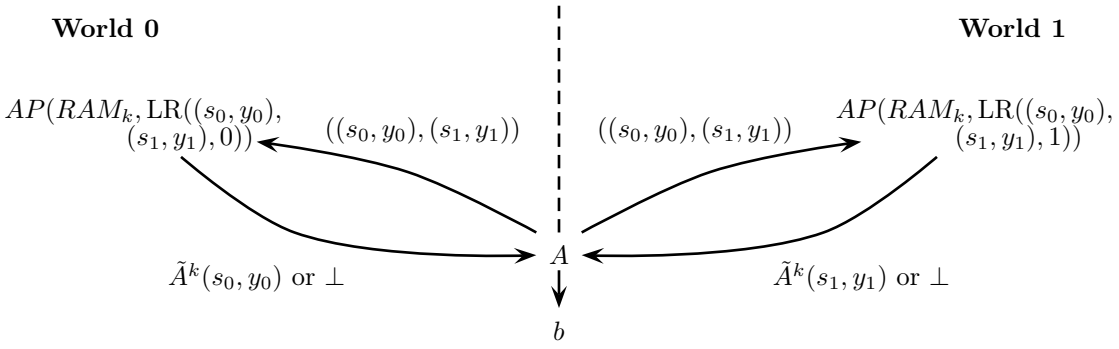

Figure 3.5: The Naive Oblivious RAM Security Game

If we are in World 0, the instance of the ORAM game with adversary A and encryption scheme RAM_k is called the World 0 experiment or $\mathbf{Exp}^{\text{oram-0}}_{RAM_k}(A)$ (Figure 3.6) and $\mathbf{Exp}^{\text{oram-1}}_{RAM_k}(A)$ in World 1 (Figure 3.7). Like the IND-CPA security game (Section 2.4), the advantage of the adversary is defined as the probability that it

actually guesses it is in World 1 when it is in World 1 minus the probability that it guesses it is in World 1 when it is really in World 0:

$$\mathbf{Adv}^{\text{oram}}_{RAM_k}(A) = \Pr\left(\mathbf{Exp}^{\text{oram-1}}_{RAM_k}(A) = 1\right) - \Pr\left(\mathbf{Exp}^{\text{oram-0}}_{RAM_k}(A) = 1\right) \tag{3.1}$$

Again, for a RAM_k to be considered oblivious, the advantage should be close to 0, indicating that the best adversary cannot do better than randomly guessing which world it is in.

Experiment $\mathbf{Exp}^{\text{oram-0}}_{RAM_k}(A)$:
$b \xleftarrow{\$} A^{AP(RAM_k,\text{LR}((\cdot,\cdot),(\cdot,\cdot),0))}$
return b

Figure 3.6: The Naive ORAM Experiment in World 0

Experiment $\mathbf{Exp}^{\text{oram-1}}_{RAM_k}(A)$:
$b \xleftarrow{\$} A^{AP(RAM_k,\text{LR}((\cdot,\cdot),(\cdot,\cdot),1))}$
return b

Figure 3.7: The Naive ORAM Experiment in World 1

Oblivious RAMs with a Deterministic Increase in Access Pattern Length

There is a special exception to the problem of knowing the distribution of the access pattern; specifically, if the access pattern generated by the oblivious RAM is a constant factor larger than the original access pattern. In fact, all of the solutions we will discuss in this paper increase the access pattern length by a constant multiple. In this case, the adversary can determine how long the access pattern of the oblivious RAM should be so we have no need to avoid penalizing the adversary for bad queries to the left-right oracle. For these solutions, the naive ORAM experiment is sufficient (Section 3.2.1).

3.3 A Related Problem: The Outsourced Data Model

A similar problem to that of oblivious RAM simulation is the *outsourced data model* [4]. In this model, the client has a large amount of data they want to store, say of size n, and they have purchased this amount of space from a server provider (such as a cloud computing service). Like the RAM model, the data on the server is indexed and the client will make indexed queries to the memory. On their local, trusted computers the client only has a fraction of this memory, on the order of $O(n^{\frac{1}{r}})$ for some constant $r > 1$. The server provider is potentially malicious so the client does not fully trust them. Perhaps the server provider has some financial interest in learning about the client's data or perhaps an employee is stealing information to sell to the client's competitors. In this case, simply encrypting information may not hide enough information. As we have seen, information can still be leaked despite using encryption [2]. The client would want their accesses to be *data oblivious* to prevent these side channel attacks.

A computation is said to be data oblivious, if for two unique *memory configurations* of the same size, an access pattern is equally likely to be seen with each configuration. A memory configuration is the physical layout of the memory as well as the values currently stored at each location. More formally, for memory configurations M and M', where $M \neq M'$ and $|M| = |M'|$, and an access sequence S, the probability that we see S given M is the same that we see S given M':

$$\Pr(S|M) = \Pr(S|M')$$

The property of data obliviousness is as general as the oblivious RAM problem; solutions meeting either definition will only leak information about the running time and amount of memory used. Either of these pieces could also be obscured by padding either the memory or the execution with fake data and additional computations. Unlike the oblivious RAM problem, this solution does not include

the requirement that the access patterns be equally distributed, removing one of the more significant complications in proving that a solution provides oblivious simulation. Going forward, one suspects that this will the more useful definition.

Chapter 4

THE SQUARE ROOT SOLUTION

Intuitively, it is trivial to hide the actual memory locations a program wants to access by having the RAM read and then write every item in memory for each access in the original program. Because we access all of the locations each time, an adversary could not determine which location we were truly interested in. Unfortunately, this is not the best solution because it has a large overhead: if m words of memory are required by the original RAM, the oblivious RAM requires $2m$ accesses per original access. Instead of scanning everything, in the square root solution, a relatively small number of locations are scanned, along with regular reshuffling of the words in memory. This solution was originally proposed via two papers by Ostrovsky and Goldreich [3, 8]. We we detail and analyze the square root solution in this section, as well as propose our own modifications.

We know that this solution must be based on the use of an oracle-RAM. Recall from our discussion of the oblivious RAM problem (Section 3.2) that an adversary would query the access pattern oracle with two different inputs $((s_0, y_0), (s_1, y_1))$. The oracle would choose one of these inputs, execute the RAM on it and return the resulting access pattern. Suppose that we used a deterministic RAM instead of an oracle-RAM for the solution. If the adversary sent the same (s_0, y_0) each time and a different (s_1, y_1) it could determine with high probability which world it was in. If the adversary was in World 0, the oracle would always execute the RAM on (s_0, y_0). Since the RAM is deterministic, this would generate the same access pattern each time. If the adversary was in World 1, then it would expect to see a different access pattern each time. This tells us that we need to use an

oracle-RAM to implement our solution.

The oblivious oracle-RAM_k is given access to some extra memory besides the m words used by the original $RAM'_{k'}$ (see Figure 4.1). Our oblivious oracle-RAM_k should provide the same functionality as the original $RAM'_{k'}$ (in other words, for a given input, both RAMs should return the same output). The extra memory is divided into two parts; half are referred to as *dummy locations* and the other half as the *shelter*. Specifically, $\sqrt{m}$ dummy locations and $\sqrt{m}$ shelter locations are added. The dummy and shelter words are used to hide whether or not RAM_k has accessed the same word more than once in a pass (more on this shortly). The first m words in RAM_k's memory are initialized to contain the same values as the m words in $RAM'_{k'}$'s memory. Specifically, the words in RAM_k are of the form (i, v_i), where v_i is the value originally stored at word i in $RAM'_{k'}$. The dummy locations are words in memory that are initialized to a dummy value and are not used by the program being simulated. The shelter locations are used to temporarily hold the words that were accessed in one pass of the oblivious simulation. The shelter locations are also initialized to a dummy value.

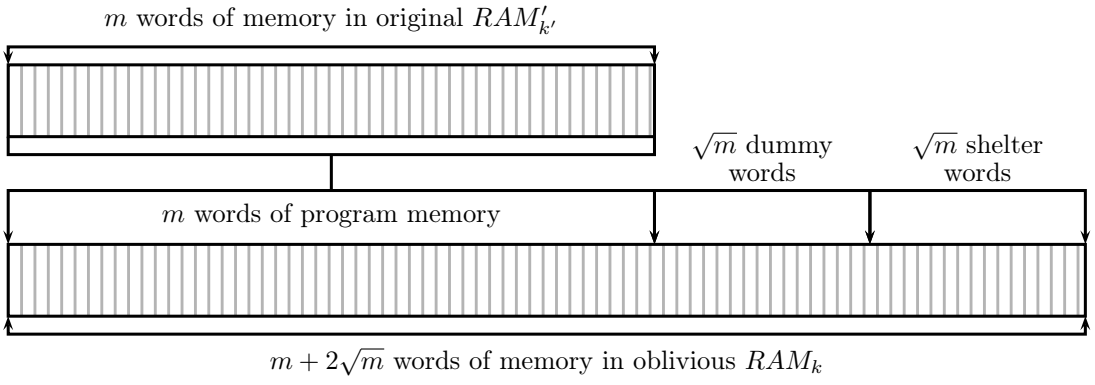

Figure 4.1: Memory Setup for RAM_k

The square root solution works in cycles, simulating $\sqrt{m}$ memory accesses of the original $RAM'_{k'}$ at a time (we will call this a *pass*). We will refer to the oracle that RAM_k is given access to as RO. RAM_k at a high level works as follows for

pass g:

1. $count \leftarrow 0$
2. Construct a random permutation $\pi : [m + \sqrt{m}] \rightarrow [m + \sqrt{m}]$ using the function RO
3. Permute the first $m + \sqrt{m}$ words in RAM_k's memory according to π (e.g. if $\pi(3) = 6$, we would move $(3, v_3)$ to the sixth word in RAM_k)
4. For $j = h + 1, h + 2, \ldots, h + \sqrt{m}$, where $h = (g - 1)\sqrt{m}$, simulate memory access j of the original $RAM'_{k'}$ (the access will be of the form (i_j, a_j, v_j) where i_j is the instruction, a_j is an address and v_j is a value):
   (a) Scan shelter for the word that contains address a_j
   (b) If a_j is not found in the shelter, look for it at location $\pi(a_j)$ in one of the non-shelter locations
   (c) If a_j was found in the shelter, access a dummy location $\pi(m + count)$
   (d) Scan the shelter again, reading then writing back a reencrypted value for each word. If a_j was previously found in the shelter, update its value (if needed). Otherwise write (a_j, v_{a_j}) into the first empty shelter location
   (e) $count \leftarrow count + 1$
5. Put each of the program words back in their original location

Each of these steps is key to obtaining oblivious simulation of the original program. Recall from Section 3.2 that an IND-CPA secure scheme is used to encrypt the value being stored each time it is written, so the square root solution only needs to hide the access pattern and obscure whether a read or write is being done.

If the accesses made when the values stored in the non-shelter locations are permuted are independent of the data stored in the RAM_k and the values are

reencrypted at each step, then the adversary will not be able to tell which values end up in each location (steps 2 and 3). By scanning the shelter and then accessing only one non-shelter location (steps 4a, 4b and 4c), the algorithm obscures whether or not the same value has been accessed twice since the adversary, again due to encryption, cannot tell whether or not a dummy location or a program memory location has been accessed. Then the entire shelter is read and written back, obscuring which shelter location was written to (step 4d). Once the $\sqrt{m}$ accesses have been simulated, the program memory is returned to its original location (step 5), then repermutes for the next pass (back to steps 2 and 3).

In the remainder of this chapter we will go into more depth on how each of these steps is implemented and show that no information about the access pattern is leaked by this solution.

4.1 Overview

As described in the previous section, the $m + 2\sqrt{m}$ memory locations are divided into three parts. The first m words are program memory and are initialized to reflect the original state of the original $RAM'_{k'}$. Specifically, a tuple is written to each word: the work tape address i and the value v_i stored at that location (e.g. if the value 36 was originally stored in the seventh word of $RAM'_{k'}$ then $(7, 36)$ would be written to the seventh word of RAM_k). The next $\sqrt{m}$ words are dummy locations initialized to some arbitrary value, which we will denote d, and are used to obfuscate what is being accessed. The final $\sqrt{m}$ locations are called the shelter, again initialized to a dummy value. The shelter is used to store the values that have been accessed by the RAM_k in this pass of the algorithm. Before the first pass through the simulation, each word will need to be updated, so it is tagged with the address it is at. The exception to this is the shelter locations, which will be tagged with an address of ∞. The program words will look like (i, v_i), where i is the address and v_i is the value originally stored there. Dummy words will

be of the form (i, d) and shelter words will be (∞, d) (we will see why they need an address of ∞ shortly). An example of the initial work tape memory setup for RAM_k, based on an original $RAM'_{k'}$ with a work tape of length four, can be see in Figure 4.2.

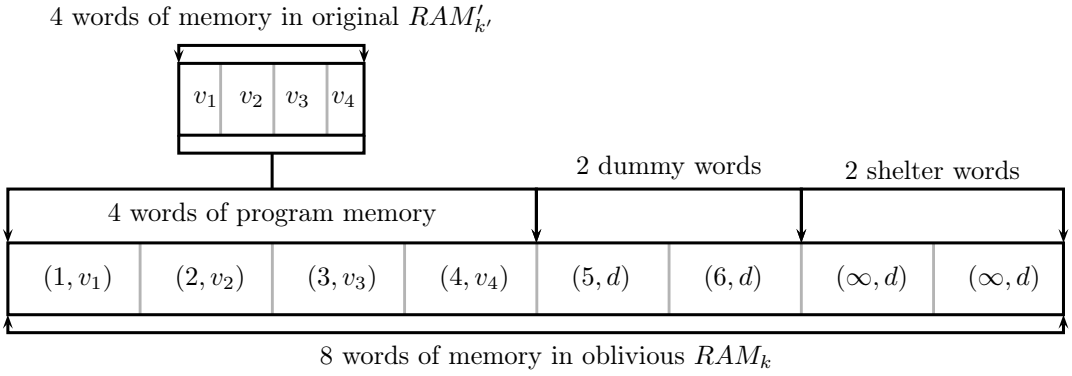

Figure 4.2: Initial State of the Work Tape for RAM_k

The simulation works in cycles, each cycle simulating $\sqrt{m}$ accesses of the original RAM. For the sake of simplicity, we will ignore the encryption as described in Section 3.2. It suffices to remember that each time a value is written, it is re-encrypted and the ciphertext is written so the value stored changes whether or not the value was actually updated.[1] Initially, the memory needs to be shuffled to hide which locations are truly being accessed. This must be done on every pass, including the first, or else the adversary could trivially distinguish between access patterns; without the initial shuffle, the first $\sqrt{m}$ accesses simulated would access the same locations as the original program. In the oblivious RAM security game (see Section 3.2.1), if the adversary simply queried the RAM oracle with the same (s_0, y_0) each time but a different (s_1, y_1), it could guess that it is in the correct world with high probability, simply by seeing if the first $\sqrt{m}$ simulated accesses

[1] An IND-CPA secure encryption scheme is probabilistic, stateful or both. This allows the encryption scheme to generate different ciphertexts each time the same plaintext is encrypted. See Section 2.4 for further details.

are the same for each returned access pattern. Additionally, choosing a new shuffle each time will put the program words in a different location each pass. This prevents the adversary from learning information by comparing the accesses made in each pass and prevents multiple passes from leaking information.

Both the program memory and dummy locations will need to be shuffled. If the dummy locations are not included in the shuffle, when the algorithm accesses dummy locations it will always access them in the same portion of memory (in step 4c above). This would leak whether or not the program accessed the same location multiple times in one pass, again leading to an adversary who could distinguish between access patterns with high probability. Leaking this information may not be bad in practice (though it has the potential) but does violate the oblivious RAM security game (Section 3.2.1).

In order to shuffle the memory, the RAM constructs a uniformly random permutation using the random oracle. Let n be the number of items that need to be permuted (in this case $n = m + \sqrt{m}$). The random oracle can be used to construct a function $f : [n] \to \mathbb{Z}_{n^{\log n}}$ such that, with high probability, $f(i)$ is distinct for all $i \in [n]$. The choice of $\mathbb{Z}_{n^{\log n}}$ for the range reduces the probability of a collision, as we will show later, while minimizing the number of bits needed to store the value computed by f on a particular input. The permutation $\pi(\cdot)$ can be constructed as follows: $\pi(i) = j$ if and only if $f(i)$ is the j^{th} smallest element in $\{f(k) \mid 1 \leq k \leq n\}$ (we will explain in Section 4.2.1 why this works). After each pass through the simulation a new function f will need to be constructed. After f has been constructed, each non-shelter memory location $1 \leq i \leq m + \sqrt{m}$ is accessed and tagged with the value $f(i)$. Each non-shelter location now stores a tuple of the tag, virtual address and value $(f(i), i, v_i)$, where v_i is the value currently stored at virtual address i. When scanning for an address a_j, RAM_k will be looking at the second value in the tuple (the virtual address). The algorithm will need to know later which $f(i)$ and i corresponds to the value being read; the only way for it to know this information

is for each location to store the tuple.

Once all of the memory locations have been tagged, they will need to be sorted according to the value of $f(\cdot)$ in order to permute them according to $\pi(\cdot)$. The sorting is done via a Batcher Sorting Network (see Section 2.5) [1]. The Batcher sort is a type of merge sort which makes $n \cdot \lceil \log_2 n \rceil^2$ comparisons when sorting n elements. Crucially, the same comparisons are done each time, regardless of the items being sorted.

Now that the memory locations have been permuted, RAM_k will begin simulating the $\sqrt{m}$ accesses of the original $RAM'_{k'}$ for this pass through the algorithm. RAM_k keeps a counter, which we will call *count*, which is initialized to 0. The i^{th} access will be for some program address a_i. To simulate accessing a_i, RAM_k will first scan the shelter to see if a_i has already been accessed. If program address a_i was not found in the shelter, we will need to find it in a non-shelter location. We do not know which physical word in RAM_k stores a_i but we do know $f(a_i)$. To find a_i, we perform a binary search over the first $m + \sqrt{m}$ words, looking for the word tagged with $f(a_i)$. Once we find a_i at some physical word j, we rewrite the value stored so that the program address is updated to infinity. In other words, the tuple $(f(a_i), \infty, v_{a_i})$ is written back to the same physical word j where we found a_i (we will see why later). If we found program address a_i in the shelter, then we need to search for one of the dummy words so the access pattern looks the same regardless of where we find a_i. Each time we access a dummy location we need it to be a unique dummy location that we have not accessed before in this pass. This is where the counter *count* comes into play. We perform the binary search, this time looking for the $count^{\text{th}}$ dummy word, which is tagged with $f(m + count)$. When we find the dummy word at some address j, we re-encrypt it and rewrite the same value back to the same location. Finally, we scan the shelter again, reading and then writing each word in place. If a_i was previously found in the shelter, when we get to that word it is rewritten with the potentially updated value for v_{a_i}. If

a_i was not in the shelter already, it is written to the first empty word found in the shelter (the first word tagged with an address of ∞). Once this scan of the shelter is complete, *count* is incremented by 1 and the next memory access is simulated.

Once all of the $\sqrt{m}$ access in this pass have been simulated, we need to get all of the words back to their original position. This is for one of two reasons. In the first case, if this is not the last pass through the algorithm, we will need to repermute the words to obscure the next $\sqrt{m}$ simulated accesses, as well as clear the shelter. By returning all of the words to their original location, the shelter will no longer contain program words, making the shelter ready for the next pass. The program words and dummy words are back in the first $m + \sqrt{m}$ locations, ready to be shuffled according to a new permutation. In the second case, if this is our last pass, when the CPU portion of RAM_k sends the halt message to the MEM, the MEM prints out its work tape until it hits a special symbol (see Section 3.1). If we have not returned the words to their original location, RAM_k will output something different than what the original $RAM'_{k'}$ returned, and thus would violate the operational requirements of oblivious simulation.

Before we describe how the words are returned to their original location, we should assure ourselves that each program word and each dummy word only exist in one location. If a program word was not accessed in this pass, then it could trivially only be in one location since it was never written. Specifically, it will be somewhere in the first $m+\sqrt{m}$ words, as determined by our permutation $\pi(\cdot)$. If we did access some program word a_i in this pass, the first time we accessed it we found it in some non-shelter location at physical address j. After we read the word, we rewrote the tuple at address j and updated the program address to ∞ (originally $(f(a_i), a_i, v_{a_i})$ would be stored at address j, after the update $(f(a_i), \infty, v_{a_i})$ would be stored there), and then wrote the tuple for a_i into the shelter. If a_i was accessed subsequently, the value was updated in the shelter location, so the word tagged with program address a_i only exists in one location. The dummy words only exist

in one location, too, since we never moved them into the shelter. Finally, the unused shelter locations will be tagged with an address of ∞ so we can safely ignore them.

To return the words to their original location, we need to do a Batcher sort across all $m + 2\sqrt{m}$ words based on the address they are tagged with (a Batcher sort is used because is it deterministic; see Section 2.5). The program words will end up in the first m words since they are tagged with addresses 1 to m. The dummy words, tagged with addresses $m + 1$ to $m + \sqrt{m}$ will end up in the next $\sqrt{m}$ words. Finally the shelter words, all tagged with an address of ∞ because they were either words accessed during this pass or are unused shelter locations, will be in the last $\sqrt{m}$ words. Now that everything has been returned to its original position, RAM_k can halt if there are no more steps to simulate, or the next $\sqrt{m}$ steps can be simulated.

Initial State

$(1, 18)$	$(2, 13)$	$(3, 42)$	$(4, 6)$	$(5, d)$	$(6, d)$	(∞, d)	(∞, d)

Each Non-Shelter Value is Tagged with $f(\cdot)$ where $f : [6] \to [24]$

$(16, 1, 18)$	$(21, 2, 13)$	$(9, 3, 42)$	$(11, 4, 6)$	$(7, 5, d)$	$(13, 6, d)$	(∞, d)	(∞, d)

A Batcher Sort is Performed Across Non-Shelter Words Based on the $f(\cdot)$ Tag

$(7, 5, d)$	$(9, 3, 42)$	$(11, 4, 6)$	$(13, 6, d)$	$(16, 1, 18)$	$(21, 2, 13)$	(∞, d)	(∞, d)

Program Location 3 is Accessed and Moved into the Shelter

$(7, 5, d)$	$(9, \infty, 42)$	$(11, 4, 6)$	$(13, 6, d)$	$(16, 1, 18)$	$(21, 2, 13)$	$(9, 3, 26)$	(∞, d)

Program Location 1 is Accessed and Moved into the Shelter

$(7, 5, d)$	$(9, \infty, 42)$	$(11, 4, 6)$	$(13, 6, d)$	$(16, \infty, 18)$	$(21, 2, 13)$	$(9, 3, 26)$	$(16, 1, 5)$

A Batcher Sort is Performed Across All Words Based on Physical Address

$(16, 1, 5)$	$(21, 2, 13)$	$(9, 3, 26)$	$(11, 4, 6)$	$(7, 5, d)$	$(13, 6, d)$	$(9, \infty, 42)$	$(16, \infty, 18)$

Figure 4.3: A Sample Round

In summary, the simulation works as follows for pass j (a sample round[2] can be seen in Figure 4.3):

1. Construct $f : [n] \rightarrow \left[n^{\log n}\right]$ where $n = m + \sqrt{m}$.
2. For $1 \leq i \leq m + \sqrt{m}$, tag each word with $f(i)$. Each non-shelter word will now be of the form $(f(i), i, v_i)$.
3. Perform a Batcher sort across all non-shelter locations based on the value of $f(\cdot)$.
4. Initialize *count* to 0.
5. For $i = (j-1)\sqrt{m} + 1, \ldots, (j-1)\sqrt{m} + \sqrt{m}$, simulate access i for program address a_i:
   (a) Scan shelter for address a_i
   (b) If a_i is not found in the shelter, do a binary search over the non-shelter locations looking for the word tagged with $f(a_i)$. When a_i is found, rewrite the value so the virtual address is now ∞.
   (c) If a_i was found in the shelter, do a binary search over the non-shelter locations looking for dummy word $m+$*count* which is tagged with $f(m+$ *count*$)$. Re-encrypt and rewrite the word when found.
   (d) Scan the shelter, rewriting each value after it is read. If a_i is already in the shelter, update the value if needed. Otherwise put the tuple for a_i in the first empty shelter location.
   (e) Increment *count* by 1.
6. Perform a Batcher sort based on the address each word is tagged with across all words in memory to return everything back to its original location

[2]See appendix B for more detailed information

4.2 Proving the Square Root Solution is Oblivious

We will build our proof that the square root solution provides oblivious simulation in pieces. First we will show that $\pi(\cdot)$ is a uniformly random permutation. Next we will show that the binary searches do not add any information about the memory being accessed. Finally, we will construct the access pattern and show that one pass through the simulation does not leak any information besides the running time and amount of memory used, nor do multiple passes through the simulation.

4.2.1 $\pi(\cdot)$ is a Uniformly Random Permutation

Before we show that $\pi(\cdot)$ is a random permutation, let us review how it is constructed. Given $n = m + \sqrt{m}$, some constant c_1 on the order of 2^k and another constant $c_2 > 0$, our random oracle will be chosen from the set of functions that map from $[c_1 n]$ to $\mathbb{Z}_{c_2 n^{\log n}}$:

$$RO \in \{g(\cdot) \mid g : [c_1 n] \to \mathbb{Z}_{c_2 n^{\log n}}\}$$

The specific random oracle that we get will be chosen uniformly at random from this set when the security game is set up. We do not have any control over the exact oracle that is selected. The domain for the random oracle is sufficiently large so that we do not query the random oracle with the same input twice. Since the adversary in the ORAM game cannot run for longer than $O(2^k)$ time (Section 3.2.1), then c_1 on the order of 2^k is sufficiently large. Recall that the function f that we use to construct $\pi(\cdot)$ maps $[n]$ to $\mathbb{Z}_{n^{\log n}}$ (Section 4.1). For some input to f, we want it equally likely to map to any point in $\mathbb{Z}_{n^{\log n}}$. For any given input, the random oracle will output a uniformly random point in its range. If the range of f evenly divides the range of the random oracle, as we have chosen, then we can construct such an f.

The function $f : [n] \to \mathbb{Z}_{n^{\log n}}$ (in this case $n = m + \sqrt{m}$) is constructed using

the random oracle, and we will need to construct a unique f for each pass through the simulation. To do so, the RAM needs to keep an internal counter, which we will call *passes* and initialize to 0. After each pass through the simulation *passes* will be incremented by 1. For each pass, $f(\cdot)$ is created as follows for $i = 1, \ldots, n$:

$$f(i) = RO(passes \cdot n + i) \pmod{n^{\log n}}$$

Now that we have constructed $f(\cdot)$, we can create our permutation $\pi : [n] \to [n]$ as follows: $\pi(i) = j$ if and only if $f(i)$ is the j^{th} smallest value in $\{f(1), f(2), \ldots, f(n)\}$, assuming there are no collisions. We need to show that $\pi(\cdot)$ is a uniformly random permutation. Assume that $\pi(\cdot)$ is not a uniformly random permutation. This means there exists some value $i \in [n]$ such that $\Pr(\pi(i) = j) > \frac{1}{n}$ for some j. Since $\pi(\cdot)$ is created using $f(\cdot)$, this equivalently means that the probability that $f(i)$ is the j^{th} smallest number in $\{f(1), f(2), \ldots, f(n)\}$ is greater than $\frac{1}{n}$. This means that $f(i)$ is weighted toward some subset of $\mathbb{Z}_{n^{\log n}}$, which contains at least one element l. In other words $\Pr(f(i) = l) > \frac{1}{n^{\log n}}$. However we have already shown that $\Pr(f(i) = l) = \frac{1}{n^{\log n}}$, which is a contradiction, therefore π must be a uniformly random permutation.

Finally, we just need to show that the chance of collisions in $f(\cdot)$ is small. First, let us consider the probability that $f(\cdot)$ has no collisions. Let $P(n, k) = \frac{n!}{(n-k)!}$ be the number of ways to choose an ordered subset of k items from a set of n items. If there are no collisions, each of the n items in the domain must map to a unique element in $\mathbb{Z}_{n^{\log n}}$. This is the same as choosing n items from a set of $n^{\log n}$ items, giving us $P\left(n^{\log n}, n\right)$ possible mappings with no collisions. If we allow collisions, we have $\left(n^{\log n}\right)^n$ possible mappings for $f(\cdot)$, giving us a probability of $\frac{P\left(n^{\log n}, n\right)}{\left(n^{\log n}\right)^n}$ that there are no collisions. Therefore the chance that we do have a collision is:

$$\Pr(collision) = 1 - \frac{P\left(n^{\log n}, n\right)}{\left(n^{\log n}\right)^n}$$

We know that for $n \gg k$, $P(n,k) \approx n^k$. For some large n, $n^{\log n} \gg n$, which tells us that $P\left(n^{\log n}, n\right) \approx \left(n^{\log n}\right)^n$. With this, our probability of a collision can be changed as follows:

$$\begin{aligned}
\Pr\left(collision\right) &= 1 - \frac{P\left(n^{\log n}, n\right)}{\left(n^{\log n}\right)^n} \\
&\approx 1 - \frac{\left(n^{\log n}\right)^n}{\left(n^{\log n}\right)^n} \\
&\approx 1 - 1 \\
&\approx 0
\end{aligned}$$

Therefore, for large n, the probability that we have a collision in $f(\cdot)$ is close to zero.

4.2.2 The Binary Search Does Not Leak Information

Next we will show that the binary searches performed do not leak any information besides the physical word they lead to and do not give any indication about the virtual address stored in that location. This allows us to reduce the binary search access pattern to simply the word it leads to. The binary searches are done over the first $m + \sqrt{m}$ words. A uniformly random permutation $\pi(\cdot)$ was constructed using $f(\cdot)$. We can view the binary search as paths through the binary tree constructed by $\pi(\cdot)$, where $\pi(i) = \frac{m+\sqrt{m}}{2}$ is the root, $\pi(j) = \frac{m+\sqrt{m}}{4}$ and $\pi(k) = \frac{3(m+\sqrt{m})}{4}$ are the children of the root and so on. The words accessed by the binary search for some word $\pi(l)$ is the same as the path through the binary tree from the root to the node containing $\pi(l)$ [6].

Suppose we construct an adversary that given a value $\pi(i)$ tries to guess the value i. Since $\pi(\cdot)$ is a uniformly random permutation, the adversary cannot do better than guessing the value of i and each possible value has the same chance of being right, in this case $\frac{1}{m+\sqrt{m}}$. Now suppose that instead of permuting using $\pi(\cdot)$,

we split the binary tree into the individual search paths that lead to each node and randomly assign each path to an address i. If we give the adversary the search path, it cannot do better than randomly guessing which value i it corresponds to since the paths are selected with equal probability. This tells us that the binary search path does not give any more information that simply the node it ends at, so we can eliminate the binary search from the access pattern and simply replace it with the physical word where the search halts.

4.2.3 Constructing the Access Pattern

Now that we know we can skip the binary search, let us construct the access pattern for one pass through the simulation before getting into the whole proof. First, each non-shelter location needs to be tagged. To do this the RAM will first need to read, then write back the value stored in each word. The access pattern for this part will be:

$$\left(1, 1, 2, 2, 3, 3, \ldots, m + \sqrt{m}, m + \sqrt{m}\right)$$

Next, the Batcher sort is performed. It will take $(m + \sqrt{m}) \cdot \lceil \log(m + \sqrt{m}) \rceil^2$ comparisons, and will need to perform two reads and two writes per comparison. This means there will be $4(m + \sqrt{m}) \cdot \lceil \log(m + \sqrt{m}) \rceil^2$ accesses for the sort, which we will denote by:

$$\left(b_1, b_2, \ldots, b_{4(m+\sqrt{m}) \cdot \lceil \log(m+\sqrt{m}) \rceil^2}\right)$$

Then we will need to simulate the $\sqrt{m}$ accesses from the original RAM. For one original access, we will read the entire shelter ($\sqrt{m}$ accesses), then read and write one non-shelter location since we can skip the binary search (2 accesses), and finishing by reading and writing the entire shelter ($2\sqrt{m}$ accesses). This will be done a total of $\sqrt{m}$ times, leading to $(2 + 3\sqrt{m})\sqrt{m} = 2\sqrt{m} + 3m$ accesses, which

we will denote by:

$$\left(a_1, a_2, \ldots, a_{2\sqrt{m}+3m}\right)$$

Finally, we will need to return everything to its original location using another Batcher sort. This one is over $m + 2\sqrt{m}$ locations, giving us an access pattern of:

$$\left(b'_1, b'_2, \ldots, b'_{4(m+2\sqrt{m})\cdot\lceil\log(m+2\sqrt{m})\rceil^2}\right)$$

For one pass through the simulation, the complete access pattern will be:

$$\begin{aligned}\big(&1, 1, 2, 2, \ldots, m+\sqrt{m}, m+\sqrt{m}, b_1, b_2, \ldots, b_{4(m+\sqrt{m})\cdot\lceil\log(m+\sqrt{m})\rceil^2},\\ &a_1, a_2, \ldots, a_{2\sqrt{m}+3m}, b'_1, b'_2, \ldots, b'_{4(m+2\sqrt{m})\cdot\lceil\log(m+2\sqrt{m})\rceil^2}\big)\end{aligned}$$

4.2.4 One Pass Through the Simulation Does Not Leak Information

We need to show that one pass through the simulation does not leak any information about the memory accesses being simulated besides the number of accesses. We can start by eliminating the accesses that are the same every pass from our access pattern. To begin with, our access pattern for one pass looks like:

$$\begin{aligned}\big(&1, 1, 2, 2, \ldots, m+\sqrt{m}, m+\sqrt{m}, b_1, b_2, \ldots, b_{4(m+\sqrt{m})\cdot\lceil\log(m+\sqrt{m})\rceil^2},\\ &a_1, a_2, \ldots, a_{2\sqrt{m}+3m}, b'_1, b'_2, \ldots, b'_{4(m+2\sqrt{m})\cdot\lceil\log(m+2\sqrt{m})\rceil^2}\big)\end{aligned}$$

The accesses to update the tags are the same every pass, so we can remove them:

$$(1, 1, 2, 2, \ldots, m+\sqrt{m}, m+\sqrt{m})$$

leaving:

$$\begin{aligned}\big(&b_1, b_2, \ldots, b_{4(m+\sqrt{m})\cdot\lceil\log(m+\sqrt{m})\rceil^2}, a_1, a_2, \ldots, a_{2\sqrt{m}+3m},\\ &b'_1, b'_2, \ldots, b'_{4(m+2\sqrt{m})\cdot\lceil\log(m+2\sqrt{m})\rceil^2}\big)\end{aligned}$$

as the portion that can leak information. We can also remove both Batcher sorts:

$$\left(b_1, b_2, \ldots, b_{4(m+\sqrt{m})\cdot\lceil\log(m+\sqrt{m})\rceil^2}\right)$$

and:

$$\left(b'_1, b'_2, \ldots, b'_{4(m+2\sqrt{m})\cdot\lceil\log(m+2\sqrt{m})\rceil^2}\right)$$

since the steps performed by the sort only depend on the number of items being sorted and not the values being sorted, leaving us:

$$\left(a_1, a_2, \ldots, a_{2\sqrt{m}+3m}\right)$$

When we simulate one access, we scan the shelter twice. The shelter scans are the same for each simulated access so those can be removed as well. For one simulated access, $3\sqrt{m}+2$ accesses are made; for the i^{th} simulated access where $k = (i-1)(3\sqrt{m}+2)$ is the number of accesses already made in this pass, the accesses would be:

$$\left(a_{k+1}, a_{k+2}, \ldots, a_{k+\sqrt{m}}, a_{k+\sqrt{m}+1}, a_{k+\sqrt{m}+2}, a_{k+\sqrt{m}+3}, a_{k+\sqrt{m}+4}, \ldots, a_{k+3\sqrt{m}+2}\right)$$

The first $\sqrt{m}$ accesses:

$$\left(a_{k+1}, a_{k+2}, \ldots, a_{k+\sqrt{m}}\right)$$

are the first scan of the shelter and can be removed. The next two accesses:

$$\left(a_{k+\sqrt{m}+1}, a_{k+\sqrt{m}+2}\right)$$

are the read and the write to the non-shelter location and need to remain in the access pattern for now. The final $2\sqrt{m}$ accesses:

$$\left(a_{k+\sqrt{m}+3}, a_{k+\sqrt{m}+4}, \ldots, a_{k+3\sqrt{m}+2}\right)$$

are the second scan of the shelter, where each shelter word is read and then rewritten, which we can remove. This reduces the portion of our access pattern that can leak information to:

$$\big(a_{\sqrt{m}+1}, a_{\sqrt{m}+2}, a_{4\sqrt{m}+3}, a_{4\sqrt{m}+4}, \ldots, a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+1},$$
$$a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+2}, \ldots, a_{3m-1}, a_{3m}\big)$$

which are the accesses to the non-shelter locations. Each non-shelter location is accessed twice, once to read and once to write, one immediately after another. This tells us accesses:

$$a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+1}$$

and

$$a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+2}$$

are the same for $i = 1, \ldots, \sqrt{m}$. The double access to the same location does not add any additional information, so we can remove the second access to the same location, $a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+2}$, leaving us with:

$$\big(a_{\sqrt{m}+1}, a_{4\sqrt{m}+3}, a_{7\sqrt{m}+5}, \ldots, a_{(i-1)(3\sqrt{m}+2)+\sqrt{m}+1}, \ldots, a_{3m-1}\big)$$

as the portion of the access pattern that can leak information. We know by definition of the square root solution that none of these accesses are to the same word. We are either accessing program word a_i for the first time (physical address is $\pi(a_i)$) or if this is not the first time we are accessing a_i we found it in the shelter and we are accessing a dummy location. If we are accessing a dummy location, we know we never access the same dummy location twice because the counter *count* is incremented after each simulated memory access. Because we never access the same memory location twice, the physical accesses themselves are the only thing

that could leak information. However, since $\pi(\cdot)$ is a uniformly random permutation, the adversary cannot do better than guessing which word is currently stored in that location. Therefore, the only thing that is leaked by these accesses is the number of accesses being simulated which does not allow the adversary to learn any additional information about the underlying access pattern of the program being simulated.

4.2.5 Multiple Passes Do Not Add Information

Now that we know one pass does not leak information besides the number of accesses, we need to show that multiple passes through the simulation do not add information. For each pass a new permutation is chosen uniformly at random. Because the new permutation is independent of the previous permutation, the locations accessed during one pass are independent of the locations accessed in the second pass even if the same memory locations are accessed. This means that one pass through the simulation does not give the adversary any information about the next pass. Therefore, the square root solution does not leak any information besides the output, running time and amount of memory used and is thus oblivious.

4.3 Why the Shelter and Dummy Elements are Needed

The shelter and dummy elements are a key pieces of the square root solution for providing oblivious simulation. Before getting into the details of why, we will quickly review the security game in question (see Section 3.2.1 for the details). In the oblivious RAM security game, the adversary is allowed to query the access pattern oracle with two RAM inputs. The oracle runs one of the programs and returns the access pattern. The adversary, after sending some number of queries, then guesses which inputs were being run on the RAM. If the adversary can guess with high probability which inputs are being used, then the adversary must be

learning some information from the access patterns returned by the oracle (in this case we say that the RAM is leaking some sort of information via the access pattern). Both the shelter and the dummy elements are required pieces of the square root solution to prevent the access pattern from leaking any information.

The shelter is used to hide multiple accesses to the same word during one pass. If there was not a space to put the words already accessed, the RAM would need to re-access those words in their original location. If we accessed the same word in the same location twice between reshuffles, it would leak information and potentially allow the adversary to distinguish between the two RAM inputs sent to the access pattern oracle in the ORAM game (one may access a unique location every time, while another accesses the same location twice, giving the adversary enough information to correctly guess which input goes with the access pattern). The other alternative would be to start a new pass if we try to access the same location twice during a pass. However, this would leak information too. The adversary would see that a reshuffle of the memory was done too soon and could distinguish between this and another RAM input that did not access the same location twice during a pass. For these two reasons the shelter must be available for the RAM to use.

Now, suppose we do not have the dummy elements and simply have the shelter. The first time we access a program address during a pass, we would scan the shelter, read/write one non-shelter address and then scan and update the shelter. If we did not have the dummy elements, the second time we accessed a program address, we would scan the shelter, find the location, then scan and update the shelter again without accessing a non-shelter location. Again, this could potentially allow the adversary to distinguish between access patterns in the ORAM security game. A potential alternative to using dummy elements would be to read a real element that has not been read already but this has an unacceptable overhead. This is

because the CPU portion of the RAM, which has a limited work tape,[3] would need to keep track of the program addresses that have been accessed so far, up to $\sqrt{m}$ addresses at a time. Because of the limited internal state, the CPU cannot afford to keep track of this information. The final alternative would have the RAM randomly selected a program word, check if that word has already been accessed, and if it has not, go ahead and access it. However, the only way for the RAM to know if it had been accessed would be to scan the shelter. These additional scans of the shelter would leak information to the adversary. Accesses patterns could be distinguished based on whether or not there are extra scans of the shelter. This leaves the $\sqrt{m}$ dummy elements as the only acceptable option to hide multiple accesses to the same program word in one pass.

From here, the next obvious question is why $\sqrt{m}$ shelter and dummy locations are used. First, it should be explained that we need to have at least as many dummy locations as shelter locations or else the square root solution breaks down (if we have $\sqrt{m}$ shelter locations, less than $\sqrt{m}$ dummy locations and we read the same program address $\sqrt{m}$ times in one pass we would run out of dummy elements to access). We also do not need more dummy locations than shelter locations because we would never access the extra locations in a single pass. The choice of $\sqrt{m}$ locations is a cost trade off on space. If we choose a smaller value, fewer original accesses could be simulated in one pass and the memory would need to be reshuffled more frequently (if m is the number of original memory locations and n is the number of dummy locations as well as the number of shelter locations, the total cost of both Batcher sorts would be $(m+n)\cdot\lceil\log_2(m+n)\rceil^2+(m+2n)\cdot\lceil\log_2(m+2n)\rceil^2$). However, if we choose a larger value, the cost of simulating a single access increases significantly. Again, if n is the number of shelter locations it would take $3n + 2$ accesses to simulate a single access and $(3n + 2)n$ accesses to simulate an

[3]Recall from Section 3.1 that the CPU has a small amount of memory on its work tape while the MEM has a much larger work tape.

entire pass. The choice of $\sqrt{m}$ balances both costs to a reasonable amount.

4.4 Improvements

A simple improvement can be made that removes roughly two thirds of the accesses per pass through the simulation. First, recall how a single access is simulated now, with a cost of $3\sqrt{m}+2$, ignoring the accesses for the binary searches:

- Read the shelter, looking for the program address a_i ($\sqrt{m}$ accesses)
- If i is found, read one dummy location at $\pi(m + count)$, else read $\pi(a_i)$. Write an updated value to that location where the program address is ∞. (2 accesses)
- Scan the entire shelter, reading than writing back each word. If a_i is already in the shelter, update the existing value if needed when read. Otherwise write to first empty location. ($2\sqrt{m}$ accesses)

This is done $\sqrt{m}$ times for a total of $3m+2\sqrt{m}$ accesses per pass. Suppose instead we take advantage of the internal counter *count* (that is used to access dummy locations). *count* is incremented each pass, whether or not a dummy location was accessed. We can use *count* to track how far into the shelter we need to scan and which shelter location we should write to next. We will also move the rewrite of shelter values to the first scan of the shelter. A single access will now be simulated as follows:

- Scan shelter locations 1 to $count - 1$, reading then rewriting every value, looking for program address a_i. If a_i is found in the shelter, update the program address to ∞ when rewriting the word. ($2(count - 1)$ accesses)
- If a_i was found in the shelter, read one dummy location at $\pi(m + count)$ and rewrite the same value back to that location. Otherwise, read $\pi(a_i)$ and

write an updated value to that location with a program address of ∞. (2 accesses)

- Write a_i into the shelter at location $m + \sqrt{m} + count$ (this will be the next empty shelter location). (1 access).

With this method, one simulated access will take $2(count - 1) + 3$ accesses for a total of $m+2\sqrt{m}$ accesses per pass through the simulation which is roughly a third of the original cost. We simply need to show that this method does not leak any additional information over the original square root solution.

The construction of $\pi(\cdot)$, both Batcher sorts and the binary searches are the same in both versions, so we can eliminate those from being potential leaks in our new version. The non-shelter accesses will also be the same in both versions so that cannot leak any additional information as well. This leaves the scan of the shelter and the write to the shelter as the only source of additional information left. If we construct the access pattern for this portion, we will see a write to the first shelter location, then a read/write to the first location and a write to the second, and so on until we read/write the entire shelter except the last position then write to the last position. This gives us an access pattern of:

$$\begin{aligned}\big(&m+\sqrt{m}+1, m+\sqrt{m}+1, m+\sqrt{m}+1, m+\sqrt{m}+2, m+\sqrt{m}+1,\\ &m+\sqrt{m}+1, m+\sqrt{m}+2, m+\sqrt{m}+2, m+\sqrt{m}+3, m+\sqrt{m}+1,\\ &m+\sqrt{m}+1, m+\sqrt{m}+2, m+\sqrt{m}+2, m+\sqrt{m}+3, m+\sqrt{m}+3,\\ &m+\sqrt{m}+4, \ldots, m+\sqrt{m}+\sqrt{m}-1, m+\sqrt{m}+\sqrt{m}-1, m+\sqrt{m}+\sqrt{m}\big)\end{aligned}$$

This portion of the access pattern will be the same each pass through the simulation so it cannot leak any information about the program words that are being accessed. The only information that can be learned is how many accesses have been done so far by looking at how far into the shelter we read and write. However, in the original square root solution, this information can also be learned by counting the number

of accesses to the non-shelter locations. Therefore, no additional information is leaked in this version of the square root solution.

Chapter 5

THE HIERARCHICAL SOLUTION

While the square root solution provides oblivious simulation, more efficient solutions are possible. The first of these is the basic hierarchical solution [8]. In this solution a series of increasingly larger hash tables are used. Each hash table operates much like the single buffer of the square root solution. Instead of permuting the program memory according to a random permutation, as is done in the square root solution, the values are stored according to a randomly selected hash function associated with each level. When a sufficient number of accesses have been made, a new hash function is chosen for that level and the entries stored in that level are rehashed. For the smaller hash tables, this will occur more frequently, while for the larger hash tables it will occur infrequently.

Accessing values in this solution is similar in some respects to the square root solution as well. The highest level hash table (and thus the smallest), is scanned completely each time like the shelter is scanned in the square root solution. For each subsequent hash table, one bucket is accessed. If the program address in question has not been found yet, the RAM will use the hash function for the level to determine which bucket to scan, otherwise the RAM will choose a random bucket. Once a bucket has been scanned in each hash table, the entry for the program address is written into the highest level hash table.

5.1 Overview

The hierarchical solution is divided into three pieces: how the RAM's memory is structured, how program memory is accessed and rehashing of individual levels. We will cover each of these in turn, building up the solution as we go.

5.1.1 Memory Layout

In the square root solution (Chapter 4), a simple table structure is used for the memory. The table is large enough to contain the original program data plus dummy and shelter words used to hide accesses to the same location between shuffles of the memory. The hierarchical model generalizes this structure (see Figure 5.1). Instead of a single table, a series of hash tables are used. Say a program has been running on the RAM for t steps (initially t is the length of the input to the RAM) so far, then each hash table consists of buckets of depth in the order of $O(\log t)$. The hash tables increase rapidly in size as you move from higher to lower levels; specifically, the hash table at level i contains 4^i buckets. As we will see, the choice of 4^i for the number of buckets, combined with how often each level is rehashed, minimizes the chance of bucket overflows during the simulation.

Each hash table is associated with a specific hash function chosen from the set:

$$h_{s_i}(a) = RO(s_i \cdot a) \pmod{4^i} \tag{5.1}$$

where $RO(\cdot)$ is the random oracle given to the RAM for this simulation and $s_i \in \mathbb{N}$. s_i is referred to as the *hash function index* and specifies which function should be used. Each hash table is associated with a hash function index which is chosen uniformly at random for each table.

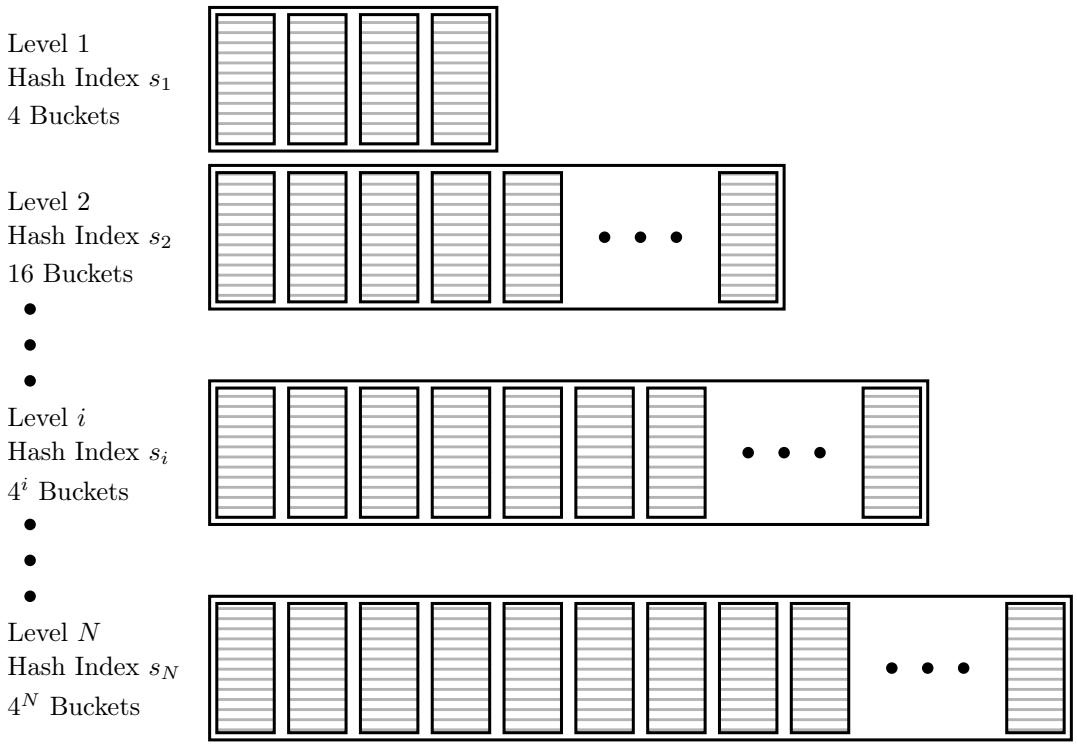

Figure 5.1: Memory Layout

5.1.2 Accessing Program Memory

The access algorithm for the hierarchical model is a generalization of the access algorithm of the square root solution (Section 4.1). When the RAM needs to access program memory location a, it starts by looking at the highest hash table (level one). At level one, the entire hash table is scanned entry by entry. If a exists at this level, the RAM remembers the value and continues scanning the rest of the memory. For the remaining hash tables, the RAM scans the hash table at level i as follows. First, the RAM computes the hash value of a at that level, $h_{s_i}(a)$. Then it chooses a random bucket in the hash table, which we will denote as j ($j \xleftarrow{\$} [4^i]$). If a has already been found at a higher level, the RAM scans bucket j of the hash table. Otherwise, if a has not been found, the RAM scans bucket $h_{s_i}(a)$ and remembers the value if it is found. Once all of the levels have been

scanned, the (potentially updated) value of (a, v), where v is value associated with program address a, is written to the first empty bucket in level one.

Most of these steps are directly comparable to the square root solution. Suppose that during a simulated access in the hierarchical solution, we found the program address we were looking for in level n. Let us compare the accesses made in this solution to the square root solution. We begin by scanning the entire hash table in level one much like we scan the entire shelter. Then, for $2 \leq i \leq n$, we scan bucket $h_{s_i}(a)$ for level i. These accesses are equivalent to the single access to $\pi(a)$ in the square root solution when we did not find the program address in the shelter. In the hierarchical model, for the accesses to the remaining levels below n, we choose a bucket at random to scan since we have already found the address in question, which is a similar idea to reading one of the dummy elements when we find the address in the shelter in the square root solution.

5.1.3 Rehashing a Level

Rehashing each level at regular intervals is key to providing oblivious simulation. If we find program memory location a in level i twice, we want the buckets we scan to be independent of each other, otherwise accessing the same bucket could leak information about the access pattern. The need for rehashing in this solution is the same reason we choose a new permutation in each pass of the square root solution. One could think of the time between the rehashing of a level as a single pass through the square root solution and choosing a new hash function for the level would be equivalent to choosing a new permutation for the next pass. Each time a specific program word is in a level, we want the location to be independent or else the adversary in the oblivious RAM game may be able to detect a difference between different inputs to the RAM.

Recall that when accessing program word a, when we find location a in level i, we move the value to the hash table at the top of the structure. When it comes

time to move it and the other contents of the highest level hash table into the second level, we begin the rehash by choosing a new hash function for the second level. For each rehashing of the contents of level j into level $j+1$, we continue to choose a new hash function uniformly at random for level $j+1$. Before we can move program word a back into level i, we choose a new hash function for level i like we did all the other levels. Since the hash functions are chosen uniformly at random, the new bucket location for a is independent of the previous bucket we found it in (see Section 5.2.1 for further details).

As originally proposed by Ostrovsky, for some level i, every 4^{i-1} accesses we move all of the contents of level i into level $i+1$ and choose a new hash function for level $i+1$ [8]. Because the smaller levels have a lower cost associated with rehashing all of the values with the new hash function, these tables are rehashed more frequently. The larger levels are rehashed infrequently because the cost is much higher. By rehashing every 4^{i-1} accesses, we reduce the possibility of a bucket overflow in level i. However, this hashing needs to be done in an oblivious manner that prevents the RAM from leaking information. We will describe the oblivious hash algorithm in Section 5.2.2.

5.2 Proof

Proving that the hierarchical model provides oblivious simulation is best done in pieces. First we will show that the access pattern for one of the hash table levels does not leak any information between rehashings. We can consider the accesses to a single level between rehashings to be equivalent to the accesses to the non-shelter locations in a single pass of the square root solution. If the accesses to the level appear to be accessing a randomly selected bucket to an outside observer, then no information is leaked by these accesses by themselves.

Once we know that the accesses to a single level are oblivious between rehashings, we will show that the accesses made by the RAM during rehashing are

oblivious. If the rehashing steps are not oblivious, they could leak information about the contents of the two levels in question, which does not meet the standard of oblivious simulation. The rehashing steps need to be independent of the contents currently stored in the two levels to prevent the adversary in the oblivious RAM game (see Section 3.2.1) from learning any information. This step is somewhat similar to the steps for choosing a new permutation in the square root solution.

Now that we know the accesses made to a level between rehashings and the rehashing steps, we need to show that looking at a level across multiple rehashings does not leak any information. Even though the accesses between rehashings do not leak information, it is possible that looking at the accesses across multiple rehashings could shed some additional information. In order to provide oblivious simulation, the hierarchical solution has to prevent this from happening. In the square root solution, to show that multiple passes do no leak information, we showed that the permutation selected for each pass was chosen uniformly at random. For this solution, we will do something very similar; we will show that the hash function chosen for a level is independent of all the other hash functions selected.

Put together, these three pieces provide oblivious simulation. We know that the accesses made between rehashings do not leak information, nor do the steps during rehashing. Additionally, the accesses made to a level before a rehashing do not leak any information about the accesses made afterward. Given these facts, the adversary will not be able to gain enough information to distinguish which world it is in with high probability.

5.2.1 A Level's Access Pattern Does Not Leak Information Between Rehashings

When we look at the hash table at level i in the hierarchical model, we will see 4^{i-2} accesses to that level between rehashings (recall that level $i-1$ is rehashed into level

i every 4^{i-2} accesses in the original program and we choose a new hash function for level i at this time). Recall from Section 5.1.2 that the locations accessed are determined either by the hashed value of the program memory location, or a bucket chosen at random if the word has already been found. We need to show that all of the accesses appear to be uniformly random.

If we have already found the program word, we choose a bucket to scan uniformly at random. Since these accesses are chosen at random, they will look like random accesses to the outside observer, so we can eliminate these accesses as a potential source for leaking information. This leaves the accesses that are determined by the hash function for the level and we will want them to look uniformly random too. We know that the RAM will look for program memory word a at most once at level i between rehashings (Section 5.1.3) so $h_{s_i}(a)$ is used to determine which bucket to look at once at the most. This means we do not have to worry about querying the hash function on the same input twice. What remains to be shown, is that the outputs look uniformly random.

Recall from Section 5.1.1 that the hash function is constructed as follows:

$$h_{s_i}(a) = RO(s_i \cdot a) \pmod{4^i}$$

where $RO(\cdot)$ is the random oracle given to the RAM for this simulation and $s_i \in \mathbb{N}$. The random oracle $RO(\cdot)$ is equally likely to map to any value in its range on some input. This means that for some input x, $RO(x) \pmod{4^i}$ is equally likely to map to any value in $\mathbb{Z}_{4^i}$. Since each output is equally likely, the output we actually see looks like it was chosen uniformly at random. Therefore, the output of $h_{s_i}(a)$ looks uniformly random as well.

Between rehashings, each access looks like a bucket was chosen at random, regardless of whether we chose a bucket at random or used the hash function to select on. Since the accesses look random, they cannot leak any information in the

oblivious RAM security game (see Section 3.2.1).

5.2.2 Rehashing is Oblivious

When one level is rehashed into the next larger level, the hashing needs to be done in a way that does not leak any information about the words being hashed, including how many real entries there are at this level. This is referred to as an *oblivious hash* and is a key piece of the hierarchical model.

Oblivious Hash Algorithm

When we rehash in the hierarchical model, we have the smaller hash table, which we will refer to as hash table A, which has n buckets of size m (if table A is at level i, $n = 4^i$ and m is of the order $O(\log t)$ where t is the running time so far), and the larger hash table, hash table B, which has $4n$ buckets of size m. When we rehash level A into level B, a new hash function is chosen and associated with hash table B. We will refer to this new hash function as $h'(\cdot)$. To begin the oblivious hash, we start by allocating a new hash table, which we will call C, with $5n$ buckets. The n buckets of hash table A are copied into the first n buckets of table C while the $4n$ buckets of hash table B are copied into the last $4n$ buckets (see Figure 5.2).

Once the buckets are copied, we scan every word in C. The non-empty words, (a, v), are tagged with the new hash value determined by $h'(\cdot)$, $(h'(a), a, v)$. In order to make sure that every bucket of B contains at least one value when we rehash (we will see why later), we will use the first $4n$ empty entries we find and tag an empty entry with one of the bucket numbers (1 to $4n$). We know that there will be at least $4n$ empty entries since we rehash every 4^{i-2} accesses and A has $4^i m$ slots available (4^i buckets, each of size m). Hash table A will be at most a quarter full, ensuring that we have at least $4n$ empty entries. The remaining empty words found in the hash table are tagged with a value of zero (see Figure 5.3).

Once all of the hash table entries have been tagged, we sort the hash table on

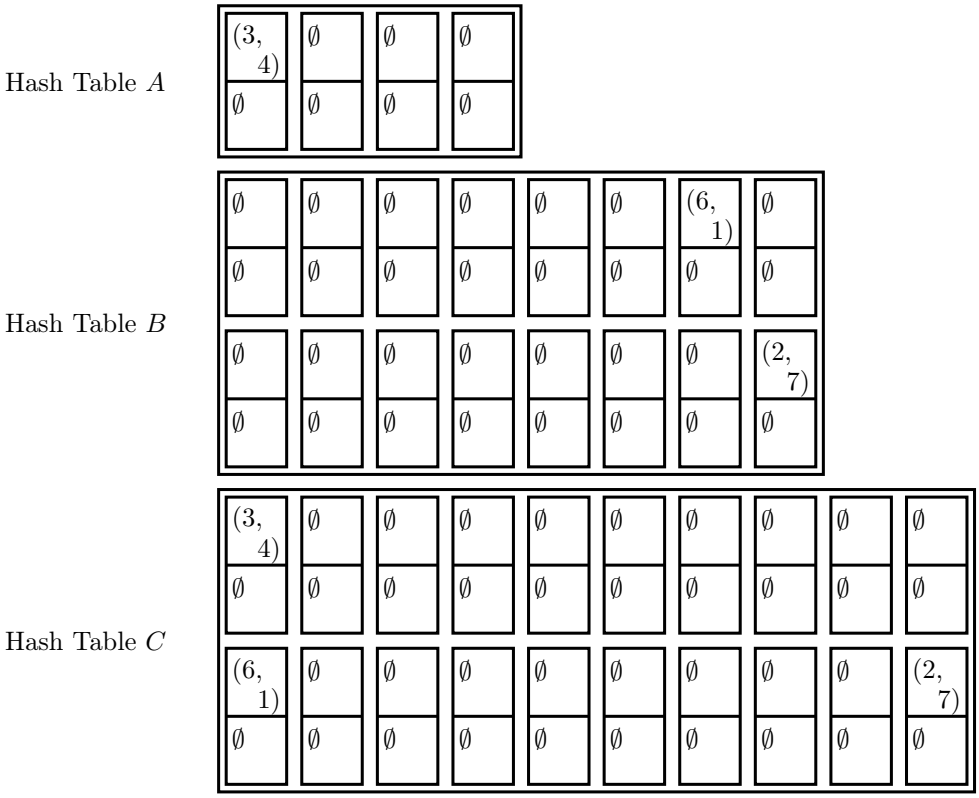

Figure 5.2: Example: Initial State of Hash Tables A, B and C

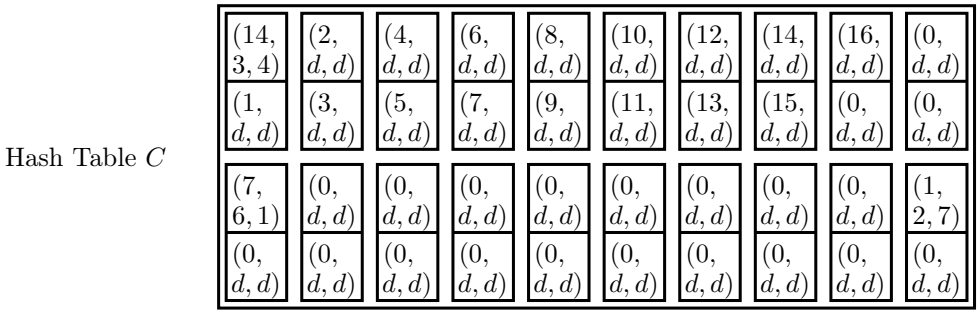

Figure 5.3: Example: Tagging the Entries in C with $h'(\cdot)$

a entry-by-entry basis (see Figure 5.4), treating the hash table as one large array instead of a hash table (instead of putting each entry into a bucket as determined by the hash function, we view the hash table as one large traditional array and apply the sorting algorithm to each individual entry in that array). The empty

words, tagged with a value of zero, will be in the front of the "array" (the entries in the buckets at the start of the hash table), while the empty elements tagged with bucket numbers and the non-empty words will be in the end of the "array" (the last buckets of the hash table). Any sorting algorithm will do, as long as it makes deterministic steps that are chosen by the number of items being sorted, such as the Batcher sort used in the square root solution (Section 2.5).

Hash Table C

$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$
$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$
$(0, d, d)$	$(1, d, d)$	$(3, d, d)$	$(5, d, d)$	$(7, d, d)$	$(8, d, d)$	$(10, d, d)$	$(12, d, d)$	$(14, d, d)$	$(15, d, d)$
$(1, 2, 7)$	$(2, d, d)$	$(4, d, d)$	$(6, d, d)$	$(7, 6, 1)$	$(9, d, d)$	$(11, d, d)$	$(13, d, d)$	$(14, 3, 4)$	$(16, d, d)$

Figure 5.4: Example: Sorting the Entries in C by Tag

Next, a new hash table, which we will refer to as C', is created and contains $5mn$ buckets. Each word in C is copied in order to the top-most word in the first empty bucket of C' (see Figure 5.5). The words that are copied from C will be tagged with the values we specified before, while the rest of the entries in C' will be empty entries with no tags.

C' is scanned, left to right, as we obliviously sort the words of every two adjacent buckets. To obliviously sort the buckets, first we scan the non-empty elements of each bucket. If the tags of the elements are different, we perform the steps of the oblivious sort without moving any elements. We will not need to actually move the entries because each bucket consists of entries with a unique tag and the entries are already in sorted order because of how we copied them from C to C'.

If the buckets have the same tag, the oblivious sort is performed as follows to accumulate the empty entries in the first bucket and the non-empty entries (which all have the same tag) in the second bucket. If a word is empty (i.e. it was not one

Hash Table C'

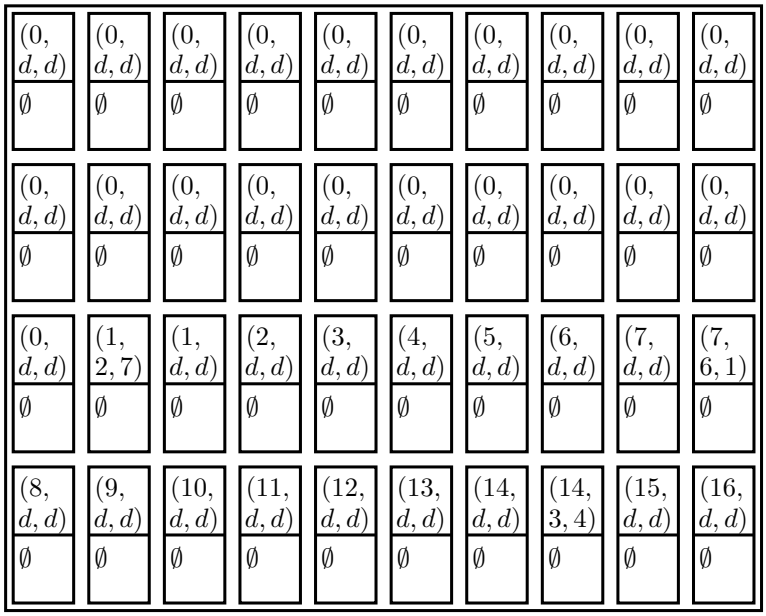

Figure 5.5: Example: The Entries Moved From C to C'

Hash Table C'

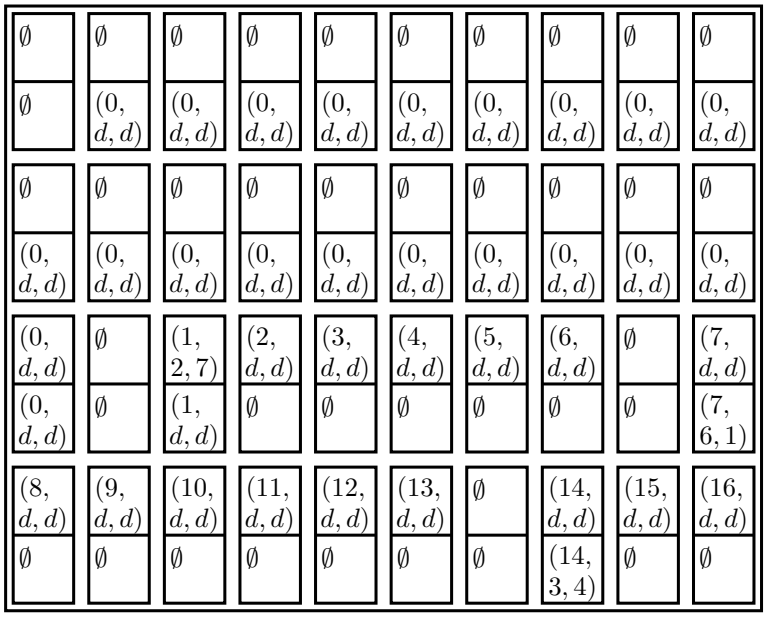

Figure 5.6: Example: Sorting the Entries in Adjacent Buckets in C'

of the words we copied in from C), we treat it as if it was tagged with a value of zero. If the word is non-empty (i.e. we copied it from C), we treat it as if it was

tagged with a value of one. We are using these fake tags for this step since the empty entries do not have a tag. The oblivious sort using these fake tags is done using a deterministic sorting algorithm such as the Batcher sort (Section 2.5).

Once we have sorted all of the adjacent buckets, all of the entries with the same tag will be in one bucket together (see Figure 5.6). All of the buckets with non-empty entries will be in sorted order by tag, potentially with empty buckets between them.

Next, we obliviously sort C' by buckets (see Figure 5.7). Empty buckets are treated as being tagged with -1, moving them to the front of the hash table. The non-empty buckets will be in the last $4n$ positions (we know there will be $4n$ since we tagged empty elements with each of the bucket numbers).

Hash Table C'

$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$
$\emptyset$	$\emptyset$	$\emptyset$	$(0, d, d)$	$(1, 2, 7)$	$(2, d, d)$	$(3, d, d)$	$(4, d, d)$	$(5, d, d)$	$(6, d, d)$
$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(0, d, d)$	$(1, d, d)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$(7, d, d)$	$(8, d, d)$	$(9, d, d)$	$(10, d, d)$	$(11, d, d)$	$(12, d, d)$	$(13, d, d)$	$(14, d, d)$	$(15, d, d)$	$(16, d, d)$
$(7, 6, 1)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(14, 3, 4)$	$\emptyset$	$\emptyset$

Figure 5.7: Example: Sorting the Buckets in C'

The last $4n$ buckets of C' are copied back into B (see Figure 5.8). Finally, we scan B and eliminate the dummy entries (see Figure 5.9).

Hash Table B

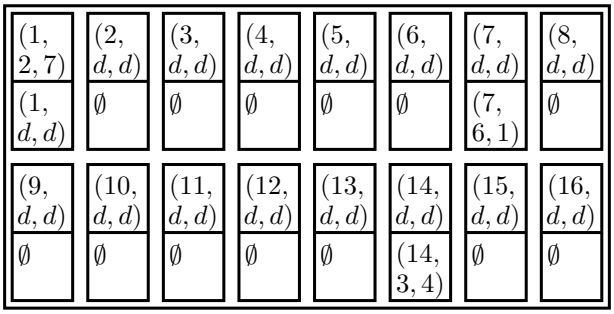

Figure 5.8: Example: Moving the Last $4n$ Buckets of C' into B

Hash Table B

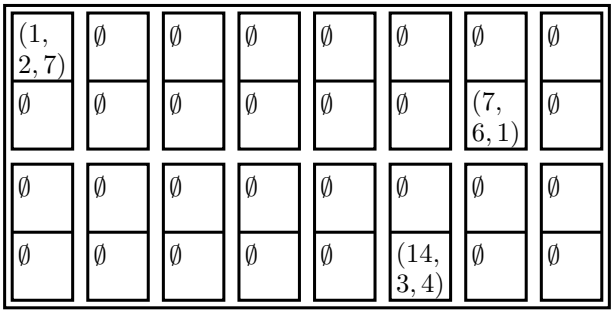

Figure 5.9: Example: Eliminate the Dummy Entries in B

Constructing the Access Pattern for Oblivious Hashing

Given the more complex memory structure used in the hierarchical solution, we will need a new message form for passing instructions from the CPU to the MEM in the RAM. In the basic RAM model, our messages are of the form (a, i, v) where a is the physical address the CPU wishes to access in the MEM, i is the instruction and v is the value (see Section 3.1 for further details). Now we will need to specify the hash table and bucket that we are interested in accessing (and in some cases the specific physical word in the bucket): $((t, b, w), i, v)$ where t is the hash table, b is the bucket, w is the word, i is the instruction and v is the value. It is important to remember that these messages are specifying physical locations to access in the MEM and not the program addresses from the original RAM during oblivious simulation. For example, if the CPU wanted to read the first value in the fifth bucket of the hash table for the second level, the message would be $((2, 5, 1), read, \perp)$. When we construct the access pattern, the only portion of the message that we care about is the physical address being accessed, in this case the tuple (t, b, w) specified above.

Now that we have a message format that works for the hierarchical solution, we can begin to construct the access pattern created by the oblivious hash. Recall from Section 5.2.2 that we have the smaller hash table at level i, which we will call table A, which has $n = 4^i$ buckets of size m, where m is of the order $O(\log t)$. Additionally, we have the larger hash table B at level $i + 1$ which has $4n$ buckets of size m. Two additional hash tables are created just for the rehash, table C which has $5n$ buckets of size m and table C' which has $5mn$ buckets of size m. The construction of the access pattern will follow the steps described in Section 5.2.2. To begin with, we need to move everything from hash tables A and B into

C, which generates accesses to the following physical memory locations:

$$((A,1,1),(C,1,1),(A,1,2),(C,1,2,),\ldots,(A,n,m),(C,n,m),(B,1,1),$$
$$(C,n+1,1),(B,1,2),(C,n+1,2),\ldots,(B,4n,m),(C,5n,m))$$

Next, we scan C and update each value with its new tag, causing these locations to be accessed:

$$((C,1,1),(C,1,1),(C,1,2),(C,1,2),\ldots,(C,4n,m),(C,4n,m))$$

After updating the tags, C is sorted at the word level using a Batcher sort. The steps taken by the Batcher sort are predictable and determined solely by the number of words being sorted. Recall from Section 2.5 that when sorting x items, the Batcher sort will make $x\lceil\log x\rceil^2$ comparisons which will mean four times that many accesses (reading the two words being compared and then writing them back). For the i^{th} comparison, we will denote the two addresses in question as $(C,b_{i,1},w_{i,1})$ and $(C,b_{i,2},w_{i,2})$ where $b_{i,1}$ is the bucket the first entry is in, $w_{i,1}$ is the specific word in the bucket that contains it and the same for $b_{i,2}$ and $w_{i,2}$ for the second entry. In this step we are sorting $5mn$ words, giving us an access pattern of:

$$((C,b_{1,1},w_{1,1}),(C,b_{1,2},w_{1,2}),(C,b_{1,1},w_{1,1}),(C,b_{1,2},w_{1,2}),$$
$$(C,b_{2,1},w_{2,1}),(C,b_{2,2},w_{2,2}),(C,b_{2,1},w_{2,1}),(C,b_{2,2},w_{2,2}),\ldots,$$
$$\left(C,b_{5mn\lceil\log 5mn\rceil^2,1},w_{5mn\lceil\log 5mn\rceil^2,1}\right),\left(C,b_{5mn\lceil\log 5mn\rceil^2,2},w_{5mn\lceil\log 5mn\rceil^2,2}\right),$$
$$\left(C,b_{5mn\lceil\log 5mn\rceil^2,1},w_{5mn\lceil\log 5mn\rceil^2,1}\right),\left(C,b_{5mn\lceil\log 5mn\rceil^2,2},w_{5mn\lceil\log 5mn\rceil^2,2}\right))$$

Next we create the new hash table C' and move each word in C into the top-most word in the first empty bucket of C':

$$((C,1,1),(C',1,1),(C,1,2),(C',2,1),(C,1,3),(C',3,1),\ldots,$$
$$(C,5n,m),(C',5mn,1))$$

The next step is more complex when it comes to constructing the access pattern. We iterate across the buckets of C', obliviously sorting every two adjacent buckets. First we will compare buckets one and two, then two and three, and so on until we compare buckets $5mn-1$ and $5mn$. Let us begin by constructing the access pattern when we compare buckets i and $i+1$. First, we will need to determine if the entries in the buckets have different tags or the same tag (recall from Section 5.2.2 that at this point all of the entries in each bucket have the same tag or are empty). To find the tag for bucket i, we will need to read both the first and last word in the bucket. The non-empty tagged value will be in the first word if we did not sort bucket i in the previous step because the tags in buckets $i-1$ and i were different (or if i is the first bucket). If we did sort the words in the previous step because the tags were the same, the non-empty words, of which there will be at least two, will be accumulated in the end of bucket i, in which case we will want to read the last word in the bucket to get the tag. We will want to read both the first and last word in bucket i each time so we do not leak any information about what was done in the previous step. For the second bucket, we know the non-empty word is in the first word of the bucket so we only need to read one word. This gives us an access pattern of:

$$((C',i,1),(C',i,m),(C',i+1,1))$$

to find the tags contained in each bucket.

If the tags are the same in both buckets, we perform a Batcher sort across

both buckets treating the empty words as zero and the non-empty words as one to accumulate the non-empty words in the second bucket. If the tags are different, then we perform the same reads and writes as the Batcher sort without moving any of the words (since the Batcher sort steps are determined only by the number of items being sorted we know the steps in advance). We are sorting $2m$ words, which gives us an access pattern of:

$$\begin{gathered}\left(\left(C', b'_{1,1}, w'_{1,1}\right), \left(C', b'_{1,2}, w'_{1,2}\right), \left(C', b'_{1,1}, w'_{1,1}\right), \left(C', b'_{1,2}, w'_{1,2}\right),\right. \\ \left(C', b'_{2,1}, w'_{2,1}\right), \left(C', b'_{2,2}, w'_{2,2}\right), \left(C', b'_{2,1}, w'_{2,1}\right), \left(C', b'_{2,2}, w'_{2,2}\right), \ldots, \\ \left(C', b'_{2m\lceil \log 2m\rceil^2,1}, w'_{2m\lceil \log 2m\rceil^2,1}\right), \left(C', b'_{2m\lceil \log 2m\rceil^2,2}, w'_{2m\lceil \log 2m\rceil^2,2}\right), \\ \left.\left(C', b'_{2m\lceil \log 2m\rceil^2,1}, w'_{2m\lceil \log 2m\rceil^2,1}\right), \left(C', b'_{2m\lceil \log 2m\rceil^2,2}, w'_{2m\lceil \log 2m\rceil^2,2}\right)\right)\end{gathered}$$

Since we have $5mn$ buckets, and we compare every pair of adjacent buckets working left to right, we will have to do this $5mn - 1$ times. Since the Batcher sort takes predictable steps, we will be making the same relative accesses each time if you just consider the two buckets in question. If we look at the access pattern formed by this, we will simply be adding one to the bucket number on each pass.

Thus, the complete access pattern for this step will be:

$$\left((C',1,1),(C',1,m),(C',2,1),\left(C',b'_{1,1},w'_{1,1}\right),\left(C',b'_{1,2},w'_{1,2}\right),\left(C',b'_{1,1},w'_{1,1}\right),\right.$$
$$\left(C',b'_{1,2},w'_{1,2}\right),\left(C',b'_{2,1},w'_{2,1}\right),\left(C',b'_{2,2},w'_{2,2}\right),\left(C',b'_{2,1},w'_{2,1}\right),\left(C',b'_{2,2},w'_{2,2}\right),\ldots,$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1},w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2},w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1},w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2},w'_{2m\lceil\log 2m\rceil^2,2}\right),(C',2,1),$$
$$(C',2,m),(C',3,1),\left(C',b'_{1,1}+1,w'_{1,1}\right),\left(C',b'_{1,2}+1,w'_{1,2}\right),\left(C',b'_{1,1}+1,w'_{1,1}\right),$$
$$\left(C',b'_{1,2}+1,w'_{1,2}\right),\left(C',b'_{2,1}+1,w'_{2,1}\right),\left(C',b'_{2,2}+1,w'_{2,2}\right),\left(C',b'_{2,1}+1,w'_{2,1}\right),$$
$$\left(C',b'_{2,2}+1,w'_{2,2}\right),\ldots,\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+1,w'_{2m\lceil\log 2m\rceil^2,1}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+1,w'_{2m\lceil\log 2m\rceil^2,2}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+1,w'_{2m\lceil\log 2m\rceil^2,1}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+1,w'_{2m\lceil\log 2m\rceil^2,2}\right),\ldots,(C',5mn-1,1),(C',5mn-1,m),$$
$$(C',5mn,1),\left(C',b'_{1,1}+5mn-2,w'_{1,1}\right),\left(C',b'_{1,2}+5mn-2,w'_{1,2}\right),$$
$$\left(C',b'_{1,1}+5mn-2,w'_{1,1}\right),\left(C',b'_{1,2}+5mn-2,w'_{1,2}\right),$$
$$\left(C',b'_{2,1}+5mn-2,w'_{2,1}\right),\left(C',b'_{2,2}+5mn-2,w'_{2,2}\right),\left(C',b'_{2,1}+5mn-2,w'_{2,1}\right),$$
$$\left(C',b'_{2,2}+5mn-2,w'_{2,2}\right),\ldots,\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+5mn-2,w'_{2m\lceil\log 2m\rceil^2,1}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+5mn-2,w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+5mn-2,w'_{2m\lceil\log 2m\rceil^2,1}\right),$$
$$\left.\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+5mn-2,w'_{2m\lceil\log 2m\rceil^2,2}\right)\right)$$

Now C' is obliviously sorted by bucket, treating empty buckets as if they were tagged with -1 and non-empty buckets as the tag value that is on the words in the bucket. Once again, this will be done with a Batcher sort which in this case will take $5mn\lceil\log 5mn\rceil^2$ comparisons. For each comparison, we will need to read every word in both buckets and then write back all of the words in each bucket according to the Batcher sort. For the i^{th} comparison, this will give us an access

pattern of:

$$\left(\left(C', b''_{i,1}, 1\right), \left(C', b''_{i,1}, 2\right), \ldots, \left(C', b''_{i,1}, m\right), \left(C', b''_{i,2}, 1\right), \left(C', b''_{i,2}, 2\right), \ldots, \left(C', b''_{i,2}, m\right), \left(C', b''_{i,1}, 1\right), \left(C', b''_{i,1}, 2\right), \ldots, \left(C', b''_{i,1}, m\right), \left(C', b''_{i,2}, 1\right), \left(C', b''_{i,2}, 2\right), \ldots, \left(C', b''_{i,2}, m\right)\right)$$

For the entire bucket-wise Batcher sort we will have an access pattern of:

$$\left(\left(C', b''_{1,1}, 1\right), \left(C', b''_{1,1}, 2\right), \ldots, \left(C', b''_{1,1}, m\right), \left(C', b''_{1,2}, 1\right), \left(C', b''_{1,2}, 2\right), \ldots, \left(C', b''_{1,2}, m\right), \left(C', b''_{1,1}, 1\right), \left(C', b''_{1,1}, 2\right), \ldots, \left(C', b''_{1,1}, m\right), \left(C', b''_{1,2}, 1\right), \left(C', b''_{1,2}, 2\right), \ldots, \left(C', b''_{1,2}, m\right), \left(C', b''_{2,1}, 1\right), \left(C', b''_{2,1}, 2\right), \ldots, \left(C', b''_{2,1}, m\right), \left(C', b''_{2,2}, 1\right), \left(C', b''_{2,2}, 2\right), \ldots, \left(C', b''_{2,2}, m\right), \left(C', b''_{2,1}, 1\right), \left(C', b''_{2,1}, 2\right), \ldots, \left(C', b''_{2,1}, m\right), \left(C', b''_{2,2}, 1\right), \left(C', b''_{2,2}, 2\right), \ldots, \left(C', b''_{2,2}, m\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 1\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 1\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 1\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 1\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, m\right)\right)$$

Finally, we will need to copy the last $4n$ buckets of C' (where the non-empty buckets accumulated) into B:

$$\left((C', 5mn - 4n + 1, 1), (B, 1, 1), (C', 5mn - 4n + 1, 2), (B, 1, 2), \ldots, (C', 5mn - 4n + 1, m), (B, 1, m), (C', 5mn - 4n + 2, 1), (B, 2, 1), (C', 5mn - 4n + 2, 2), (B, 2, 2), \ldots, (C', 5mn - 4n + 2, m), (B, 2, m), \ldots, (C', 5mn, 1), (B, 4n, 1), (C', 5mn, 2), (B, 4n, 2), \ldots, (C', 5mn, m), (B, 4n, m)\right)$$

and eliminate any dummy entries:

$$((B,1,1),(B,1,1),(B,1,2),(B,1,2),\ldots,(B,1,m),(B,1,m),$$
$$(B,2,1),(B,2,1),(B,2,2),(B,2,2),\ldots,(B,2,m),(B,2,m),\ldots$$
$$(B,4n,1),(B,4n,1),(B,4n,2),(B,4n,2),\ldots,(B,4n,m),(B,4n,m))$$

This gives us a complete access pattern of:

$$((A,1,1),(C,1,1),(A,1,2),(C,1,2,),\ldots,(A,n,m),(C,n,m),(B,1,1),$$
$$(C,n+1,1),(B,1,2),(C,n+1,2),\ldots,(B,4n,m),(C,5n,m),(C,1,1),$$
$$(C,1,1),(C,1,2),(C,1,2),\ldots,(C,4n,m),(C,4n,m),(C,b_{1,1},w_{1,1}),$$
$$(C,b_{1,2},w_{1,2}),(C,b_{1,1},w_{1,1}),(C,b_{1,2},w_{1,2}),(C,b_{2,1},w_{2,1}),(C,b_{2,2},w_{2,2}),$$
$$(C,b_{2,1},w_{2,1}),(C,b_{2,2},w_{2,2}),\ldots,\left(C,b_{5mn\lceil\log 5mn\rceil^2,1},w_{5mn\lceil\log 5mn\rceil^2,1}\right),$$
$$\left(C,b_{5mn\lceil\log 5mn\rceil^2,2},w_{5mn\lceil\log 5mn\rceil^2,2}\right),\left(C,b_{5mn\lceil\log 5mn\rceil^2,1},w_{5mn\lceil\log 5mn\rceil^2,1}\right),$$
$$\left(C,b_{5mn\lceil\log 5mn\rceil^2,2},w_{5mn\lceil\log 5mn\rceil^2,2}\right),(C,1,1),(C',1,1),(C,1,2),(C',2,1),$$
$$(C,1,3),(C',3,1),\ldots,(C,5n,m),(C',5mn,1),(C',1,1),(C',1,m),$$
$$(C',2,1),\left(C',b'_{1,1},w'_{1,1}\right),\left(C',b'_{1,2},w'_{1,2}\right),\left(C',b'_{1,1},w'_{1,1}\right),\left(C',b'_{1,2},w'_{1,2}\right),$$
$$\left(C',b'_{2,1},w'_{2,1}\right),\left(C',b'_{2,2},w'_{2,2}\right),\left(C',b'_{2,1},w'_{2,1}\right),\left(C',b'_{2,2},w'_{2,2}\right),\ldots,$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1},w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2},w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1},w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2},w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$(C',2,1),(C',2,m),(C',3,1),\left(C',b'_{1,1}+1,w'_{1,1}\right),\left(C',b'_{1,2}+1,w'_{1,2}\right),$$
$$\left(C',b'_{1,1}+1,w'_{1,1}\right),\left(C',b'_{1,2}+1,w'_{1,2}\right),\left(C',b'_{2,1}+1,w'_{2,1}\right),$$
$$\left(C',b'_{2,2}+1,w'_{2,2}\right),\left(C',b'_{2,1}+1,w'_{2,1}\right),\left(C',b'_{2,2}+1,w'_{2,2}\right),\ldots,$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+1,w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+1,w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$\left(C',b'_{2m\lceil\log 2m\rceil^2,1}+1,w'_{2m\lceil\log 2m\rceil^2,1}\right),\left(C',b'_{2m\lceil\log 2m\rceil^2,2}+1,w'_{2m\lceil\log 2m\rceil^2,2}\right),$$
$$\ldots,(C',5mn-1,1),(C',5mn-1,m),(C',5mn,1),$$

$$\left(C', b'_{1,1} + 5mn - 2, w'_{1,1}\right), \left(C', b'_{1,2} + 5mn - 2, w'_{1,2}\right),$$
$$\left(C', b'_{1,1} + 5mn - 2, w'_{1,1}\right), \left(C', b'_{1,2} + 5mn - 2, w'_{1,2}\right),$$
$$\left(C', b'_{2,1} + 5mn - 2, w'_{2,1}\right), \left(C', b'_{2,2} + 5mn - 2, w'_{2,2}\right),$$
$$\left(C', b'_{2,1} + 5mn - 2, w'_{2,1}\right), \left(C', b'_{2,2} + 5mn - 2, w'_{2,2}\right), \ldots,$$
$$\left(C', b'_{2m\lceil \log 2m\rceil^2,1} + 5mn - 2, w'_{2m\lceil \log 2m\rceil^2,1}\right),$$
$$\left(C', b'_{2m\lceil \log 2m\rceil^2,2} + 5mn - 2, w'_{2m\lceil \log 2m\rceil^2,2}\right),$$
$$\left(C', b'_{2m\lceil \log 2m\rceil^2,1} + 5mn - 2, w'_{2m\lceil \log 2m\rceil^2,1}\right),$$
$$\left(C', b'_{2m\lceil \log 2m\rceil^2,2} + 5mn - 2, w'_{2m\lceil \log 2m\rceil^2,2}\right),$$
$$\left(C', b''_{1,1}, 1\right), \left(C', b''_{1,1}, 2\right), \ldots, \left(C', b''_{1,1}, m\right), \left(C', b''_{1,2}, 1\right), \left(C', b''_{1,2}, 2\right), \ldots,$$
$$\left(C', b''_{1,2}, m\right), \left(C', b''_{1,1}, 1\right), \left(C', b''_{1,1}, 2\right), \ldots, \left(C', b''_{1,1}, m\right), \left(C', b''_{1,2}, 1\right), \left(C', b''_{1,2}, 2\right),$$
$$\ldots, \left(C', b''_{1,2}, m\right), \left(C', b''_{2,1}, 1\right), \left(C', b''_{2,1}, 2\right), \ldots, \left(C', b''_{2,1}, m\right), \left(C', b''_{2,2}, 1\right),$$
$$\left(C', b''_{2,2}, 2\right), \ldots, \left(C', b''_{2,2}, m\right), \left(C', b''_{2,1}, 1\right), \left(C', b''_{2,1}, 2\right), \ldots, \left(C', b''_{2,1}, m\right),$$
$$\left(C', b''_{2,2}, 1\right), \left(C', b''_{2,2}, 2\right), \ldots, \left(C', b''_{2,2}, m\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 1\right),$$
$$\left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 1\right),$$
$$\left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 1\right),$$
$$\left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,1}, m\right), \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 1\right),$$
$$\left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, 2\right), \ldots, \left(C', b''_{5mn\lceil \log 5mn\rceil^2,2}, m\right), (C', 5mn - 4n + 1, 1),$$
$$(B, 1, 1), (C', 5mn - 4n + 1, 2), (B, 1, 2), \ldots, (C', 5mn - 4n + 1, m), (B, 1, m),$$
$$(C', 5mn - 4n + 2, 1), (B, 2, 1), (C', 5mn - 4n + 2, 2), (B, 2, 2), \ldots,$$
$$(C', 5mn - 4n + 2, m), (B, 2, m), \ldots, (C', 5mn, 1), (B, 4n, 1), (C', 5mn, 2),$$
$$(B, 4n, 2), \ldots, (C', 5mn, m), (B, 4n, m), (B, 1, 1), (B, 1, 1), (B, 1, 2), (B, 1, 2),$$
$$\ldots, (B, 1, m), (B, 1, m), (B, 2, 1), (B, 2, 1), (B, 2, 2), (B, 2, 2), \ldots, (B, 2, m),$$
$$(B, 2, m), \ldots, (B, 4n, 1), (B, 4n, 1), (B, 4n, 2), (B, 4n, 2), \ldots,$$
$$(B, 4n, m), (B, 4n, m))$$

The Access Pattern Does Not Leak Information

Now that we have constructed the access pattern, let us reduce it to the portion that can leak information. To begin with, we know we can remove all of the Batcher sorts since the steps made during the Batcher sort are determined only by the number of items being sorted (Section 2.5). We can remove the sort of C at the word level, the $5mn - 1$ Batcher sorts of adjacent buckets in C' and the final sort of C' by buckets, leaving us with an access pattern of:

$$((A,1,1),(C,1,1),(A,1,2),(C,1,2,),\ldots,(A,n,m),(C,n,m),(B,1,1),$$
$$(C,n+1,1),(B,1,2),(C,n+1,2),\ldots,(B,4n,m),(C,5n,m),(C,1,1),$$
$$(C,1,1),(C,1,2),(C,1,2),\ldots,(C,4n,m),(C,4n,m),(C,1,1),(C',1,1),$$
$$(C,1,2),(C',2,1),(C,1,3),(C',3,1),\ldots,(C,5n,m),(C',5mn,1),(C',1,1),$$
$$(C',1,m),(C',2,1),(C',2,1),(C',2,m),(C',3,1),\ldots,(C',5mn-1,1),$$
$$(C',5mn-1,m),(C',5mn,1),(C',5mn-4n+1,1),(B,1,1),$$
$$(C',5mn-4n+1,2),(B,1,2),\ldots,(C',5mn-4n+1,m),(B,1,m),$$
$$(C',5mn-4n+2,1),(B,2,1),(C',5mn-4n+2,2),(B,2,2),\ldots,$$
$$(C',5mn-4n+2,m),(B,2,m),\ldots,(C',5mn,1),(B,4n,1),(C',5mn,2),$$
$$(B,4n,2),\ldots,(C',5mn,m),(B,4n,m),(B,1,1),(B,1,1),(B,1,2),$$
$$(B,1,2),\ldots,(B,1,m),(B,1,m),(B,2,1),(B,2,1),(B,2,2),$$
$$(B,2,2),\ldots,(B,2,m),(B,2,m),\ldots,(B,4n,1),(B,4n,1),$$
$$(B,4n,2),(B,4n,2),\ldots,(B,4n,m),(B,4n,m))$$

We also know that the steps for copying between tables will be the same each time. This means we can remove the accesses for copying tables A and B to C, copying

C to C' and copying the last $4n$ buckets from C' to B, leaving us with:

$$((C,1,1),(C,1,1),(C,1,2),(C,1,2),\ldots,(C,4n,m),(C,4n,m),(C',1,1),$$
$$(C',1,m),(C',2,1),(C',2,1),(C',2,m),(C',3,1),\ldots,(C',5mn-1,1),$$
$$(C',5mn-1,m),(C',5mn,1),(B,1,1),(B,1,1),(B,1,2),(B,1,2),\ldots,$$
$$(B,1,m),(B,1,m),(B,2,1),(B,2,1),(B,2,2),(B,2,2),\ldots,(B,2,m),$$
$$(B,2,m),\ldots,(B,4n,1),(B,4n,1),(B,4n,2),(B,4n,2),\ldots,$$
$$(B,4n,m),(B,4n,m))$$

The accesses when we update each word in C with its tag created by h' will be the same each time as well, reading and then writing back every word in the table. Since the accesses are the same each time, it is not possible for them to leak information, leaving us with:

$$((C',1,1),(C',1,m),(C',2,1),(C',2,1),(C',2,m),(C',3,1),\ldots,$$
$$(C',5mn-1,1),(C',5mn-1,m),(C',5mn,1),(B,1,1),(B,1,1),(B,1,2),$$
$$(B,1,2),\ldots,(B,1,m),(B,1,m),(B,2,1),(B,2,1),(B,2,2),(B,2,2),\ldots,$$
$$(B,2,m),(B,2,m),\ldots,(B,4n,1),(B,4n,1),(B,4n,2),$$
$$(B,4n,2),\ldots,(B,4n,m),(B,4n,m))$$

We already removed the accesses from the Batcher sorts of C' when we compared adjacent buckets, but we still have the accesses remaining from when we looked at the buckets to determine their tags, which for the i^{th} pair of buckets would be: $((C',i,1),(C',i,m),(C',i+1,1))$. These accesses will also be the same each time, preventing them from leaking any information. This leaves the following as the

portion of the access pattern that can leak information:

$$((B,1,1),(B,1,1),(B,1,2),(B,1,2),\ldots,(B,1,m),(B,1,m),(B,2,1),$$
$$(B,2,1),(B,2,2),(B,2,2),\ldots,(B,2,m),(B,2,m),\ldots,(B,4n,1),$$
$$(B,4n,1),(B,4n,2),(B,4n,2),\ldots,(B,4n,m),(B,4n,m))$$

This remaining portion of the access pattern is from the scan of B to remove the dummy entries at the end of the oblivious sort. Again, these steps are exactly the same each time: reading and then writing back each word in the table. Because these steps are the same each time, they cannot leak any information as well, leaving us with no accesses that can leak information. Therefore, this algorithm provides oblivious rehashing of the tables.

5.2.3 Hashes are Independent

It remains to be shown that when we rehash a level, the new hash function we select is independent of the previous hash functions. In the hierarchical model, level i has a hash function index s_i associated with the level. The hash function created by this index is defined as:

$$h_{s_i}(a) = RO(s_i \cdot a) \pmod{4^i} \tag{5.2}$$

where $RO(\cdot)$ is the oracle given to the RAM. Each time level i is rehashed, a new value for s_i is chosen uniformly at random. Since s_i is chosen uniformly at random each time, the hash function h_{s_i} is independent of all of the previous hash functions selected for this level, giving us oblivious simulation by using the hierarchical model.

5.3 Analysis

There is one key assumption in the hierarchical solution: at no point is there a bucket overflow in any of the hash tables. With the exception of the hash table at level one, the only time one of the lower hash tables has data written to it is during the rehashing of the tables. If a bucket overflows during this step, it will happen during the pass over C' when the adjacent buckets are sorted. In this case, words tagged with the same value will end up in both buckets. The algorithm, as written, does not detect this case, however it could be easily modified to do so. At the end of the rehashing, we scan B to remove the dummy entries. During this scan we could check for overflows. If an overflow has occurred, we would need to choose a new hash function and perform the rehash again or abort the simulation. As pointed out by Kushilevitz et. al., this has the potential to leak information [7]. Since then, a number of solutions have been proposed that reduce the risk of a bucket overflow and improve efficiency, which we will discuss in Chapter 6.

Chapter 6

VARIATIONS ON THE HIERARCHICAL SOLUTION

After the initial solutions to the oblivious RAM problem were published by Goldreich and Ostrovsky [3, 8], many variations on the hierarchical solution were proposed. The vast majority of these variations focused on making the solution more feasible in practice. Improvements were focused along two key tracks: decreasing the amount of memory required compared to the original program and making simulated accesses faster. Both of these areas could be improved by removing the buckets from the hash tables, which is the main focus of the solutions we will discuss here. The buckets were removed by adding in new features such as Bloom filters [11] or swapping out cuckoo hash tables for the bucket hash tables [5, 10].

6.1 Using Bloom Filters to Improve Efficiency

One of the earlier improvements to the hierarchical model was done by incorporating Bloom filters into each level [11]. Williams, Sion and Carbunar point out that the existing oblivious RAM solutions at the time have a relatively high computational overhead and can significantly slow down the execution of the program being simulated. In response, they took the basic hierarchical solution model and associated a Bloom filter with the hash table for each level. This allows the CPU to securely determine, in advance, whether or not the item they are looking for is stored at some particular level. Using the Bloom filter reduces the amortized cost per query to $O(\log n \log \log n)$. Additionally, the amount of memory needed by the

MEM portion of the RAM is reduced as well to $O(n)$.

The primary cost reduction comes from the elimination of buckets in the hash table and just having a flat table of 4^i elements for level i. In the original hierarchical solution, if buckets were not used in the hash table, the adversary could potentially learn whether or not an item was found in a particular level. Here, the CPU uses the Bloom filter to determine whether or not the item being searched for is at the level in question, and if it is not, looks up a dummy element at that level instead. Like the original solution, the accesses to each level will appear random and unique between reshuffles, but by the elimination of buckets, reduces the number of look ups at each level from $O(\log n)$ to $O(1)$. The rest of the simulation of an access remains the same. The CPU starts by scanning the entire cache (level 1 in the original hierarchical solution), then scanning a unique element at each of the subsequent levels (either dummy elements or the real element once we find the level that it is in). Once all of the levels have been accessed, the item is inserted at the top level, with its value updated if a write access was being simulated.

Like the original solution, the adversary will see the same pattern of accesses each time and will not be able to distinguish them from random accesses to each level since each item is only accessed once between reshuffles of the level. The reshuffle for this solution is similar in nature to the original hierarchical solution, but differs in some key ways because of the removal of buckets and the addition of the Bloom filter. The Bloom filter, as it is stored on the server, needs to be constructed in an oblivious manner to prevent information leakage.[1]

Additionally, the two levels need to be combined in a way that does not leak information as well. Since we have removed the buckets from this solution, it is actually easier to combine the two levels. When level $i-1$ is being reshuffled into level i, the items for both levels are moved into a temporary buffer on the server. Once there, the items are randomly permuted using the Oblivious Scramble

[1]For the details of how to obliviously construct a Bloom filter, see [11].

Algorithm [11] and moved back into level i. The scramble is more efficient that the oblivious sort of the original solution, requiring $O(m \log \log m)$ time and $O(\sqrt{m})$ private storage on the CPU, where m is the size of the level being reshuffled.

Going a step further than earlier papers, Williams, Sion and Carbunar implemented their solution using Java on a computer running Redhat Fedora [11]. Their performance analysis showed that this particular solution shows promise as a real world application. Outside of reshuffles, most queries completed within a few hundred milliseconds. For small, fast computations this overhead may be impractical, but for larger data retrievals or data being sent across a network instead of a local physical device, this slow down would not be catastrophic.

6.2 Improvements Using MapReduce and Cuckoo Hashing

Goodrich and Mitzenmacher proposed a variation on the hierarchical solution that uses cuckoo hashing and the MapReduce paradigm [4]. A cuckoo hash table [9] is actually two hash tables T_1 and T_2, each associated with their own hash function h_1 and h_2. When an item x is inserted into the hash table, it is put into T_1 at location $h_1(x)$. If there is an item y already in that location, x is put in and y is moved into T_2 at location $h_2(y)$. If an item is already there, y is inserted in that location and the item there is moved back into T_1. This continues until the final item is placed in an empty location and nothing needs to be bumped or the insertion has run for too long and a failure is reported. The expected time to insert a new item is constant and should succeed with high probability.

The MapReduce paradigm has become a popular method of parallel computing [5]. MapReduce provides a way to divide up data for parallel computations. Goodrich and Mitzenmacher created a MapReduce algorithm for inserting items into a cuckoo hash table [4]. This along with a new algorithm for sorting external memory (a k-way modular mergesort) provided significant improvements to the base hierarchical solution.

In this solution, the high level hash tables retain their buckets and are used the same as the base solution but the larger hash tables become cuckoo hash tables. Dummy elements are used as well, like the square root solution. More formally, the highest level hash table, H_k is used as a flat array. Hash tables H_{k+1} to H_l, for some constant l are bucket hash tables and H_{l+1} to H_L are cuckoo hash tables. Each begins as an empty table with 2^i dummy elements at level i, indexed by the values -1 to -2^i, and a counter d_i initialized to zero and incremented each access.

When an access for address a is being simulated, level H_k is scanned completely. For levels H_{k+1} to H_l, if the item has not been found bucket $h_i(a)$ is scanned; otherwise $h_i(-d_i)$ is scanned. For the cuckoo hash levels H_{l+1} to H_L, if the item has not been found, both tables inside the cuckoo hash table are accessed ($h_{i_1}(a)$ in the first and $h_{i_2}(a)$ in the second). If the item has been found, then we access a dummy element at $h_{i_1}(d_i)$ in the first table and $h_{i_2}(d_i)$ in the second. Once all of the tables have been accessed, the (potentially updated) value is inserted in the top level table.

Like the hierarchical solution, all of the levels are rehashed at regular intervals. Here, each level is rehashed every 2^i accesses, which is slightly more frequent that the original solution. However, since the larger levels contain less data since they are cuckoo hash tables instead of bucket hash tables, less data is being rehashed so the overhead is reduced. In this solution, the amortized cost per access is $O(\log n)$ and the storage overhead is $O(n)$.

6.3 Switching Completely to Cuckoo Hashing

As had been pointed out in previous papers, Pinkas and Reinman begin by pointing out that the merely theoretical solutions proposed by Goldreich and Ostrovsky have such high overheads that they are more theoretical than practical constructs [10]. Instead, they set out to design a variation that was practical enough for real world use. Like the other variations, they based theirs on the basic hierarchical solution.

Instead of using hash tables with buckets, this solution uses cuckoo hash tables for all of the levels. Unlike the previous solution (Section 6.2) which uses two hash tables, each with their own hash function, for each level, this solution uses one flat hash table indexed by both hash functions. In this construction, when item x is inserted into the cuckoo hash table, it is inserted at location $h_1(x)$. If item y is already at that location, it is moved to location $h_2(y)$. If an item is already at that location, it is bounced out and put in the spot specified by h_1 and so on, until the chain has ended or it has run for too long.

In this solution, level i is a flat array of $4 \cdot 2^i$ entries which may contain up to 2^i program addresses. Additionally, there may be up to 2^i dummy elements in the level. Every 2^{i-1} simulated accesses, the level is rehashed. Initially, each level is empty and a counter that is incremented each simulated access is started at zero. To simulate an access for program address a, the CPU begins incrementing the counter and then scanning every item in the first level. For the remaining levels, if the address has not been found yet, the CPU looks at entries $h_{i_1}(a)$ and $h_{i_2}(a)$ for level i. If the address has already been found, then the CPU looks at two dummy locations, $h_{i_1}(dummy \cdot count)$ and $h_{i_2}(dummy \cdot count)$, where *count* is our counter. Once all of the levels have been accessed, the top level is rescanned and the (potentially updated) value for a is written into this level.

While this solution may seem like an obvious improvement to the hierarchical model by using smaller hash tables, Kushilevitz, Lu and Ostrovsky point out a key flaw that allows an adversary to distinguish access patterns [7]. Because the hash tables start empty in this solution and are not encrypted so the adversary can see the empty elements, the adversary can distinguish access patterns with high probability. Suppose that the CPU is searching for address a but has not found it yet. When it gets to level i, it will look at locations $h_{i_1}(a)$ and $h_{i_2}(a)$. If a is not stored at this level currently, there is a significant chance that both locations it hashes to are empty. If the adversary sees an access to two empty locations,

it knows the program word being searched for has not been found yet (if it was already found, we would be looking for a dummy element that exists in at least one of those two locations). By seeing accesses to two empty locations within a single level, the adversary could use this information to distinguish between two different access patterns. In practice, this may not leak enough information for the adversary to learn something useful, but it does allow the adversary to win at the oblivious RAM security game (Section 3.2.1).

Chapter 7

CONCLUSION

There is a real need for oblivious simulation as we have seen; whether that be cloud computing, physically secure hardware units or web based applications [2, 4, 8]. Since the original solutions were proposed by Ostrovsky and Goldreich [3, 8], later work has primarily focused on variations of the hierarchical solution. Most of this work has focused on decreasing the cost of oblivious simulation. In this paper, we proposed a variation of the square root solution with a significantly lower overhead for each simulated pass.

While it is important to lower the cost of oblivious simulation, the formalization of the problem still leaves a lot to be desired. We have proposed several formal security games using the definition of oblivious simulation proposed by Ostrovsky and Goldreich [3, 8] but the definitions are not truly complete. Using their definition, we do not have a realistic way to determine if two inputs create an access pattern with the same distribution. In the future, it will be extremely important to flesh out the definition and the security games so that solutions can be formally measured in terms of how well they provide oblivious simulation.

Despite the lack of a fully formalized problem, we predict that the notion of oblivious simulation will become important over the years to come as malicious entities try to find ways to learn information about encrypted data. Many companies are beginning to use the Internet as a key piece of their software, opening them up to any number of adversaries looking to get information about their program. By using an IND-CPA secure encryption scheme and oblivious simulation, we have shown that one can obtain a high level of protection for software. While

the overhead may be prohibitively high for many applications, this will be a useful technique for sensitive applications.

REFERENCES

[1] Kenneth Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[2] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 191–206, 2010.

[3] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, May 1996.

[4] Michael Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, pages 576–587, 2011.

[5] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948, 2010.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 1998.

[7] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. *Cryptology ePrint Archive*, Report 2011/327, 2011.

[8] Rafial Ostrovsky. *Software Protection and Simulation on Oblivious RAMs*. PhD thesis, MIT, 1992.

[9] Rasmus Pagh and Flemming Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.

[10] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *CRYPTO*, pages 502–519, 2010.

[11] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148, 2008.

Appendix A

SOFTWARE PROTECTION

As the software market becomes more competitive, companies are becoming more concerned with protecting their intellectual property rights. Preventing competitors from learning information about your algorithms is a key piece in protecting your software. Ostrovsky dubbed this *software protection*, and it was a major component of his dissertation [8]. If an adversary is given a compiled program and allowed to run it, the compiler is said to provide software protection if the adversary is only able to learn the bounds for running time and memory usage as well as which outputs correspond to which inputs. The adversary is allowed to run the program as many times as it wants, within a reasonable time bound, and may potentially be able to tamper with the messages sent from the CPU to the MEM.

We are going to expand our notion of the RAM definition a bit for the concept of software protection. Recall from Section 3.1, that $RAM_k = (CPU_k, MEM_k)$, for $k \in \mathbb{N}$, is a family of types where $CPU_k = ITM_{(k,k)}$ and $MEM_k = ITM_{(k,2^k k)}$. An instance of RAM_k is started on some input (s, y), where s is the input for CPU_k and y is the input for MEM_k. For the purposes of defining software protection, the input to MEM_k is going to be the tuple (Π, x), where Π is the program the RAM_k is going to run and x is the input to the program. Our input to RAM_k is now $(s, (\Pi, x))$.

A.1 Software Protection Security Game

Suppose we have some compiler C which takes as input an integer k and some program Π and produces a pair (f, Π_f) where f is a randomly selected Boolean function ($f : \{0,1\}^* \rightarrow \{0,1\}$) and Π_f is the encrypted program. Additionally, $|\Pi_f|$ is of the order $|\Pi|$ ($|\Pi_f| = O(|\Pi|)$) and for some $k' = k + O(\log k)$ there exists an oracle $RAM_{k'}$ such that for all possible programs Π, all possible functions f, all strings $x \in \{0,1\}^*$ and all strings $s \in \{0,1\}^*$, $RAM_{k'}(s, (\Pi_f, x)) = \Pi(x)$.

We also have a specification oracle for the program Π that on an input $x = \{0,1\}^*$, the oracle returns $(\Pi(x), t_\Pi(x), s_\Pi(x))$, where $\Pi(x)$ is the output for program Π running on input x, $t_\Pi(x)$ is the running time of the program on input x and $s_\Pi(x)$ is the amount of memory used by the program while running on input x.

Suppose we have two adversaries, A and B. Both adversaries run in about the same time, within some constant factor. A is given as input the compiled program Π_f and is given access to an oracle that contains $RAM^f_{k'}$. A is allowed to call the oracle as many times as it wants with different inputs (inputs would be of the form $(s, (\Pi_f, x))$). B is given as input k' and the order of the size of Π_f, namely $O(|\Pi|)$ and has access to the specification oracle for Π, $spec_\Pi$. B is allowed to query its oracle as many times as it wants as well. After both adversaries have finished executing, they return all of the information they were able to learn about the program as a string (out_A for A and out_B for B). Suppose we have a third adversary D, which is our adversary in the software protection security game (see Figure A.1). In World 0, D runs adversary B (see Figure A.2), and in World 1, D runs adversary A (see Figure A.3). D must look at the information returned by the adversary it ran and determine which world it is in. If D is able to guess with high probability which world that it is in, then adversary A must have been able to learn information about the program it was running in addition to running

time/space bounds and input/output relationships. Either A must have been able to learn information from the encrypted program Π_f or from the access patterns generated by watching the $RAM_{k'}$ execute Π_f.

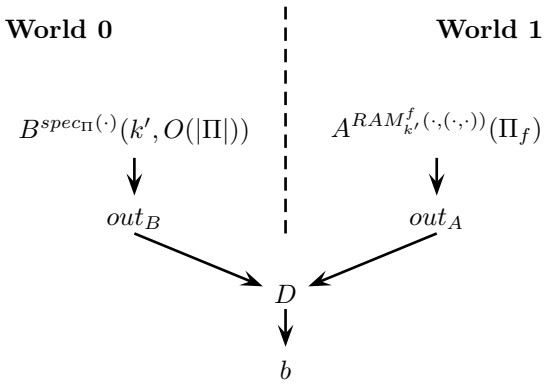

Figure A.1: The Software Protection Security Game

Experiment $\mathbf{Exp}^{\text{sp-0}}_C(D)$:
$out_B \leftarrow B^{spec_\Pi(\cdot)}(k', O(|\Pi|))$
$b \xleftarrow{\$} D(out_B)$
return b

Figure A.2: The Software Protection Experiment in World 0

Experiment $\mathbf{Exp}^{\text{sp-1}}_C(D)$:
$out_A \leftarrow A^{RAM^f_{k'}(\cdot,(\cdot,\cdot))}(\Pi_f)$
$b \xleftarrow{\$} D(out_A)$
return b

Figure A.3: The Software Protection Experiment in World 1

The advantage of the adversary D (a measure of how well it does) is defined as the probability that it actually guesses it is in World 1 when it is in World 1 minus the probability that it guesses it is in World 1 when it is really in World 0:

$$\mathbf{Adv}^{\text{sp}}_C(D) = \Pr\left(\mathbf{Exp}^{\text{sp-1}}_C(D) = 1\right) - \Pr\left(\mathbf{Exp}^{\text{sp-0}}_C(D) = 1\right) \qquad \text{(A.1)}$$

A.2 Encryption and Oblivious Simulation Give Software Protection

We will begin by reviewing the original reduction of Ostrovsky's from software protection to oblivious RAM simulation. This reduction uses the Boolean function f given to the RAM by the compiler to provide encryption of the items being stored on the MEM component. After explaining the reduction, we will redo the formulation of the software protection game to use an IND-CPA secure encryption scheme in place of f and redo the reduction using this new definition.

A.2.1 The Original Reduction

We will start by reviewing the reduction for an adversary who does not tamper with the messages sent between the CPU and the MEM. Suppose we have some RAM_k which provides oblivious simulation for some universal $RAM_{k'}$. If the original $RAM_{k'}$ took time t' to complete, RAM_k runs in time t that is some polynomial factor of t' ($t = t' \cdot g(t)$). If such a RAM_k exists, then there is a compiler C that protects against non-tampering adversaries with an overhead bounded by $O(g(t))$.

Recall from our RAM model (Section 3.1) that there are three fields in the messages sent between the CPU and the MEM: (a, i, v) where $a \in \{0,1\}^k$ is an address on the MEM work tape, $i \in \{read, write, halt\}$ is an instruction and $v \in \{0,1\}^k$ is a value. For the compiler to provide software protection, none of these pieces can leak information. Since we are running the compiled program on a RAM_k that provides oblivious simulation, we know by definition that the address field cannot leak any information (otherwise the RAM would not provide oblivious simulation). This leaves the instruction and value field. To prevent these fields from leaking information, we modify the CPU component of our RAM_k. For the value field, we use the function f to encrypt the values being stored on the MEM. Encryption is provided by constructing a new function $f'(\cdot,\cdot)$ from $f(\cdot)$. On inputs *encount* and *length*, f' returns a bit string that is *length* bits long and

uses *encount* as its beginning input to f (Figure A.4). The CPU keeps a counter *encount* which is initialized to 0 and is incremented by the length of v after each access. When a value is being sent to the MEM, instead of sending the original value v, it is replaced by the tuple $(v \oplus f(encount), encount)$. Finally, to prevent the instruction from leaking information, for each read or write instruction execute by the CPU, the CPU will first send a read instruction and then a write instruction. Because the value is reencrypted each write, the adversary will not be able to tell whether or not the CPU was interested in the read or write instruction.

```
f'(encount, length):
        s ← f(encount)
        For i = 1, ..., length − 1
                s ← s||f(encount + i)
        Return s
```

Figure A.4: Definition of f'

By preventing all of the fields from leaking information, the only information the adversary will be able to learn is the running time of the program on each input, the amount of memory it uses and the output corresponding to that input.

A.2.2 Reformulation of the Software Protection Game

Let us begin by redefining how our compiler C works. In this new version, the compiler takes as input an integer k', a program Π and an IND-CPA secure encryption scheme $\mathcal{ES} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and returns the tuple $(K, \mathcal{E}_K(\Pi))$, where K is a key generated by $\mathcal{K}$ and $\mathcal{E}_K(\Pi)$ is the encrypted program. For k' in the order of $k + O(\log k)$, there exists an oblivious $RAM_{k'}$ given access to K, $\mathcal{E}$, $\mathcal{D}$ such that for all possible programs Π, all possible keys K, and all inputs $s \in \{0,1\}^*$ and $x \in \{0,1\}^*$, $RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}(s, (\mathcal{E}_K(\Pi), x)) = \Pi(x)$.

We will modify our original adversaries as follows. Adversary A is given access to the encrypted program $\mathcal{E}_K(\Pi)$ and an oracle that contains $RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}$ (a

RAM given access to the encryption and decryption algorithms for our encryption scheme). Adversary A can call the oracle with as many inputs as it likes (this time the inputs would be of the form $(s, (\mathcal{E}_K(\Pi), x))$). Adversary B is given inputs k' and $|\mathcal{E}_K(\Pi)|$ and has access to the specification oracle for Π. The rest of the setup remains the same. Both adversaries are allowed to query the oracle with inputs of their choice and at the end of their execution must output everything they have learned about the program. The final adversary, D, gets one of these strings and must guess which world it is in (Figure A.5).

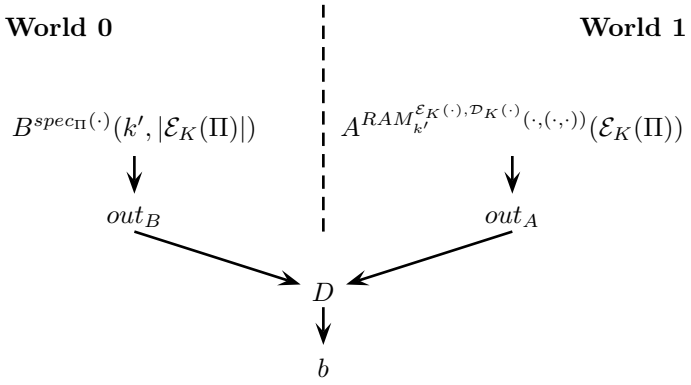

Figure A.5: The Modified Software Protection Security Game

Our experiments in this world are slightly redefined to account for the use of the encryption scheme (Figures A.6, A.7).

Experiment $\mathbf{Exp}_C^{\text{sp-ind-cpa-0}}(D)$:
$out_B \leftarrow B^{spec_\Pi(\cdot)}(k', |\mathcal{E}_K(\Pi)|)$
$b \xleftarrow{\$} D(out_B)$
return b

Figure A.6: The Software Protection Experiment with an IND-CPA Encryption Scheme in World 0

Experiment $\mathbf{Exp}_C^{\text{sp-ind-cpa-1}}(D)$:
$out_A \leftarrow A^{RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}(\cdot,(\cdot,\cdot))}(\mathcal{E}_K(\Pi))$
$b \xleftarrow{\$} D(out_A)$
return b

Figure A.7: The Software Protection Experiment with an IND-CPA Encryption Scheme in World 1

The advantage of the adversary D is the same as our original definition:

$$\mathbf{Adv}_C^{\text{sp}-\text{ind}-\text{cpa}}(D) = \Pr\left(\mathbf{Exp}_C^{\text{sp-ind-cpa-1}}(D) = 1\right) - \Pr\left(\mathbf{Exp}_C^{\text{sp-ind-cpa-0}}(D) = 1\right) \tag{A.2}$$

A.2.3 The New Reduction

Given some compiler C and an adversary D, as described in the previous section, where the advantage of the adversary against the compiler is bound by ε:

$$\mathbf{Adv}_C^{\text{sp}-\text{ind}-\text{cpa}}(D) < \varepsilon \tag{A.3}$$

$$\Pr\left(\mathbf{Exp}_C^{\text{sp-ind-cpa-1}}(D) = 1\right) - \Pr\left(\mathbf{Exp}_C^{\text{sp-ind-cpa-0}}(D) = 1\right) < \varepsilon \tag{A.4}$$

$$\Pr\left(D(out_A) = 1\right) - \Pr\left(D(out_B) = 1\right) < \varepsilon \tag{A.5}$$

Suppose we create a new security game (see Figure A.8) based on the original where the adversary D is also given access to the left-right encryption oracle from the IND-CPA security game (see Section 2.4) and the access pattern oracle from the ORAM security game (see Section 3.2.1). Like the original games, the adversary can query both oracles repeatedly with the inputs of its own choosing.

We can modify Equation A.3 as follows:

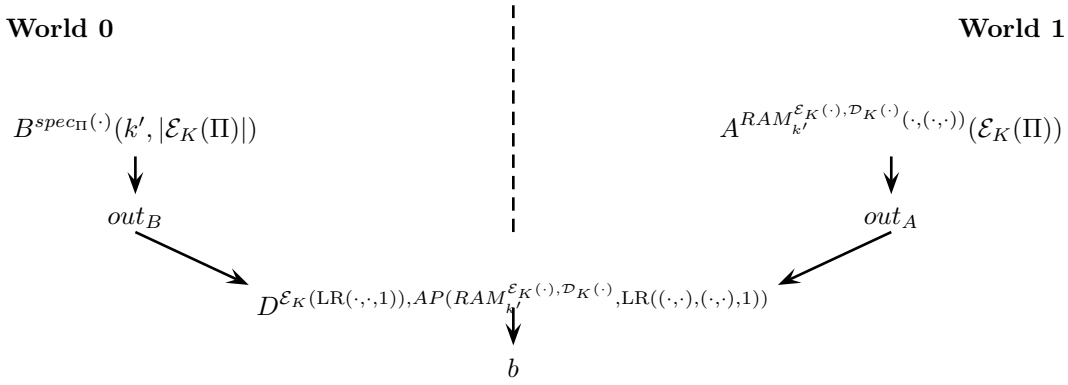

Figure A.8: The Modified Software Protection Security Game

$$\begin{aligned} &\Pr\left(D(out_A)=1\right) - \\ &\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_A)=1\right) + \\ &\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_B)=1\right) - \\ &\Pr\left(D(out_B)=1\right) < \varepsilon \end{aligned} \tag{A.6}$$

Let us start by considering the difference between the following probabilities:

$$\begin{aligned} &\Pr\left(D(out_A)=1\right) \\ &\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_A)=1\right) \end{aligned}$$

In World 1, A is given full access to the $RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}$ that is running the programs. Any information that A learns by watching execution will be included in out_A. Because A already sees the access patterns, giving D the access pattern oracle for $RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}$ does not allow it to learn any additional information. Thus, if D is able to learn any additional information, it must be from queries to the left-right encryption oracle (for example, it could learn information about the compiled program given to adversary A). Therefore, the difference between these two probabilities is bounded by the advantage for some IND-CPA adversary

E against our encryption scheme $\mathcal{ES}$:

$$\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_A)=1\right) - \Pr\left(D(out_A)=1\right) < \mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(E) \tag{A.7}$$

Next, consider the difference between the probabilities:

$$\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_B)=1\right) \Pr\left(D(out_B)=1\right)$$

In this case, giving D access to the left-right encryption oracle does not allow it to learn any additional information because neither adversary D or B see any encrypted information in this game. Therefore, any additional information that D learns must come from its access to the access pattern oracle. Thus, the difference between these two probabilities is bounded by the advantage for some ORAM adversary F:

$$\Pr\left(D^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),AP(RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)},\mathrm{LR}((\cdot,\cdot),(\cdot,\cdot),1))}(out_B)=1\right) - \Pr\left(D(out_B)=1\right) < \mathbf{Adv}^{\text{oram}}_{RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}}(F) \tag{A.8}$$

Thus, the software protection advantage is bounded by the combination of the advantages of the ORAM and IND-CPA schemes used:

$$\mathbf{Adv}_C^{\text{sp}-\text{ind}-\text{cpa}}(D) < \mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(E) + \mathbf{Adv}^{\text{oram}}_{RAM_{k'}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot)}}(F) \tag{A.9}$$

Appendix B

EXAMPLE SQUARE ROOT SOLUTION SIMULATION

It may be useful for the reader to see an example of a RAM being simulated by an oblivious RAM. In this section we will describe a RAM and show a portion of its execution. We will then describe the oblivious RAM and show how it simulates the execution of the original RAM.

B.1 The Original RAM

Suppose we have some RAM_4 in the middle of execution with the work tape of the MEM_4 component (of length 2^4) having the state shown in Figure B.1.

2	12	5	6	1	5	4	6	29	17	11	1	19	23	11	9

Figure B.1: Starting State of RAM_4

We are going to show eight execution steps on this RAM:

$$((read, 7, \perp), (write, 8, 28), (write, 2, 15), (write, 7, 14),$$
$$(read, 9, \perp), (read, 11, \perp), (write, 8, 22), (read, 4, \perp))$$

To begin with, the CPU_4 is going to send the message $(read, 7, \perp)$ to MEM_4. MEM_4 will read the value at the seventh word on its work tape and send a message of 4 back to CPU_4. Because this is a read operation, the work tape is not updated (see Figure B.2).

2	12	5	6	1	5	4	6	29	17	11	1	19	23	11	9

Figure B.2: $(read, 7, \perp)$

Next, the message $(write, 8, 28)$ is sent to MEM_4. The eighth word on MEM_4's work tape is updated to 28 and the same value is returned to CPU_4 (see Figure B.3).

2	12	5	6	1	5	4	28	29	17	11	1	19	23	11	9

Figure B.3: $(write, 8, 28)$

The remaining reads and writes behave the same as the previous ones (see Figures B.4 to B.9).

2	15	5	6	1	5	4	28	29	17	11	1	19	23	11	9

Figure B.4: $(write, 2, 15)$

2	15	5	6	1	5	14	28	29	17	11	1	19	23	11	9

Figure B.5: $(write, 7, 14)$

2	15	5	6	1	5	14	28	29	17	11	1	19	23	11	9

Figure B.6: $(read, 9, \perp)$

2	15	5	6	1	5	14	28	29	17	11	1	19	23	11	9

Figure B.7: $(read, 11, \perp)$

2	15	5	6	1	5	14	22	29	17	11	1	19	23	11	9

Figure B.8: $(write, 8, 22)$

2	15	5	6	1	5	14	22	29	17	11	1	19	23	11	9

Figure B.9: $(read, 4, \perp)$

B.2 Oblivious Simulation of the Original RAM

Our oblivious RAM simulating the original RAM_4 will be a RAM_5 that contains a MEM_5 which has a work tape of length 2^5. However, our oblivious RAM_5 will only need to use the first 24 words on the tape since the original RAM has a work tape of length 16 ($16 + 2\sqrt{16} = 24$). We will only depict the words we are using during simulation. Suppose that the eight steps we showed in Section B.1 begin at the start of a new pass in the simulation and the work tape for the MEM_5 has the state shown in Figure B.10 at the end of the previous pass. For the sake of simplicity, we are again omitting that every value written to the work tape is encrypted by the CPU before being sent to the MEM.

Before beginning our first pass, we will need to construct our function f for this pass. In this simulation, f will map [20] to $\mathbb{Z}_{49}$. For this particular pass, suppose that the f we construct using the random oracle gives the values as shown in Table B.1.

Once we have generated f, we will need to update the non-shelter locations with their new tags (Figure B.11).

$(9, 1, 2)$	$(17, 2, 12)$	$(14, 3, 5)$	$(15, 4, 6)$	$(34, 5, 1)$	$(11, 6, 5)$	$(27, 7, 4)$	$(26, 8, 6)$
$(41, 9, 29)$	$(18, 10, 17)$	$(7, 11, 11)$	$(33, 12, 1)$	$(23, 13, 19)$	$(39, 14, 23)$	$(4, 15, 11)$	$(47, 16, 9)$
$(46, 17, d)$	$(30, 18, d)$	$(25, 19, d)$	$(48, 20, d)$	$(11, \infty, 21)$	$(41, \infty, 29)$	$(14, \infty, 3)$	$(23, \infty, 19)$

Figure B.10: Initial State of the Oblivious RAM_5

i	$f(i)$
1	19
2	43
3	14
4	27
5	42
6	32
7	46
8	12
9	45
10	29
11	34
12	6
13	28
14	35
15	5
16	21
17	36
18	30
19	44
20	25

Table B.1: $f : [20] \to \mathbb{Z}_{49}$ for the First Pass

Once we have updated the tags, we will need to perform a Batcher sort across the non-shelter locations based on the value each word is tagged with. A Batcher sort for 20 elements will generate the following comparisons: **Order**$(1, 2)$, **Order**$(1, 2)$, **Order**$(4, 5)$, **Order**$(1, 3)$, **Order**$(2, 4)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(6, 7)$, **Order**$(6, 7)$, **Order**$(9, 10)$, **Order**$(6, 8)$, **Order**$(7, 9)$, **Order**$(7, 8)$, **Order**$(9, 10)$, **Order**$(1, 9)$, **Order**$(5, 13)$, **Order**$(5, 9)$, **Order**$(3, 7)$, **Order**$(3, 5)$, **Order**$(7, 9)$, **Order**$(2, 6)$, **Order**$(4, 8)$, **Order**$(4, 6)$, **Order**$(8, 10)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(6, 7)$, **Order**$(8, 9)$, **Order**$(11, 12)$, **Order**$(11, 12)$,

$(19, 1, 2)$	$(43, 2, 12)$	$(14, 3, 5)$	$(27, 4, 6)$	$(42, 5, 1)$	$(32, 6, 5)$	$(46, 7, 4)$	$(12, 8, 6)$
$(45, 9, 29)$	$(29, 10, 17)$	$(34, 11, 11)$	$(6, 12, 1)$	$(28, 13, 19)$	$(35, 14, 23)$	$(5, 15, 11)$	$(21, 16, 9)$
$(36, 17, d)$	$(30, 18, d)$	$(44, 19, d)$	$(25, 20, d)$	$(11, \infty, 21)$	$(41, \infty, 29)$	$(14, \infty, 3)$	$(23, \infty, 19)$

Figure B.11: Updated Tags

Order$(14, 15)$, **Order**$(11, 13)$, **Order**$(12, 14)$, **Order**$(12, 13)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(16, 17)$, **Order**$(19, 20)$, **Order**$(16, 18)$, **Order**$(17, 19)$, **Order**$(17, 18)$, **Order**$(19, 20)$, **Order**$(11, 19)$, **Order**$(15, 23)$, **Order**$(15, 19)$, **Order**$(13, 17)$, **Order**$(13, 15)$, **Order**$(17, 19)$, **Order**$(12, 16)$, **Order**$(14, 18)$, **Order**$(14, 16)$, **Order**$(18, 20)$, **Order**$(12, 13)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(18, 19)$, **Order**$(1, 17)$, **Order**$(9, 25)$, **Order**$(9, 17)$, **Order**$(5, 13)$, **Order**$(5, 9)$, **Order**$(13, 17)$, **Order**$(3, 19)$, **Order**$(11, 27)$, **Order**$(11, 19)$, **Order**$(7, 15)$, **Order**$(7, 11)$, **Order**$(15, 19)$, **Order**$(3, 5)$, **Order**$(7, 9)$, **Order**$(11, 13)$, **Order**$(15, 17)$, **Order**$(2, 18)$, **Order**$(10, 26)$, **Order**$(10, 18)$, **Order**$(6, 14)$, **Order**$(6, 10)$, **Order**$(14, 18)$, **Order**$(4, 12)$, **Order**$(8, 16)$, **Order**$(8, 12)$, **Order**$(16, 20)$, **Order**$(4, 6)$, **Order**$(8, 10)$, **Order**$(12, 14)$, **Order**$(16, 18)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(6, 7)$, **Order**$(8, 9)$, **Order**$(10, 11)$, **Order**$(12, 13)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(18, 19)$. For each comparison, the two values will both be read and then written back. After all of the order operations have been performed, the work tape will be as shown in Figure B.12.

$(5, 15, 11)$	$(6, 12, 1)$	$(12, 8, 6)$	$(14, 3, 5)$	$(19, 1, 2)$	$(21, 16, 9)$	$(25, 20, d)$	$(27, 4, 6)$
$(28, 13, 19)$	$(29, 10, 17)$	$(30, 18, d)$	$(32, 6, 5)$	$(34, 11, 11)$	$(35, 14, 23)$	$(36, 17, d)$	$(42, 5, 1)$
$(43, 2, 12)$	$(44, 19, d)$	$(45, 9, 29)$	$(46, 7, 4)$	$(11, \infty, 21)$	$(41, \infty, 29)$	$(14, \infty, 3)$	$(23, \infty, 19)$

Figure B.12: Batcher Sort of Non-Shelter Locations

count is initialized to zero and we begin the simulation of the next four accesses:

$$((read, 7, \perp), (write, 8, 28), (write, 2, 15), (write, 7, 14))$$

When simulating access $(read, 7, \perp)$, we begin by scanning the entire shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Since program address 7 was not found in the shelter, we will perform a binary search to find it in one of the non-shelter locations using the knowledge that $f(7) = 46$:

$$((read, 10, \perp), (read, 15, \perp), (read, 17, \perp), (read, 19, \perp), (read, 20, \perp))$$

Once we find program address 7 at physical address 20, we update the word at address 20 to be $(46, \infty, 4)$. Then we scan the shelter, reading and writing each value. Since program address 7 does not already exist in the shelter, we write it to the first empty word, physical word 21 (see Figure B.13):

$$((read, 21, \perp), (write, 21, (46, 7, 4)), (read, 22, \perp), (write, 22, (41, \infty, 29)),$$
$$(read, 23, \perp), (write, 23, (14, \infty, 3)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

$(5, 15, 11)$	$(6, 12, 1)$	$(12, 8, 6)$	$(14, 3, 5)$	$(19, 1, 2)$	$(21, 16, 9)$	$(25, 20, d)$	$(27, 4, 6)$
$(28, 13, 19)$	$(29, 10, 17)$	$(30, 18, d)$	$(32, 6, 5)$	$(34, 11, 11)$	$(35, 14, 23)$	$(36, 17, d)$	$(42, 5, 1)$
$(43, 2, 12)$	$(44, 19, d)$	$(45, 9, 29)$	$(46, \infty, 4)$	$(46, 7, 4)$	$(41, \infty, 29)$	$(14, \infty, 3)$	$(23, \infty, 19)$

Figure B.13: Simulate $(read, 7, \perp)$

count is incremented to 1 and we simulate the next access: $(write, 8, 28)$. Again,

we begin by reading the entire shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Since we did not find the word in the shelter, we perform a binary search across the non-shelter locations using the tag $f(8) = 12$:

$$((read, 10, \perp), (read, 5, \perp), (read, 3, \perp))$$

After locating it in physical word 3, we rewrite the value at that location:

$$(write, 3, (12, \infty, 6))$$

Next, we scan the shelter again. Since the word was not found in the shelter, we write it into the first empty location we find (see Figure B.14):

$$((read, 21, \perp), (write, 21, (46, 7, 4)), (read, 22, \perp), (write, 22, (12, 8, 28)),$$
$$(read, 23, \perp), (write, 23, (14, \infty, 3)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

$(5, 15, 11)$	$(6, 12, 1)$	$(12, \infty, 6)$	$(14, 3, 5)$	$(19, 1, 2)$	$(21, 16, 9)$	$(25, 20, d)$	$(27, 4, 6)$
$(28, 13, 19)$	$(29, 10, 17)$	$(30, 18, d)$	$(32, 6, 5)$	$(34, 11, 11)$	$(35, 14, 23)$	$(36, 17, d)$	$(42, 5, 1)$
$(43, 2, 12)$	$(44, 19, d)$	$(45, 9, 29)$	$(46, \infty, 4)$	$(46, 7, 4)$	$(12, 8, 28)$	$(14, \infty, 3)$	$(23, \infty, 19)$

Figure B.14: Simulate $(write, 8, 28)$

count is incremented to 3 and we simulate $(write, 2, 15)$ beginning with the scan of the shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Since we did not find program address 2 in the shelter, we perform a binary search across the non-shelter locations looking for the tag $f(2) = 43$:

$$((read, 10, \perp), (read, 15, \perp), (read, 17, \perp))$$

Now that we have found program address 2 at physical word 17, we update the value at that physical location:

$$(write, 17, (43, \infty, 12))$$

Next we scan the shelter. Since program address 2 was in a non-shelter location, we will put it into the first empty shelter location we find (Figure B.15):

$$((read, 21, \perp), (write, 21, (46, 7, 4)), (read, 22, \perp), (write, 22, (12, 8, 28)),$$
$$(read, 23, \perp), (write, 23, (43, 2, 15)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

$(5, 15, 11)$	$(6, 12, 1)$	$(12, \infty, 6)$	$(14, 3, 5)$	$(19, 1, 2)$	$(21, 16, 9)$	$(25, 20, d)$	$(27, 4, 6)$
$(28, 13, 19)$	$(29, 10, 17)$	$(30, 18, d)$	$(32, 6, 5)$	$(34, 11, 11)$	$(35, 14, 23)$	$(36, 17, d)$	$(42, 5, 1)$
$(43, \infty, 12)$	$(44, 19, d)$	$(45, 9, 29)$	$(46, \infty, 4)$	$(46, 7, 4)$	$(12, 8, 28)$	$(43, 2, 15)$	$(23, \infty, 19)$

Figure B.15: Simulate $(write, 2, 15)$

We increment *count* to four and begin the last access we will simulate in this pass: $(write, 7, 14)$. Once again we start by scanning the shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

This time we find program address 7 in the shelter at physical word 21. Since we found it in the shelter, a dummy value will be accessed. Because *count* is four, we

will be looking for dummy address 20 (we have 16 program words which tells us we should look for address $16 + 4 = 20$). We will perform a binary search across the non-shelter locations looking for the word tagged with $f(20) = 25$:

$$((read, 10, \perp), (read, 5, \perp), (read, 7, \perp))$$

We find the dummy word at physical word 7 on the tape and write the same value back to the tape:

$$(write, 7, (25, d, d))$$

Next we scan the shelter and update the value for program word 7 when we find it in the shelter (Figure B.16):

$$((read, 21, \perp), (write, 21, (46, 7, 14)), (read, 22, \perp), (write, 22, (12, 8, 28)),$$
$$(read, 23, \perp), (write, 23, (43, 2, 15)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

$(5, 15, 11)$	$(6, 12, 1)$	$(12, \infty, 6)$	$(14, 3, 5)$	$(19, 1, 2)$	$(21, 16, 9)$	$(25, 20, d)$	$(27, 4, 6)$
$(28, 13, 19)$	$(29, 10, 17)$	$(30, 18, d)$	$(32, 6, 5)$	$(34, 11, 11)$	$(35, 14, 23)$	$(36, 17, d)$	$(42, 5, 1)$
$(43, \infty, 12)$	$(44, 19, d)$	$(45, 9, 29)$	$(46, \infty, 4)$	$(46, 7, 14)$	$(12, 8, 28)$	$(43, 2, 15)$	$(23, \infty, 19)$

Figure B.16: Simulate $(write, 7, 14)$

Now that we have completed simulation of the four accesses for this pass, we need to return all of the words to their original location on the tape. This is done by a Batcher sort across all of the words on the work tape using the program address of each word, which generates the following sequence of comparisons: **Order**$(1, 2)$, **Order**$(1, 2)$, **Order**$(4, 5)$, **Order**$(4, 5)$, **Order**$(1, 5)$, **Order**$(3, 7)$, **Order**$(3, 5)$, **Order**$(2, 4)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(7, 8)$, **Order**$(7, 8)$, **Order**$(10, 11)$, **Order**$(10, 11)$, **Order**$(7, 11)$, **Order**$(9, 13)$, **Order**$(9, 11)$, **Order**$(8, 10)$,

Order$(8, 9)$, **Order**$(10, 11)$, **Order**$(1, 9)$, **Order**$(5, 13)$, **Order**$(5, 9)$, **Order**$(3, 11)$, **Order**$(7, 15)$, **Order**$(7, 11)$, **Order**$(3, 5)$, **Order**$(7, 9)$, **Order**$(2, 10)$, **Order**$(6, 14)$, **Order**$(6, 10)$, **Order**$(4, 8)$, **Order**$(4, 6)$, **Order**$(8, 10)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(6, 7)$, **Order**$(8, 9)$, **Order**$(10, 11)$, **Order**$(13, 14)$, **Order**$(13, 14)$, **Order**$(16, 17)$, **Order**$(16, 17)$, **Order**$(13, 17)$, **Order**$(15, 19)$, **Order**$(15, 17)$, **Order**$(14, 16)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(19, 20)$, **Order**$(19, 20)$, **Order**$(22, 23)$, **Order**$(22, 23)$, **Order**$(19, 23)$, **Order**$(21, 25)$, **Order**$(21, 23)$, **Order**$(20, 22)$, **Order**$(20, 21)$, **Order**$(22, 23)$, **Order**$(13, 21)$, **Order**$(17, 25)$, **Order**$(17, 21)$, **Order**$(15, 23)$, **Order**$(19, 27)$, **Order**$(19, 23)$, **Order**$(15, 17)$, **Order**$(19, 21)$, **Order**$(14, 22)$, **Order**$(18, 26)$, **Order**$(18, 22)$, **Order**$(16, 20)$, **Order**$(16, 18)$, **Order**$(20, 22)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(18, 19)$, **Order**$(20, 21)$, **Order**$(22, 23)$, **Order**$(1, 17)$, **Order**$(9, 25)$, **Order**$(9, 17)$, **Order**$(5, 21)$, **Order**$(13, 29)$, **Order**$(13, 21)$, **Order**$(5, 9)$, **Order**$(13, 17)$, **Order**$(3, 19)$, **Order**$(11, 27)$, **Order**$(11, 19)$, **Order**$(7, 23)$, **Order**$(15, 31)$, **Order**$(15, 23)$, **Order**$(7, 11)$, **Order**$(15, 19)$, **Order**$(3, 5)$, **Order**$(7, 9)$, **Order**$(11, 13)$, **Order**$(15, 17)$, **Order**$(19, 21)$, **Order**$(2, 18)$, **Order**$(10, 26)$, **Order**$(10, 18)$, **Order**$(6, 22)$, **Order**$(14, 30)$, **Order**$(14, 22)$, **Order**$(6, 10)$, **Order**$(14, 18)$, **Order**$(4, 20)$, **Order**$(12, 28)$, **Order**$(12, 20)$, **Order**$(8, 16)$, **Order**$(8, 12)$, **Order**$(16, 20)$, **Order**$(4, 6)$, **Order**$(8, 10)$, **Order**$(12, 14)$, **Order**$(16, 18)$, **Order**$(20, 22)$, **Order**$(2, 3)$, **Order**$(4, 5)$, **Order**$(6, 7)$, **Order**$(8, 9)$, **Order**$(10, 11)$, **Order**$(12, 13)$, **Order**$(14, 15)$, **Order**$(16, 17)$, **Order**$(18, 19)$, **Order**$(20, 21)$, **Order**$(22, 23)$.

Each comparison generates a read of each of the two locations and then a write to both locations. After the Batcher sort, the work tape will be as depicted in Figure B.17.

Now we are ready to begin the second pass and simulate the next four accesses:

$$((read, 9, \perp), (read, 11, \perp), (write, 8, 22), (read, 4, \perp))$$

$(19, 1, 2)$	$(43, 2, 15)$	$(14, 3, 5)$	$(27, 4, 6)$	$(42, 5, 1)$	$(32, 6, 5)$	$(46, 7, 14)$	$(12, 8, 28)$
$(45, 9, 29)$	$(29, 10, 17)$	$(34, 11, 11)$	$(6, 12, 1)$	$(28, 13, 19)$	$(35, 14, 23)$	$(5, 15, 11)$	$(21, 16, 9)$
$(36, 17, d)$	$(30, 18, d)$	$(44, 19, d)$	$(25, 20, d)$	$(12, \infty, 6)$	$(43, \infty, 12)$	$(46, \infty, 4)$	$(23, \infty, 19)$

Figure B.17: Batcher Sort Across All Words

To begin with, we need to construct a new $f : [20] \to \mathbb{Z}_{49}$ using the random oracle. Suppose the f we construct for this pass maps the values as shown in Table B.2.

i	$f(i)$
1	39
2	26
3	36
4	30
5	16
6	6
7	48
8	10
9	34
10	7
11	9
12	43
13	33
14	28
15	25
16	45
17	35
18	1
19	19
20	15

Table B.2: $f : [20] \to \mathbb{Z}_{49}$ for the Second Pass

Once again we begin by updating the tags on all of the non-shelter locations, as seen in Figure B.18.

Next we perform a Batcher sort across the non-shelter locations, sorting by tags. As we already know, the Batcher sort is deterministic (Section 2.5) so the comparisons will be the same as in the first pass. After the Batcher sort, the work

$(39, 1, 2)$	$(26, 2, 15)$	$(36, 3, 5)$	$(30, 4, 6)$	$(16, 5, 1)$	$(6, 6, 5)$	$(48, 7, 14)$	$(10, 8, 28)$
$(34, 9, 29)$	$(7, 10, 17)$	$(9, 11, 11)$	$(43, 12, 1)$	$(33, 13, 19)$	$(28, 14, 23)$	$(25, 15, 11)$	$(45, 16, 9)$
$(35, 17, d)$	$(1, 18, d)$	$(19, 19, d)$	$(15, 20, d)$	$(12, \infty, 6)$	$(43, \infty, 12)$	$(46, \infty, 4)$	$(23, \infty, 19)$

Figure B.18: Update the Tags on all Non-Shelter Locations

tape will be as shown in Figure B.19.

$(1, 18, d)$	$(6, 6, 5)$	$(7, 10, 17)$	$(9, 11, 11)$	$(10, 8, 28)$	$(15, 20, d)$	$(16, 5, 1)$	$(19, 19, d)$
$(25, 15, 11)$	$(26, 2, 15)$	$(28, 14, 23)$	$(30, 4, 6)$	$(33, 13, 19)$	$(34, 9, 29)$	$(35, 17, d)$	$(36, 3, 5)$
$(39, 1, 2)$	$(43, 12, 1)$	$(45, 16, 9)$	$(48, 7, 14)$	$(12, \infty, 6)$	$(43, \infty, 12)$	$(46, \infty, 4)$	$(23, \infty, 19)$

Figure B.19: Batcher Sort by Tag Across Non-Shelter Locations

We initialize *count* to 1 and begin to simulate $(read, 9, \perp)$ by scanning the entire shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Since nothing has been put in the shelter yet for this pass, we do not find program word 9 in the shelter, so we begin a binary search for it outside of the shelter looking for tag $f(9) = 34$:

$$((read, 10, \perp), (read, 15, \perp), (read, 12, \perp), (read, 13, \perp), (read, 14, \perp))$$

Finding the word at physical address 14, we update the value at that location to be $(34, \infty, 29)$ and rescan the shelter, putting program word 9 into the first empty

shelter location (Figure B.20):

$$((read, 21, \perp), (write, 21, (34, 9, 29)), (read, 22, \perp), (write, 22, (43, \infty, 12)),$$
$$(read, 23, \perp), (write, 23, (46, \infty, 4)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

$(1, 18, d)$	$(6, 6, 5)$	$(7, 10, 17)$	$(9, 11, 11)$	$(10, 8, 28)$	$(15, 20, d)$	$(16, 5, 1)$	$(19, 19, d)$
$(25, 15, 11)$	$(26, 2, 15)$	$(28, 14, 23)$	$(30, 4, 6)$	$(33, 13, 19)$	$(34, \infty, 29)$	$(35, 17, d)$	$(36, 3, 5)$
$(39, 1, 2)$	$(43, 12, 1)$	$(45, 16, 9)$	$(48, 7, 14)$	$(34, 9, 29)$	$(43, \infty, 12)$	$(46, \infty, 4)$	$(23, \infty, 19)$

Figure B.20: Simulate $(read, 9, \perp)$

We increment *count* to two and simulate our next access, $(read, 11, \perp)$, by scanning the shelter again:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Not finding the word in the shelter, we perform a binary search across the non-shelter locations using the tag $f(11) = 9$:

$$((read, 10, \perp), (read, 5, \perp), (read, 3, \perp), (read, 4, \perp))$$

We update the value at physical word 4 (where we found program word 11) to be $(9, \infty, 11)$. Next we rescan the shelter and insert the word into the first empty location (Figure B.21):

$$((read, 21, \perp), (write, 21, (34, 9, 29)), (read, 22, \perp), (write, 22, (9, 11, 11)),$$
$$(read, 23, \perp), (write, 23, (46, \infty, 4)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

count is increased to three. The third access we are simulating on this pass is

$(1, 18, d)$	$(6, 6, 5)$	$(7, 10, 17)$	$(9, \infty, 11)$	$(10, 8, 28)$	$(15, 20, d)$	$(16, 5, 1)$	$(19, 19, d)$
$(25, 15, 11)$	$(26, 2, 15)$	$(28, 14, 23)$	$(30, 4, 6)$	$(33, 13, 19)$	$(34, \infty, 29)$	$(35, 17, d)$	$(36, 3, 5)$
$(39, 1, 2)$	$(43, 12, 1)$	$(45, 16, 9)$	$(48, 7, 14)$	$(34, 9, 29)$	$(9, 11, 11)$	$(46, \infty, 4)$	$(23, \infty, 19)$

Figure B.21: Simulate $(read, 11, \perp)$

$(write, 8, 22)$. As usual, we start by scanning the shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Again, we do not find the word in the shelter so we perform a binary search across the rest of the work tape, looking for the tag $f(8) = 10$:

$$((read, 10, \perp), (read, 5, \perp))$$

When we find the word at physical address 5, we update the value there to be $(10, \infty, 28)$. Next, we rescan and update the shelter, putting program address 8 into the first empty shelter location we find (Figure B.22):

$$((read, 21, \perp), (write, 21, (34, 9, 29)), (read, 22, \perp), (write, 22, (9, 11, 11)),$$
$$(read, 23, \perp), (write, 23, (10, 8, 22)), (read, 24, \perp), (write, 24, (23, \infty, 19)))$$

count is incremented to four and we simulate the last access for this pass: $(read, 4, \perp)$. We begin by scanning the entire shelter:

$$((read, 21, \perp), (read, 22, \perp), (read, 23, \perp), (read, 24, \perp))$$

Program word 4 is not found in the shelter so we perform a binary search across

$(1, 18, d)$	$(6, 6, 5)$	$(7, 10, 17)$	$(9, \infty, 11)$	$(10, \infty, 28)$	$(15, 20, d)$	$(16, 5, 1)$	$(19, 19, d)$
$(25, 15, 11)$	$(26, 2, 15)$	$(28, 14, 23)$	$(30, 4, 6)$	$(33, 13, 19)$	$(34, \infty, 29)$	$(35, 17, d)$	$(36, 3, 5)$
$(39, 1, 2)$	$(43, 12, 1)$	$(45, 16, 9)$	$(48, 7, 14)$	$(34, 9, 29)$	$(9, 11, 11)$	$(10, 8, 22)$	$(23, \infty, 19)$

Figure B.22: Simulate $(write, 8, 22)$

the non-shelter locations looking for tag $f(4) = 30$:

$$((read, 10, \perp), (read, 15, \perp), (read, 12, \perp))$$

The word at physical address 12 is updated to be $(30, \infty, 6)$ and we scan the shelter, putting the program word into the last empty shelter location (Figure B.23):

$$((read, 21, \perp), (write, 21, (34, 9, 29)), (read, 22, \perp), (write, 22, (9, 11, 11)),$$
$$(read, 23, \perp), (write, 23, (10, 8, 22)), (read, 24, \perp), (write, 24, (30, 4, 6)))$$

$(1, 18, d)$	$(6, 6, 5)$	$(7, 10, 17)$	$(9, \infty, 11)$	$(10, \infty, 28)$	$(15, 20, d)$	$(16, 5, 1)$	$(19, 19, d)$
$(25, 15, 11)$	$(26, 2, 15)$	$(28, 14, 23)$	$(30, \infty, 6)$	$(33, 13, 19)$	$(34, \infty, 29)$	$(35, 17, d)$	$(36, 3, 5)$
$(39, 1, 2)$	$(43, 12, 1)$	$(45, 16, 9)$	$(48, 7, 14)$	$(34, 9, 29)$	$(9, 11, 11)$	$(10, 8, 22)$	$(30, 4, 6)$

Figure B.23: Simulate $(read, 4, \perp)$

Now that we have finished simulating the next four accesses, we need to perform a Batcher sort across the entire work tape by program address to return all of the values to their original location. The comparisons that are made in this pass are the same as in the previous pass for this particular step. After the sort is complete, the work tape will be as in Figure B.24.

This completes the second pass of our simulation and all of the accesses we are

$(39, 1, 2)$	$(26, 2, 15)$	$(36, 3, 5)$	$(30, 4, 6)$	$(16, 5, 1)$	$(6, 6, 5)$	$(48, 7, 14)$	$(10, 8, 22)$
$(34, 9, 29)$	$(7, 10, 17)$	$(9, 11, 11)$	$(43, 12, 1)$	$(33, 13, 19)$	$(28, 14, 23)$	$(25, 15, 11)$	$(45, 16, 9)$
$(35, 17, d)$	$(1, 18, d)$	$(19, 19, d)$	$(15, 20, d)$	$(9, \infty, 11)$	$(10, \infty, 28)$	$(30, \infty, 6)$	$(34, \infty, 29)$

Figure B.24: Batcher Sort by Address Across All Words

simulating in this demonstration. If the original RAM continued execution, the remaining passes would continue from this point.