UNIVERSITY OF PIRAEUS
DEPARTMENT OF DIGITAL SYSTEMS

DIGITAL SYSTEMS SECURITY DIRECTION

Oblivious RAM from Theory to Practice

Master Thesis

By Tsaktsiras Dimitris

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in the Department of Digital Systems at University of Piraeus

PIRAEUS, MARCH 2017

Abstract

Outsourcing storage/computation has been gaining popularity because its elasticity. However, this new type of storage model also brings security concerns. Privacy of data storage has long been a central problem in computer security. The most common technique to protect our data is to encrypt it. However, encryption does not prevent information disclosure about where we read or write in our data. This additional information, the access pattern, can be used to reverse-engineer proprietary programs as they run, reveal a user's physical location or health information, and more, even if data is correctly encrypted.

A cryptographic primitive, which provably hides a client' access pattern as seen by untrusted storage, called Oblivious RAM (ORAM). ORAM was introduced by Goldreich and Ostrovsky [1], where in the key motivation was stated as software protection from an adversary who can observe the memory access pattern (but not the contents of the memory). ORAM incurs a large performance overhead and can require a large amount of client, who is considered trusted, storage. In particular, ORAM schemes require the client to continuously shuffle the data stored in the untrusted storage, using the trusted storage. Early work on ORAM proves that this operation must incur a client-storage bandwidth blowup that is logarithmic in the dataset size, which can translate to > 100× in practice. This thesis studies several ORAM schemes that could make feasible the use of ORAM in practice by reducing performance overhead and in some cases client storage.

We address this challenge by presenting ORAM schemes that make both theoretical and practical contributions. Those schemes are categorized based on their characteristics. The 4 main categories are Path ORAM, Constant worst-case bandwidth blowup, ObliviStore, and Applied ORAM. In the Path ORAM family we present the schemes Path ORAM [2], Path Oblivious RAM in Secure Processors [7], Circuit ORAM [8], Bucket ORAM [44] and Ring ORAM [11] and a comparison between them. The Constant worst-case bandwidth blowup ORAM family includes Onion ORAM [14] and C – ORAM [22] and we present the respective comparison. In Chapter 5 we present and compare ObliviStore ORAM [5], Burst ORAM [13] and CURIOUS ORAM [35], which are included in the ObliviStore ORAM Family. Finally, we present two Applied ORAM schemes ObliviSync [50] and Tiny ORAM [33]. ObliviSync is an oblivious cloud storage system that specifically targets one of the most widely-used personal cloud storage paradigms. Tiny ORAM is a hardware ORAM with small client storage, integrity verification, or encryption units.

Acknowledgments

I would like to thank my advisor Konstantinos Lambrinoudakis, and Panagiotis Rizomiliotis (my de facto co-advisor) for help, advice, inspiration and guidance.

Furthermore, I want to thank Marianna and my parents for their patience, understanding and support during my postgraduate studies and the development of my master thesis.

July 2017

Tsaktsiras Dimitris

Table of Contents

Content of Figures and Tables

1 Introduction

Security of data storage is a huge problem in nearly all aspects of the Internet connected world. Consider several ubiquitous settings: outsourced storage, computation outsourcing and the Internet of Things (IoT).

In outsourced storage, users outsource private data storage from their private infrastructure to remote cloud servers. Data can now be stolen at any point in the cloud infrastructure; for instance, at the server itself (e.g., by insiders [56]), at the internet-server boundary (based on the Snowden revelations [57]) or in transit (e.g., [58]).

Further, in computation outsourcing and IoT, sensitive information is stored on cloud servers, or other potentially hostile environments, as it is being computed upon. Despite the promise of tamper-resistant systems (e.g., [59]) and bootstrapping trust from a known CPU state (e.g., [60]), which protect data while it resides on-chip (or on-package), data can still be stolen via software or physical attacks when it is stored on-chip (e.g., in main memory or disk). For instance, it has been shown how memory can be accessed by exploiting cloud resource sharing [61] and vulnerable firmware [39, 41]. In IoT, the attacker may have physical access to devices, which it can use to extract data using (for example) test cards [37], bus probing [38] or technology-specific techniques [36].

A natural starting point to address this issue is to encrypt all data written to untrusted storage. For example, consider client-side encryption which defines two parties: a trusted client and untrusted server (storage). When data passes to/from the server, it is encrypted/ decrypted by the client. Only the client holds the secret key. Thus, the server cannot decrypt the data it stores unless it is able to break the encryption scheme. Client-side encryption is used today. For example, it is implemented at the chip boundary in remote processors to protect main memory (e.g., Intel SGX [62]) and at the client boundary to protect outsourced storage applications (e.g., [63]).

A big problem with client-side encryption (and other systems that protect only the data itself) is that it does not protect all aspects of how the client interacts with the server's storage. Where storage is accessed, the access pattern, can also reveal secret information. For example, consider the following:

- Suppose a patient stores his/her genome on a remote server and wishes to check if he/she has an allele/SNP (i.e., which is located at a specific point on the genome) which corresponds to cancer. If an observer (e.g., an insurance company) learns where that patient is looking in its genome, the observer can infer that the patient was concerned about cancer. Similar examples can be

drawn from users requesting geo-location, financial and database queries over other sensitive information (e.g., [47, 64]).

- A common task in personal and cloud computing is to run a proprietary program on a remote processor. One of the open challenges with this deployment is to prevent software IP theft: the program distributor wants to avoid malicious parties from being able to reverse-engineer the program as it runs. Unfortunately, an observer capable of monitoring how a program accesses main memory can, in fact, reverse engineer the program's conditional and loop structure, simply by monitoring address requests to main memory [65, 66, 47].
- In the inverse of the software IP theft setting, a user may wish to outsource private data to a remote processor to compute some result. In this case, the program may be selected by the server hosting the processor (e.g., a cloud service which cannot be attested by the user) and is therefore untrustworthy [67, 68]. Untrusted programs running on sensitive data are a serious concern: the program may directly or inadvertently leak the user's data.

1.1 Challenges in Protecting Access Pattern

The underlying problem in the above examples is inherent in how programs are written today: to be performant, program control flow and memory access behavior depends on the sensitive information we wish to hide. Indeed, a strawman solution to eliminate all access pattern leakage is to perform the same amount of work, regardless of the program's sensitive inputs. In the worst case, this requires that the program scan all of memory on every access {e.g., download the entire genome to analyze a single allele {incurring huge performance overheads.

A natural question to ask is: can encryption solve this problem? Generally, the answer is no, considering practical constraints. Encrypting an address makes that address unusable by the memory unless the remote memory has the corresponding decryption key or the system is using certain cryptographic schemes. First, distributing decryption keys has serious limitations. In particular, the trust boundary now includes all of remote storage and the burden is on memory manufacturers to re-design their products to (safely) perform key exchange. Second, certain encryption schemes (e.g., private information retrieval [23] or homomorphic encryption [69]) can securely search over encrypted data. However, these schemes have an inherent problem: the scheme must compute over every element in the database. Otherwise, an observer trivially knows what elements were not selected. This has even worse overheads than scanning memory due to these schemes' computational complexities. To summarize, we desire address pattern protection that doesn't make assumptions on the untrusted storage, and is asymptotically more efficient than scanning memory.

Another question is: can we get away with incomplete protection? Incomplete access pattern protection is implicit in the state of the art hardware extensions from Intel, called Intel SGX [62, 41]. In that system, the access pattern may be called 'partially hidden' because the subset of memory accesses that cause page faults are directly revealed to the untrusted operating system. Recently, however, researchers showed how even this amount of leakage can be used to reconstruct the outline of medical images in medical applications [70]. Another example in this vein is the HIDE framework, by Zhuang et al. [65]. HIDE provides access pattern protection assuming constraints on the spatial locality in the program access pattern. But HIDE makes no guarantees for programs with arbitrary access patterns and, in particular, leaks non-negligible information for even a single access if there are no restrictions on where that access may occur. To summarize, we desire a general solution that doesn't make assumptions about the program access pattern, or about how much privacy is leaked on a particular memory access.

Another way to see the danger in the above attacks is to look at society's move from deterministic to randomized encryption schemes. In the University and Industry, students learn not to use deterministic encryption because it is "insecure", in particular it is subject to frequency analysis attacks. The access pattern can be viewed in a similar light: as client-side encryption becomes ubiquitous, frequency attacks on the remaining un-encrypted information (the access pattern) can become the new low-hanging fruit.

1.2 The Case for Oblivious RAM

To address the above problems, this thesis studies a cryptographic primitive called Oblivious RAM (ORAM), which provably eliminates all information leakage in memory access patterns [1, 3].

As with client-side encryption, an ORAM scheme is made up of a client and server with data blocks residing on the server. Consider two sequences of storage requests and A' made to the server, where each sequence is made up of read $(read, addr)$ and write $(write, addr, data)$ tuples. ORAM guarantees that from the server's perspective: if $|A| = |A'|$, then A is computationally indistinguishable from A'. Informally, this hides all information in A and A': whether the client is reading/writing to the storage, where the client is accessing, and the underlying data that the client is accessing.

ORAM addresses all the weaknesses discussed in the previous section. First, ORAM is asymptotically efficient: for a database of size N, modern ORAM schemes only need to download/re-upload $O(polylog\ N)$ data blocks from untrusted memory, per access (as opposed to the $O(N)$ cost of scanning memory). Second, ORAM makes no

assumptions on the external memory. Memory is considered untrusted, or actively malicious, and need not manage private keys. Finally, ORAM provides the same level of protection regardless of the access pattern and assumes all memory accesses are visible to the adversary.

Since its proposal by Goldreich and Ostrovsky [1, 3], ORAM has become an important part of the cryptographic "swiss army" knife, and has been proposed to secure numerous settings, both practical and theoretical. On the practice side, ORAM has been proposed to secure outsourced storage (e.g., [71]), hide secure processor behavior to external memory (e.g., [68, 47]) and implement searchable encryption with small leakage (e.g., [72]). Additionally on the cryptography side, ORAM has become an important building block in constructing efficient secure multi-party computation protocols (e.g., [73]), proofs of retrievability [74], and Garbled RAM [75].

Despite recent advancements and numerous potential applications, however, the primary impedance to ORAM adoption continues to be its practical efficiency. To achieve privacy as advertised, ORAM schemes require that the client continuously shuffle (i.e., physically re-locate) data as it is stored on the server. This shuffling has incurred $\Omega(\log N)$ bandwidth blowup between client and server in all ORAM proposals – which translates to 25× > 100× overhead in practice. In fact, the seminal work by Goldreich and Ostrovsky [1, 3] proved that the shuffling bandwidth must be at least logarithmic in N for an ORAM scheme to be secure.

To confound the problem, the shuffling requires a potentially large amount of trusted storage on the client side, and the most performant schemes require more storage. It is especially challenging to reduce bandwidth overhead while maintaining small client storage. This is obviously desirable: ORAM exists to securely outsource storage. Indeed, the most performant ORAM schemes (e.g., [13]) require GBytes (to tens of GBytes) of client storage to handle TByte-range ORAMs. This immediately rules out their applicability to settings where the client storage must be small; for example, if it must fit in the on-chip memory of a remote processor (which is the case with the software IP theft and computation outsourcing settings discussed above). On the other hand, the state of the art construction that can be deployed in a remote processor (i.e., requires only KBytes to MBytes of client storage) incurs > 8× the bandwidth overhead of the most performant schemes [2].

1.3 Thesis overview

We now give an overview of each chapter.

Chapter 2 – Preliminaries. We start by introducing security definitions for ORAM in several settings and efficiency metrics which will be studied later in the thesis. We then describe the usage settings for ORAM most related to the thesis and give a

history of prior work in ORAM starting with the first ORAM schemes by Goldreich and Ostrovsky.

Chapter 3 – Path ORAM Family. We present Path ORAM [2] and several ORAM schemes, which were based on Path ORAM and a comparison between them. The schemes that presented on this section are *a)* Optimization of Path Oblivious RAM in Secure Processors [7], *b)* Circuit ORAM [8], *c)* Bucket ORAM [44], and *d)* Ring ORAM [11].

Chapter 4 – Constant worst-case bandwidth blowup. We present ORAM schemes, which achieve Constant worst-case Bandwidth blowup and a comparison between them. The ORAM schemes are Onion ORAM [14] and C – ORAM [22].

Chapter 5 – ObliviStore ORAM Family. We present ObliviStore ORAM [5] and ORAM schemes, which were based on ObliviStore ORAM and a comparison between them. The schemes that presented on this section are a) Burst ORAM [13], and b) CURIOUS ORAM [35].

Chapter 6 – Applied ORAM Schemes. In this section, we present two applied ORAM schemes (ObliviSync [50] and Tiny ORAM [33]) that could be used in *real world*. ObliviSync is an oblivious cloud storage system that specifically targets one of the most widely-used personal cloud storage paradigms: synchronization and backup services, popular examples of which are Dropbox, iCloud Drive, and Google Drive. Tiny ORAM is a hardware ORAM with small client storage, integrity verification, or encryption units.

Chapter 7 – Conclusion. We summarize the results of the thesis.

2 Preliminaries

In this chapter, we give formal definitions for ORAM. We first give a strong and general security definition, which achieves simulator-based security against a malicious adversary. We then discuss ORAM metrics and how they impact practice, and give a history of ORAM schemes.

2.1 Problem Definition

We consider a client that wishes to store data at a remote untrusted server while preserving its privacy. While traditional encryption schemes can provide data confidentiality, they do not hide the data access pattern which can reveal very sensitive information to the untrusted server. In other words, the blocks accessed on the server and the order in which they were accessed is revealed. We assume that the server is untrusted, and the client is trusted, including the client's processor, memory, and disk. The goal of ORAM is to completely hide the data access pattern (which blocks were read/written) from the server. From the server's perspective, the data access patterns from two sequences of read/write operations with the same length must be indistinguishable.

2.2 ORAM Definition

Following Apon *et al.* **[28]**, we define ORAM as a reactive two-party protocol between the client and the server, and define its security in the Universal Composability model **[29]**. We use the notation

$$((c\_out, c\_state), (s\_out, s\_state)) \leftarrow \text{protocol}((c\_in, c\_state), (s\_in, s\_state))$$

to denote a (stateful) protocol between a client and server, where *c_in* and *c_out* are the client's input and output; *s_in* and *s_out* are the server's input and output; and *c_state* and *s_state* are the client and server's status before and after the protocol.

Definition 1 An ORAM scheme consists of the following interactive protocols between a client and a server.

$((\perp,\mathcal{C}),(\perp,\mathcal{D})) \leftarrow \text{Setup}(1^{\lambda}, (D,\perp),(\perp,\perp))$: An interactive protocol where the client's input is a memory array $D[1..N]$ where each memory block has bit-length B; and the server's input is $\perp$. At the end of the Setup protocol, the client has secret state $\mathcal{C}$, and the server's state is $\mathcal{D}$ (which typically encodes the memory array D).

$((data,\mathcal{C}'),(\perp,\mathcal{D}')) \leftarrow \text{Access}((op,\mathcal{C}),(\perp,\mathcal{D}))$: To access data, the client starts in state $\mathcal{C}$, with an input op where op := (read, *addr*) or op := (write, *addr, data*); the server starts

in state $\mathcal{D}$, and has no input. In a correct execution of the protocol, the client's output *data* is the current value of the memory $\mathcal{D}$ at location *addr* (for writes, the output is the old value of *D*[*addr*] before the write takes place). The client and the server also update their state to $\mathcal{C}'$ and $\mathcal{D}'$ respectively. The client outputs *data* := ⊥ if the protocol execution aborted.

We say that the ORAM scheme is correct, if for any initial memory $D \in \{0, 1\}^{BN}$, for any operation sequence $op_1, op_2, \ldots, op_m$ where m = poly(λ), an op := (read, *addr*) operation would always return the last value written to the logical location *addr* (except with negligible probability).

2.2.1 Tree-based ORAM Framework

Shi *et al.* **[4]** proposed a new tree-based framework, which was adopted subsequently by several improved constructions **[9, 12, 30, 2, 31]**. We now briefly review the framework.

Notation. We use N to denote the number of (real) data blocks in ORAM, B to denote the bit-length of a block in ORAM, Z to denote the capacity of each bucket in the ORAM tree, and λ to denote the ORAM's statistical security parameter. For convenience in algorithm descriptions, we sometimes treat the stash as a depth-0 bucket with some capacity R that is the imaginary parent of the root. We assume that leaves are numbered sequentially from 0 to $N-1$. We also denote [a…b] := {a, a+1, …, b}.

```
Access(op) // wher op = ("read", idx) or op = ("write", idx, data*)
1: label := PositionMap[idx]
2: {idx||label||data} := ReadAndRm(idx, label)
3: PositionMap [idx] := UniformRandom(0…N – 1)
4: If op is "read" : data* := data
5: stash.add({idx||PositionMap[idx]||data*})
6:Evict()
7:Return darax ← position[a]
```

Figure 1: Generic Access Algorithm

Data structure. The server organizes blocks into a binary tree of height $L = \log N + 1$; each node of the tree is a bucket containing Z blocks. Each block is of the form:

$$\{\text{idx}||\text{label}||\text{data}\},$$

where idx is the index of the block, e.g. the (logical) address of desired block; label is a leaf identifier specifying the path on which the block resides; and data is the payload of the block, of B bits in size.

The client stores a stash for buffering overflowing blocks. In certain schemes such as the original binary-tree scheme **[4]**, a stash is not necessary. In this case, we can simply treat this as a degenerate stash of size 0.

The client also stores a position map, mapping a block's idx to a leaf label. As described later, position map storage can be reduced to O(1) by recursively storing the position map in a smaller ORAM. These leaf labels are assigned randomly and are reassigned as blocks are accessed. If we label the leaves from 0 to N - 1 then each label is associated with a path from the root to the corresponding leaf.

Main path invariant. Three-based ORAMs maintain the invariant that a block marked label resides on the path from the stash (to the root) to the lead node marked label

Operations. Tree-based ORAMs all follow a similar recipe as shown in Figure 1. In particular, the ReadAndRm operation would read every block on the path leading to the leaf node marked label, and fetches and removes the block idx from the path.

Various tree-based ORAMs are differentiated by the eviction algorithm denoted Evict(). For example, the original binay-tree ORAM adopts a simple eviction algorithm engineered to make their proof easy: with each data access, two distinct buckets are chosen at random from each level to evict from. By contrast, the Path ORAM algorithm performs eviction on the read path, and the eviction strategy is aggressive: pack all blocks as close to the leaf as possible respecting the main invariant. In Path ORAM, a $O(\log N) \cdot \omega(1)$ stash is necessary to buffer overflowing blocks.

Recursion. Instead of storing the entire position map in the client's local memory, the client can store it in a smaller ORAM on the server. In particular, this position map ORAM needs to store N labels each of $\log N$ bits. We can apply this idea recursively until we get down to a constant amount of metadata, which the client could store locally.

2.2.2 Security Definition

We will adopt two security definitions throughout this thesis. The first follows a standard simulation-based definition of secure computation **[32]**, requiring that a real-world execution "simulate" an ideal-world (reactive) functionality $\mathcal{F}$.

Ideal world. We define an ideal functionality $\mathcal{F}$ that maintains an up-to-date version of the data *D* on behalf of the client, and answers the client's access queries.

- **Setup.** An environment $\mathcal{Z}$ gives an initial database *D* to the client. The client sends *D* to an ideal functionality $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ of the fact that the setup operation occurred as well as the size of the database $N = |D|$, but

not of the data contents *D*. The ideal-world adversary $\mathcal{S}$ says ok or abort to $\mathcal{F}$. $\mathcal{F}$ then says ok or ⊥ to the client accordingly.

- **Access.** In each time step, the environment $\mathcal{Z}$ specifies an operation op := (read, addr) or op := (write, addr, data) as the client's input. The client sends op to $\mathcal{F}$. $\mathcal{F}$ notifies the ideal-world adversary $\mathcal{S}$ (without revealing to $\mathcal{S}$ the operation op). If $\mathcal{S}$ says ok to $\mathcal{F}$, $\mathcal{F}$ sends D[addr] to the client, and updates D[addr] := data accordingly if this is a write operation. The client then forwards D[addr] to the environment $\mathcal{Z}$. If $\mathcal{S}$ says abort to $\mathcal{F}$, $\mathcal{F}$ sends ⊥ to the client

Real world. In the real world, an environment $\mathcal{Z}$ gives an honest client a database D. The honest client runs the Setup protocol with the server $\mathcal{A}$. Then, at each time step, $\mathcal{Z}$ specifies an input op := (read, addr) or op := (write, addr, data) to the client. The client runs the Access protocol with the server. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment the data fetched or ⊥ (indicating abort).

Definition 2 (Simulation-based security: privacy + verifiability). We say that a protocol $\Pi_{\mathcal{Z}}$ securely computes the ideal functionality $\mathcal{F}$ if for all probabilistic polynomial-time real-world adversaries (i.e. server) $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$, such that for all non-uniform, polynomial-time environments $\mathcal{Z}$, there exists a negligible function negl such that

$$|\Pr[REAL_{\Pi_{\mathcal{Z}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[IDEAL_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda) = 1]| \leq \text{negl}(\lambda)$$

At an intuitive level, our definition captures the privacy and verifiability requirements for an honest client (the client is never malicious in our setting), in the presence of a malicious server. The definition simultaneously captures privacy and verifiability. Privacy ensures that the server cannot observe the data contents or the access pattern (the contents of any op_i). Verifiability ensures that the client is guaranteed to read the correct data from the server.

2.2.3 Termination Channel Leakage

The two outcomes from running the protocol in the previous section are: (1) the adversary deviates from the protocol, which may cause the client to prematurely abort, or (2) the adversary lets the protocol complete. In (1), which we call the termination channel, there is no privacy leakage since we require the existence of $\mathcal{S}$ that, for all $\mathcal{Z}$, can predict when the termination occurs a-priori (i.e. independent of the access pattern given by $\mathcal{Z}$). In (2), there is some leakage: the adversary learns how many accesses were requested by $\mathcal{Z}$. However we are not interested in preventing

this leakage since it once again does not depend on the operations (the access pattern) submitted by $\mathcal{Z}$.

In this section. We want to provide a more relaxed definition, which permits some small, but access pattern-related, leakage through the termination channel. Doing so will enable several performance optimizations on Tiny ORAM scheme (hardware ORAM) **[33]**. Informally, the change to the real world adversary's view is the following. If the adversary is semi-honest, the protocol terminates when $\mathcal{Z}$ stops submitting operations and the adversary only learns (2) above, as with the previous definition. If the adversary is malicious, the protocol may terminate before (2) occurs, but when it terminates will be a function of the adversary's strategy, randomness in the protocol, **and importantly the access pattern specified by $\mathcal{Z}$**. For the purposes of satisfying the definition, we wish to show that this termination channel leakage is the only new information the adversary is able to learn. More formally,

Definition 3 (Termination channel security: privacy). An ORAM scheme is secure by the termination channel definition if for every (malicious) adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}'$ such that the following two distributions are computationally indistinguishable.

1. **(Real world).** $\mathcal{A}$ choose D. The experiment runs $((\perp,\mathcal{C}_0), (\perp, ,\mathcal{D}_0)) \leftarrow \text{Setup}(1^\lambda, (D, \perp), (\perp,\perp))$. $\mathcal{A}$ then adaptively makes read/write queries through $\mathcal{Z}$, which runs the protocol (for i=0, 1, …)

$$((data_{i+1},\mathcal{C}_{i+1}), (\perp, \mathcal{D}_{i+1})) \leftarrow \text{Access}((\text{op}_i, \mathcal{C}_i), (\perp,\mathcal{D}_i))$$

Denote the transcript of events visible to $\mathcal{A}$ during this call to Access as t_i. After the i-th call to Access, there are one of two outcomes. First, the client aborts with $\perp$ and the protocol terminates. Second, the client makes the (i + 1)-th call to Access, during which $\mathcal{A}$ may adaptively change its strategy based on its current view, namely $\{D_0,t_0, \ldots, t_i\}$. (Implicit in the adversary's view is that the client did not abort during the first i calls.)

Repeat this procedure for m' calls to Access, which denotes the point when the client aborts or $\mathcal{Z}$ stops submitting operations (whichever comes first). The distribution output by the experiment is then $\{D_0,t_0, \ldots, t_{m'}\}$ or $\{D_0,t_0, \ldots, t_{m'}, \perp\}$ depending on whether an abort occurred.

2. **(Ideal world).** Repeat the experiment in the real world with $\mathcal{S}'$, except for the following: Replace the transcript of events visible to $\mathcal{S}'$ during each call to Access with the symbol T (if the client does not abort) and $\perp$ (otherwise). Suppose the experiment terminates or concludes after m'' calls to Access. The distribution

output by the experiment is $\{D_0, \{T^{m''}\}\}$ or $\{D_0, T^{m''-1}, \perp\}$ depending on whether an abort occurred.

Definition 4 (Termination channel security: verifiability/integrity). Consider the following correctness experiment. The client and $\mathcal{A}$ run m' rounds of the Access protocol, at which point the protocol naturally terminates or prematurely aborts as described above. Correctness requires that except with negligible probability:

1. If the client aborted: $op_1, op_2, \ldots, op_{m'-1}$ are correct.
2. Otherwise (if the protocol naturally concluded): $op_1, op_2, \ldots, op_{m'}$ are correct.

Correctness of each trace follows the definition from Section 2.2 and is from the perspective of $\mathcal{Z}$.

2.3 Metrics

We will gauge ORAM schemes primarily on the following performance metrics. Note regarding notation: Metrics are in bits unless otherwise specified.

Client/server storage. The client/server's storage, given by $|\mathcal{C}|$ and $|\mathcal{D}|$ in the above definitions, refers to the number of blocks held by the client and server at the start of an Access operation. In all the schemes we describe, the client/server storage after and during each call to Access will be the same asymptotically as the starting size, so we will not distinguish these cases. Following conventions from related work, we say client storage is small if it is $O(B\ polylog\ N)$ and large if it is $\Omega(B\sqrt{N})$ – where B is the data block size in bits. We consider an insecure block storage system to require $O(BN)$ server storage and O(B) client storage, thus this is optimal for an ORAM as well.

Bandwidth cost and bandwidth blowup. An ORAM's bandwidth cost refers to the average number of bits transferred for accessing each block of B bits. An ORAM's bandwidth blowup is defined as its bandwidth cost divided by B (i.e., the bit-length of a data block). Effectively, the bandwidth blowup means the multiplicative factor in bandwidth one needs to pay to get obliviousness.

Client-server bandwidth. Client-server bandwidth (bandwidth for short) refers to the number of blocks sent between the client and server to serve all Access operations, over the number of accesses made (i.e., is amortized). Insecure block storage systems require O(B) bandwidth. When we say an ORAM requires $O(B\ logN)$ bandwidth, this may also be interpreted as $O(logN)$ bandwidth blowup/overhead relative to the insecure system. Note that some ORAM schemes only achieve their best bandwidth given large-enough blocks. If the allowed block size is larger than the client application's desired block size, the bandwidth blowup increases proportionally.

In addition to the primary metrics, we will analyze the following as they become relevant to different constructions.

Online bandwidth. The online bandwidth during each access refers to the blocks transferred before the access is completed from the client's point of view. By the “client's point of view," we are mainly interested in the case when the access type is read: i.e., online bandwidth represents the critical-path operation, the time between when the client requests a block and receives that block. To hide whether the operation type is read or write, however, ORAM schemes typically make Access perform the same operations from the server's perspective, regardless of operation type. So, for the rest of the thesis online bandwidth will refer to the blocks transferred before data is returned to the client, as if every client operation was a read. After the online phase of Access, more block transfers may be required before the access is complete, which we call the offline phase.

Worst-case bandwidth. The worst-case bandwidth refers to the per-Access bandwidth if amortization is not possible. For certain ORAM schemes, the bandwidth per call to Access is naturally the same for every call (in which case worst-case equals bandwidth). In other schemes, offline bandwidth can be pushed to future calls to Access to improve the online bandwidth of multiple consecutive requests. This is to improve performance of “bursty workloads:" if the client must make two read requests before proceeding in its computation, the effective online bandwidth is the online bandwidth of both calls to Access and the offline bandwidth of the first call to Access.

Server computation. The server computation is the amount of untrusted, local computation performed by the server, in addition to performing simple memory read/write operations. In the first ORAM papers **[1, 34]**, the server is assumed to only perform read and write operations to untrusted storage. In practice, many recent constructions have implicitly assumed the server is able to perform some amount of computation on data to reduce client-server bandwidth. Depending on the amount of computation, the computation may become the system bottleneck.

Number of round-trips. The number of round-trips refers to the round-trip block traffic between client and server during each call to Access. As in regular systems design, more roundtrips means worse performance since future operations must wait for the interconnect latency between client and server.

Number of accesses. The “number of accesses" metric characterizes how many times the ORAM client must access physical memory on average to satisfy each ORAM request. Two blocks at different addresses count as two distinct accesses even if they are accessed in the same roundtrip. This was the original metric considered by Goldreich and Ostrovsky in their original ORAM work (where they equivalently call it

the runtime blowup comparing the Oblivious RAM simulation and the original non-oblivious RAM).

We note that the “number of accesses metric" is in fact the same as bandwidth blowup if block sizes are uniform. However, these metrics do not necessarily agree when blocks have non-uniform sizes, e.g., in recent tree-based ORAM schemes **[2, 4]**, a “big data block, little metadata block" trick is commonly used to achieve better bandwidth costs.

Circuit size. The circuit size metric for ORAMs was first raised by Wang et al. **[31]**, and is defined as the total circuit size of the ORAM client algorithm Next over all execution rounds during each ORAM request.

2.4 Settings

Many of the techniques presented in the thesis are general and help improve any ORAM deployment.

Outsourced storage. Here, a client (e.g., a mobile device or in-house data management system) wishes to securely store data on a remote storage provider. We assume the storage provider acts as block storage (e.g., Amazon S3): the operations exposed to the client are to read/write blocks of data **[35]**. The trusted computing base (TCB) is the client machine: we wish to eliminate access pattern leakage, given a potentially malicious adversary, at all points beyond the client (e.g., the network, server, etc).

Secure processor. Here, a client wishes to outsource computation to a server or to obfuscate its execution in an Internet-of-Things (IoT) environment. In the outsourcing setting, there is a remote client, a secure processor on the server, and the rest of the server state (e.g., its DRAM / disk hierarchy). In a setup phase, the client loads data and (possibly) a program into the secure processor using conventional secure channels and attestation techniques. Once setup, the secure processor computes the result of running the program on the provided data, and sends it back to the client (also using secure channels). In the IoT setting, a processor collects and computes on data in a hostile environment where the adversary may have physical access to the device. The TCB is the secure processor and the remote client (if one exists).

As the program runs, we wish to eliminate access pattern leakage to main memory, given a potentially malicious adversary, when last-level cache (LLC) misses occur. Several possible attacks include cold boot **[36]**, intercepting data on the memory bus **[37, 38]**, BIOS flashing **[39, 40]**, and in general multiple processors (or helper modules such as the Intel Management Engine **[41]**) sharing main memory in space or time.

2.5 ORAM History

We now review prior ORAM work more generally. We start by describing the three main families of ORAM schemes. The goal is to show the progression of ideas over time. The three schemes detailed below all require O(B) client storage (asymptotically optimal).

2.5.1 Square root ORAM (1987)

The study of ORAM was initiated by Goldreich **[34]**, who sought to address the problem of software IP theft. This problem is similar to our secure processor setting: for a program running on a remote secure processor, one wishes to hide a program's control flow as determined by the address pattern to main memory. The trivial solution is to scan all of memory on each access, which has O(BN) online/overall bandwidth.

To address the high online bandwidth in the trivial scheme, Goldreich proposed the square root ORAM. In this design, the server memory is into two regions: a main O(N) block region and a shelter of size O($\sqrt{N}$) blocks. The main region is filled with O(N) real blocks and O($\sqrt{N}$) dummy blocks. All blocks are encrypted using a semantically secures scheme and shuffled together. How the permutation is selected is implementation dependent; the square root ORAM uses a random oracle / hash function followed by an oblivious sort.

To make an access, the client first scans the entire shelter. If the block is found there, the client reads a random, previously unread dummy block from the main region. Otherwise, the client uses the hash function to determine the address of the block of interest in the main region. Finally, the real or dummy block is re-encrypted and appended to the shelter. Thus, the online bandwidth is O($B\sqrt{N}$). The intuition for security is that each read scans the shelter, and performs a read to a random, previously unread slot in the main region.

Every O($\sqrt{N}$) accesses, the main region runs out of dummies and must be fully re-permuted by the client. **[34]** achieves this *eviction* step by using an oblivious sort and a new keyed hash function to re-mix the shelter into the main region and re-permute the main region. Using the sorting algorithm described in the paper, this step requires O(B) client storage, and O($BN\ logN$) bits to be transferred every O($\sqrt{N}$) accesses, giving the scheme a O($B\sqrt{N}\ logN$) amortized bandwidth.

2.5.2 Hierarchical ORAM (1996)

Goldreich and Ostrovsky proposed the hierarchical ORAM **[1]** to improve the online and overall bandwidth of the square root algorithm. The key idea is to, instead of having one main memory region and shelter, organize the server as a pyramid of permuted arrays where each array is geometrically (e.g., a factor of 2) larger than the previous array. Each permuted array acts as the main region in the square root ORAM, and thus is parameterized by a hash function and has space reserved for dummy blocks.

To access a block, each level in the pyramid is accessed as if it were the main region in the square root ORAM. To avoid collisions in the hash function for the smaller levels, each slot in each permuted array is treated as a bucket of size O($logN$) blocks. The hash function now maps blocks to random buckets. Buckets are downloaded atomically by the client when read/written to, and the bucket size is set to make overflow probability negligible. Thus, online bandwidth is O(log^2N): the cost to access O($logN$) buckets (one per level in the pyramid) of O($logN$) blocks each.

Instead of scanning a shelter, each block accessed is appended to the smallest level of the pyramid after that access. Eventually (like the shelter), the top (or root) of the pyramid will fill, and an eviction step must merge it into the second pyramid level. When the second level fills, it along with the first level is merged into the third level, so on to the largest level of O(N) blocks. In general, merging levels 0 through i involves completely re-shuffling the contents of those levels into a new array which becomes level *i + 1*. The worst case and amortized bandwidth cost of this operation is O($N\ log^2N$) and O(log^3N) blocks, respectively.

2.5.3 Tree ORAM (2011)

Shi et al. **[4]** proposed the tree ORAM to decrease the worst-case bandwidth cost of the hierarchical ORAM to be O($polylog\ N$) blocks.

The key idea in the tree ORAM is that, instead of blocks stored in level *i* having complete freedom on where they will be re-shuffled into in level *i+1* (as with the hierarchical solution), blocks may only live in a single pre-ordained bucket per level. This is accomplished by connecting the buckets in the hierarchical ORAM pyramid as if they were nodes in a **binary tree**, and associating each block to a **random path** of buckets from the top bucket (the root bucket) to a leaf in the tree.

To access a block the client first looks up a **position map**, a table in client storage which tracks the path each block is currently mapped to, and then reads all the buckets on the block's assigned path. The scheme achieves access pattern privacy by

re-mapping the accessed block to a new random path when it is accessed. Similar to **[1]**, the tree ORAM requires buckets to be size O($logN$) for reasons that will be described below. Thus, online bandwidth is also O($B\ log^2N$).

Similar to the hierarchical ORAM, each block accessed is appended to the root bucket at the end of each access. To prevent the root bucket (or any other bucket) from overflowing, an eviction procedure downloads O(1) buckets per level per access to try and push blocks down the tree subject to blocks needing to stay on their assigned paths. This operation has an amortized and worst-case bandwidth overhead of O($B\ log^2N$).

3 Path ORAM Family

3.1 Path ORAM

We present Path ORAM, an extremely simple Oblivious RAM protocol with a small amount of client storage. Partly due to its simplicity, Path ORAM is one of the most practical ORAM scheme known to date with small client storage. In paper **[2]** is proven that Path ORAM has a O($logN$) bandwidth cost for blocks of size B = (log^2N) bits. For such block sizes, Path ORAM is asymptotically better than the best known ORAM schemes with small client storage. Due to its practicality, Path ORAM has been adopted in the design of secure processors since its proposal.

3.1.1 The Path ORAM Protocol

We give an informal overview of the Path ORAM protocol. The client stores a small amount of local data in a stash. The server-side storage is treated as a binary tree where each node is a bucket that can hold up to a fixed number of blocks.

Main invariant. We maintain the invariant that at any time, each block is mapped to a uniformly random leaf bucket in the tree, and unstashed blocks are always placed in some bucket along the path to the mapped leaf.

Whenever a block is read from the server, the entire path to the mapped leaf is read into the stash, the requested block is remapped to another leaf, and then the path that was just read is written back to the server. When the path is written back to the server, additional blocks in the stash may be evicted into the path as long as the invariant is preserved and there is remaining space in the buckets.

3.1.1.1 Server Storage

Data on the server is stored in a tree consisting of buckets as nodes. The tree does not have to necessarily be a binary tree, but we use a binary tree in our description for simplicity.

Binary tree. The server stores a binary tree data structure of height L and 2^L leaves. The tree can easily be laid out as a flat array when stored on disk. The levels of the tree are numbered 0 to L where level 0 denotes the root of the tree and level L denotes the leaves.

Bucket. Each node in the tree is called a bucket. Each bucket can contain up to Z real blocks. If a bucket has less than Z real blocks, it is padded with dummy blocks to always be of size Z. It suffices to choose the bucket size Z to be a small constant such as Z = 4.

Path. Let $x \in \{0, 1, ..., 2^L - 1\}$ denote the *x-th* leaf node in the tree. Any leaf node x defines a unique path from leaf x to the root of the tree. $\mathcal{P}(x)$ is used to denote set of buckets along the path from leaf x to the root. Additionally, $\mathcal{P}(x, \ell)$ denotes the bucket in $\mathcal{P}(x)$ at level ℓ in the tree.

Server storage size. Since there are about N buckets in the tree, the total server storage used is about $Z \cdot N$ blocks.

3.1.1.2 Client Storage and Bandwidth

The storage on the client consists of 2 data structures, a stash and a position map.

Stash. During the course of the algorithm, a small number of blocks might overflow from the tree buckets on the server. The client locally stores these overflowing blocks in a local data structure S called the stash.

Position map. The client stores a position map, such that x := position[a] means that block a is currently mapped to the x -th leaf node – this means that block a resides in some bucket in path $\mathcal{P}(x)$, or in the stash. The position map changes over time as blocks are accessed and remapped.

3.1.1.3 Path ORAM Initialization

The client stash S is initially empty. The server buckets are initialized to contain random encryptions of the dummy block (i.e., initially no block is stored on the server). The client's position map is filled with independent random numbers between 0 and $2^L - 1$.

Access(op, a, data*)

1: x ← position[a]
2: position[a] ← UniformRandom (0…$2^L - 1$)
3: **for** $\ell \in \{0,1, ..., L\}$ **do**
4: $S \leftarrow S \cup$ ReadBucket($\mathcal{P}(x, \ell)$)
5: **end for**
6: data ← Read block a from S
7: **if** op := write **then**
8: $S \leftarrow (S - \{(a,data)\}) \cup \{(a, data^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, ..., 0\}$ **do**
11: $S' \leftarrow \{(a', data') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(position[a'], \ell)\}$
12: $S' \leftarrow$ Select min($|S'|$,Z) blocks from S'
13: $S \leftarrow S - S'$
14: WriteBucket($\mathcal{P}(x, \ell)$,S')
15: **end for**
16: **return** data

Figure 2: Path ORAM Access Algorithm

3.1.1.4 Path ORAM Reads and Writes

In Path ORAM construction, reading and writing a block to ORAM is done via a single protocol called Access described in Figure 2. Specifically, to read block a, the client performs data ← Access(read, a, None)and to write to write data* to block a, the client performs Access(write, a, data*). The Access protocol can be summarized in 4 simple steps:

1. **Remap block:** Randomly remap the position of block a to a new random position. Let x denote the block's old position.
2. **Read path:** Read the path $\mathcal{P}(x)$ containing block a.
3. **Update block:** If the access is a write, update the data stored for block a.
4. **Write path:** Write the path back and possibly include some additional blocks from the stash if they can be placed into the path. Buckets are greedily filled with blocks in the stash in the order of leaf to root, ensuring that blocks get pushed as deep down into the tree as possible. A block a' can be placed in the bucket at level ℓ only if the path $\mathcal{P}$(position[a']) to the leaf of block a' intersects the path accessed $\mathcal{P}(x)$ at level ℓ. In otherwords, if $\mathcal{P}(x, \ell)$= $\mathcal{P}$(position[a'], ℓ).

Note that when the client performs Access on a block for the first time, it will not find it in the tree or stash, and should assume that the block has a default value of zero.

Subroutines. We now explain the ReadBucket and theWriteBucket subroutine. For ReadBucket(bucket),the client reads all Z blocks (including any dummy blocks) from the bucket stored on the server. Blocks are decrypted as they are read. For WriteBucket(bucket, blocks), the client writes the blocks into the specified bucket on the server. When writing, the client pads blocks with dummy blocks to make it of size Z - note that this is important for security. All blocks (including dummy blocks) are re-encrypted, using a randomized encryption scheme, as they are written.

Computation. Client's computation is O($logN$) · ω(1) per data access. In practice, the majority of this time is spent decrypting and encrypting O($logN$) blocks per data access. We treat the server as a network storage device, so it only needs to do the computation necessary to retrieve and store O($logN$) blocks per data access.

3.1.2 Security Analysis

To prove the security of Path-ORAM, let $\vec{y}$ be a data request of size *M*. By the definition of Path-ORAM, the server sees $A(\vec{y})$ which is a sequence

$$p = (position_M[a_M], position_{M-1}[a_{M-1}], ..., position_1[a_1])$$

where position$_j$[a$_j$] is the position of the address a$_j$ indicated by the position map for the j-th load/store operation, together with a sequence of encrypted paths $\mathcal{P}$(position$_j$(a$_j$)), $1 \leq j \leq M$, each encrypted using randomized encryption. The sequence of encrypted paths is computationally indistinguishable from a random sequence of bit strings by the definition of randomized encryption (note that ciphertexts that correspond to the same plaintext use different randomness and are therefore indistinguishable from one another).

Notice that once position$_i$(a$_i$) is revealed to the server, it is remapped to a completely new random label, hence, position$_i$(a$_i$) is statistically independent of $\mathcal{P}$(position$_j$(a$_j$) for $i < j$ with a$_j$ = a$_i$. Since the positions of different addresses do not affect one another in Path ORAM, position$_i$(a$_i$) is statistically independent of position$_j$(a$_j$) for $i < j$ with a$_j \neq$ a$_i$. This shows that position$_i$(a$_i$) is statistically independent of position$_j$(a$_j$) for $i < j$, therefore, (by using Bayes rule) $\Pr(\mathrm{p}) = \prod_{j=1}^{\mathrm{M}} \Pr(\text{position}_j(a_j)) = (\frac{1}{2^L})^M$. This proves that $A(\vec{y})$ is computationally indistinguishable from a random sequence of bit strings.

3.1.3 Recursion

In the non-recursive scheme described in the previous section, the client must store a relatively large position map. We can leverage the same recursion idea as described in the ORAM constructions of Stafanov *et al.* [3] and Shi *et al.* [4] to reduce the client-side storage. The idea is simple: instead of storing the position ma on the client side, we store the position on the server side in a smaller ORAM, and recurs.

More concretely, consider a recursive Path ORAM made up of series of ORAMs called ORAM$_0$, ORAM$_1$, ORAM$_2$, ..., ORAM$_X$ where ORAM$_0$ contains the data blocks, the position map of ORAM$_i$ ORAM$_{i+1}$, and the client stores the position map for ORAM$_X$. To access a block in ORAM$_0$ the client looks up its position in ORAM$_1$, which triggers a recursive call to look up the position of the position in ORAM$_2$, and so on until finally a position of ORAM$_X$ is looked up in the client storage.

3.2 Optimization of Path Oblivious RAM in Secure Processors

In this section, we present techniques that proposed at **[7]** and make Path ORAM practical in a secure processor setting. The first technique is called background eviction scheme to prevent Path ORAM failure and allow for a performance-driven design space exploration. The second technique is called super blocks to further improve Path ORAM's performance, and also show an efficient integrity verification scheme for Path ORAM.

3.2.1 Background Eviction

To be usable, a background eviction scheme must:

a) Not change the ORAM's security guarantees,
b) Make the probability of stash overflow negligible and
c) Introduce as little additional overhead to the ORAM's normal operation as possible.

For instance, a strawman scheme could be to read/write every bucket in the ORAM tree when stash occupancy reaches a threshold – clearly not acceptable from a performance standpoint. Unfortunately, the strawman scheme is also not secure. If background evictions occur when stash occupancy reaches a threshold, the fact that background evictions occurred can leak privacy because some access patterns fill up the stash faster than others. For example, if a program keeps accessing the same block over and over again, the requested block is likely to be already in the stash – not increasing the number of blocks in the stash. In contrast, a program that scans the memory (i.e., accesses all the blocks one by one) fills up the stash much faster. If an attacker realizes that background evictions happen frequently, the attacker can infer that the access pattern of the program is similar to a memory scan and can possibly learn something about private data based on the access pattern.

One way to prevent attacks based on when background evictions take place is to make background evictions indistinguishable from regular ORAM accesses. The proposed background eviction scheme prevents Path ORAM stash overflow using dummy load/stores. To prevent stash overflow, it stops serving real memory requests and issue dummy requests whenever the number of blocks in the stash exceeds $C - Z(L+1)$. (Since there can be up to $Z(L+1)$ real blocks on a path, the next access has a chance to overflow the stash at this point.) A dummy access reads and decrypts a random path and writes back (after re-encryption) as many blocks from the path and stash as possible. A dummy access will at least not add blocks to the stash because all the blocks on that path can at least go back to their original places (note that no block is remapped on a dummy access). Furthermore, there is a possibility that some blocks in the stash will find places on this path. Thus, the stash cannot overflow and Path ORAM cannot fail, with this background eviction scheme. It keeps issuing dummy accesses until the number of blocks in the stash drops below the $C - Z(L+1)$ threshold, at which point the ORAM can resume serving real requests again.

This background eviction scheme can be easily extended to a hierarchical Path ORAM. If the stash of any of the ORAMs in the hierarchy exceeds the threshold, it issues a dummy request to each of the path ORAMs in the same order as a normal access, i.e., the smallest Path ORAM first and the data ORAM last.

Liveclock. The proposed background eviction scheme does have an extremely low probability of livelock. Livelock occurs when no finite number of background evictions is able to reduce the stash occupancy to below $C - Z(L+1)$ blocks. For example, all

blocks along a path may be mapped to the same leaf ℓ and every block in the (full) stash might also map to leaf ℓ. In that case no blocks in the stash can be evicted, and dummy accesses are continually performed (this is similar to a program hanging). However, the possibility of such a scenario is similar to that of randomly throwing 32 million balls (blocks) to 16 million bins (leafs) with more than 200 balls (stash size) landing into the same bin—astronomically small (on the 10^{-100} scale). Therefore, we do not try to detect or deal with this type of livelock. It is noted that livelock does not compromise security.

3.2.1.1 Security of the background eviction

Background eviction scheme does not leak any information. Recall that the original Path ORAM (with an infinite stash and no background eviction) is secure because, independent of the memory requests, an observer sees a sequence of random paths being accessed, denoted as

$$P = \{p_1, p_2, \ldots, p_k, \ldots\},$$

where p_k is the is the path that is accessed on k - th memory access. Each p_k, (k = 1, 2, …) follows a uniformly random distribution and is independent of any other p_j in the sequence. Background eviction interleaves another sequence of random paths q_m caused by dummy accesses, producing a new sequence

$$Q = \{p_1, p_2, \ldots, p_{k1}, q_1, \ldots, q_{k2}, q_2, \ldots\},$$

since q_m follows the same uniformly random distribution with p_k and q_m is independent of any p_k and any q_n ($n \neq m$), Q also consists of randomly selected paths, and thus is indistinguishable from P. This shows the security of the proposed background eviction.

3.2.2 Super Blocks

Another way to improve Path ORAM's efficiency is to increase the amount of useful data per ORAM access, by loading multiple useful blocks on an ORAM access. However, this is almost impossible in the original Path ORAM, since blocks are randomly dispersed to all the leaves and are unlikely to reside on the same path.

To load a group of blocks on a single access, these blocks have to be intentionally mapped to the same leaf in the ORAM tree. Such a group of blocks is called as super block. It is important to note that the blocks within a super block S do not have to reside in the same bucket. Rather, they only have to be along the same path so that an access to any of them can load all the blocks in S.

When a block b ∈ S is evicted from on-chip cache, it is put back into the ORAM stash without waiting for other blocks in S. At this point it can find its way to the ORAM tree alone. When other blocks in S get evicted from on-chip cache at a later time, they will be assigned and evicted to the same path as b. We remark that this is the reason why super blocks are not equivalent to having larger blocks (cache lines): a cache line is either entirely in on-chip cache, or entirely in main memory.

Super blocks create other design spaces for Path ORAM, such as super block size, which blocks to merge, etc. In paper **[7]**, only adjacent blocks are merged in the address space into super blocks. This can exploit most of the spatial locality in an application, while keeping the implementation simple. The following scheme is used.

Static merging scheme. Only merge adjacent blocks (cache lines) into super blocks of a fixed size. The super block size is determined and specified to the ORAM interface before the program starts. At initialization stage, data blocks are initially written to ORAM, and the ORAM interface simply assigns the same leaf label to the blocks from the same super block. The additional hardware required is small.

3.2.2.1 Security of Super Blocks

For the same reasons as background eviction, an access to a super block must be indistinguishable from an access to a normal block for security reasons. In the scheme above, a super block is always mapped to a random leaf in the ORAM tree in the same way as a normal block. If any block in the super block is accessed, all the blocks are moved from ORAM to on-chip cache and also remapped to a new random leaf. A super block access also reads and writes a path, which is randomly selected at the previous access to this super block. This is exactly the Path ORAM operation. Splitting/merging super blocks is performed on-chip and is not revealed to an observer.

3.3 Circuit ORAM

In this section we present the tree-based ORAM scheme called Circuit ORAM [8]. Circuit ORAM shows that the well-known Goldreich-Ostrovsky logarithmic ORAM lower bound is tight under certain parameter ranges, for several performance metrics. From a practical perspective, Circuit ORAM earns its name because it achieves (almost) optimal circuit size both in theory and in practice for realistic choices of block sizes. Thus, Circuit ORAM is ideal for secure multi-party computation applications.

3.3.1 The Circuit ORAM Protocol

Circuit ORAM follows the tree-based ORAM framework, by building a binary tree containing N nodes (referred as buckets), where each bucket can store Z = O(1) number of blocks. We provide two definitions *Legally side* and *Deepness w.r.t eviction path before* below before we provide a detailed scheme description.

Definition 5 (Legally reside). We say that a block B can legally reside in path[ℓ] if by placing B in path[ℓ], the main path invariant is satisfied.

Definition 6 (Deepness w.r.t eviction path). For a given eviction path, block B_0 is deeper that block B_1 (with respect to path), if there exists some path[ℓ] such that B_0 can legally reside in path[ℓ], but B_1 cannot; in the case when both blocks can legally reside in the same buckets along path, the block with smaller index **idx** will be considered deeper.

In other word B_0 is deeper on the current eviction path than B_1 if it can legally reside nearer to the leaf along path. If two blocks have the same deepness, we use their indices idx to resolve ambiguity. The notion of deepness and greedy eviction choice of the deepest block on a path came by the novel ideas of the CLP ORAM [9], but it is applied in a different manner.

3.3.1.1 Intuition

The main idea is to have an eviction algorithm that is easy to implement as a small circuit. Ideally it should make a single scan of the data blocks on the eviction path from the stash to leaf (and only a constant number of metadata scans), and still try to push blocks towards the leaf as much as possible.

During the one-pass scan of the data blocks, the client “picks up" (i.e., remove from path) and holds onto one block, which can later be “dropped" somewhere further along the path. At any point of time, the client should hold onto at most one block. Further, it makes sense for the client to hold onto the currently deepest block when it does decide to hold a block. This way, the block in holding will have the maximum chance of being dropped later. On encountering a deeper block, the client could swap it with the one in holding.

However, a dilemma arises. How does the client decide when it should pick up a block and hold onto it? Maybe this block will never get a chance to be dropped later, in which case there will be two equally bad choices: 1) put the block into the stash - which results in rapid stash growth; and 2) go back and revisit the path to write the block back. However, doing this obliviously results in high cost.

Remedy: lookahead mechanism with two metadata scans. The above issues result from the lack of foresight. If the client could only know when to pick up a block and place it in holding, and when to write the block back into an available slot, then these issues would have been resolved. Instead of that the idea is to rely on two metadata scans prior to the real block scan, to compute all the information necessary for the client to develop this foresight. These metadata scans need not touch the actual blocks on the eviction path, but only metadata information such as the leaf label for each block, and the dummy bit indicator for each block. If the bucket size is O(1), then the bandwidth blowup is O($logN$).

EvictOnceSlow(path)

1: := L
2: **while** $i \leq 1$ **do**:
3: **if** path[i] has empty slot **then**
4: (B, ℓ) := Deepest block in path[0.. i - 1] that can legally reside in path[i]
5: **end if**
6: **if** B $\neq \perp$ **then**
7: Move B from path[ℓ] to path[i]
8: $i := \ell$
9: **else**
10: $i := i - 1$
11: **end if**
12: **end while**

Figure 3: EvictOnceSlow Algorithm

3.3.1.2 Detailed Scheme Description

A slow and non-oblivious version of the eviction algorithm. To aid understanding, we first describe a slow, non-oblivious version of the Circuit eviction algorithm, *EvictOnceSlow*, as shown in Figure 3. This slow version only serves to illustrate the effect of the eviction algorithm, but does not describe how the algorithm can be efficiently implemented in circuit. Furthermore, this slow, non-oblivious version of the Circuit eviction algorithm gives a simpler way to reason about the stash usage of the algorithm. Later in this section, we describe how to implement the Circuit eviction algorithm efficiently and obliviously by making use of two metadata scans and a one real block scan; this can be readily converted into a small-sized circuit. The *EvictOnceSlow* algorithm makes a reverse (i.e., leaf to stash) scan over the current eviction path. When it first encounters an empty slot in path[i], it will try to evict the deepest block B in path[0..i – 1] to this empty slot, provided that the block B can legally reside in path[i]. Suppose this deepest block B resides in path[ℓ] where ℓ< i. After relocating the block B to path[i], the algorithm now skips levels path[ℓ+1.. i – 1], and continues its reverse scan at level ℓ instead (Line 8 in Algorithm 1). In case no block in path[0..i – 1] can fill the empty slot in path[i], the scan simply continues to level path[i – 1].

Circuit ORAM eviction algorithm. In Figure 3, Line 4 is inefficient, and Line 8 is non-oblivious. We now present how to implement the same EvictSlow algorithm obliviously and efficiently, but using two metadata scans (Algorithms *PrepareDeepest* and PrepareTarget) plus a single real block scan (Algorithm 4). Since metadata is typically much smaller than real data blocks, a metadata scan is faster than a real block scan. The two metadata scans will generate two helper data structures:

- An array deepest[1..L], where deepest[i] = ℓ means that the deepest block in path[0..i – 1] that can legally reside in path[i] is now in level ℓ < i. If no block in path[0..i – 1] can legally reside in path[i], then deepest[i] := ⊥. In the pre-processing state, one metadata scan will be used, namely the PrepareDeepest subroutine (see Figure 4), to populate the deepest array. This allows us to avoid Line 4 in Figure 3 causing an additional Θ(L) overhead.
- An array target [0..L], where target[i] stores which level the deepest block in path[i] will be evicted to. This target array is prepopulated using a backward metadata scan as depicted in the PrepareTarget algorithm (see Figure 5).

Observe that the prepopulated target array basically gives a precise prescription of the client's actions (including when to pick up a block and when to drop it) during the real block scan At this moment, the client performs a forward block scan from stash to leaf, as depicted in the EvictOnceFast algorithm (see Figure 6). The high level idea here is to “hold a block in one's hand" as one scans through the path, where the block-in-hand is denoted as hold in the algorithm. This block hold will later be written to its appropriate destination level, when the scan reaches that level.

Eviction rate and choice of eviction path. For each data access, two paths are chosen for eviction using the EvictOnceFast algorithm. While other approaches are conceivable, we describe two simple ways for choosing the eviction paths:

- A *random-order* eviction strategy denoted EvictRandom() (see Figure 7). The randomized strategy chooses two random paths that are non-overlapping except at the stash and the root. This means that one path is randomly chosen from each of the left and the right branches of the root.
- A *deterministic-order* strategy denoted EvictDeterministic() (see *Figure 8*). The deterministic-order strategy is inspired by Gentry et al. [12] and several subsequent works [42, 43].

Recursion. So far, we have assumed that the client stores the entire position map. Based on a standard trick [2, 3, 4], we can recursively store the position map on the server. In the position map recursion levels, Circuit ORAM uses a different block size than the main data level as suggested by Stefanov et al. [2]. Specifically, a group c number of labels in one block for an appropriate constant c > 1. In other words, the

block size for position map levels is set to be D' = O($logN$), resulting in O($logN$) depth of recursion. In this way, the total bandwidth cost over all recursion levels would be O(D $logN + log^3 N$) · ω(1) (for negligible failure probability), assuming that the stashes reside on the server side, and the hence client only needs to hold a constant number of blocks at any time. For inverse polynomial failure probability, the total bandwidth cost is O(D $logN + log^3 N$).

Security Proof. The security proof is trivial. First, as in all tree-based ORAMs, every time a block is read or written, a random path is read, where the random choice has not been revealed to the server before. This part of the proof is trivial, and the same as Shi et al. [4]. It remains to show that the eviction process is oblivious too. As we can see from Algorithms PrepareDeepest, PrepareTarget and EvictOnceFast, eviction on a selected path always reads blocks or metadata (stored on the server) in a sequential manner, either from leaf to root or from root to leaf. Clearly this does not depend on the logical address being read or written. In fact, saying that the eviction algorithm (Figure 6) is oblivious is the same as saying that it can be implemented efficiently in circuit representation. Finally, no matter whether it is used random-order eviction (Figure 7) or deterministic order eviction (*Figure 8*), the choice of the eviction path is also independent of the logical address sequence being read/written.

```
PrepareDeepst(path)
1: Initialize deepest := (⊥, ⊥, …, ⊥), src := ⊥, goal := ⊥
2: if stash not empty then
3:    src := 0
4:    goal := Deepest level that a block in path[0] can legally reside on path
5: end if
6: for i = 1 to L do
7:    if goal ≥ i then deepest[i] := src
8:    end if
9:    ℓ := Deepest level that a block in path[i] can legally reside on path
10:   if ℓ > goal then
11:      goal := ℓ, src := i
12:   end if
13: end while
```

Figure 4: PrepareDeepest Algorithm

```
PrepareTarget(path)
1: dest := ⊥, src := ⊥, target := (⊥, ⊥, …, ⊥)
2: for i = L downto 0 do
3:    if (i == src) then
4:       target[i] := dest, dest := ⊥, src := ⊥
5:    end if
6:    if ((dest = ⊥ and path[i] has empty slot) or (target[i] ≠ ⊥) and (deepest[i] ≠ ⊥) then
7:          src := deepest[i]
8:          dest := i
9:    end if
10: end for
```

Figure 5: PrepareTarget Algorithm

```
EvictOnceFast(path)
1: Call the PrepareDeepest and PRepareTarget subroutins to pre-process arrays deepest and target
2: hold := ⊥, dest := ⊥
3: for i = 0 to  L do
4:      towrite := ⊥ (B, ℓ) := Deepest block in path[0.. i - 1] that can legally reside in path[i]
5:      if (hold ≠ ⊥) and (i == dest) then
6:         towrite := hold
7:         hold := ⊥, dest := ⊥
8:      end if
9:      if (target[i] ≠ ⊥) then
10:        hold := read and remove deepest block in path[i] dest :=
11:        dest := target[i]
12:     end if
13:     Place towrite into bucket patjh[i] if towrite ≠ ⊥
14: end for
```

Figure 6: EvictOnceFast Algorithm

```
EvictRandom()
1: Choose a leaf from each of the left and the right branches of the root independently, and denote the two
   corresponding (stash-to-leaf) paths by path0 path1
2: Call EvictOnceFast(path0) and EvictOnceFast(path1)
```

Figure 7: EvictRandom Algorithm

```
EvictDeterministic()
In timestep t:
1: Choose two paths, path0 and path1, corresponding to the leaves labeled with integers bitrev(2t mod N) and
   bitrev((2t + 1) mod N), respectively. In the above bitrev(i) denotes the integer obtained by reversing the bit
   order of i when expressed in binary
2: Call EvictOnceFast(path0) and EvictOnceFast(path1)
3: Increase t by 1 for the next access
```

Figure 8: EvictDeterministic Algorithm

3.4 Bucket ORAM

Bucket ORAM [44] protocol designed to achieve bandwidth efficiency, and at the same time to be compatible with known techniques for both single online roundtrip as well as constant bandwidth blowup. Bucket ORAM is a hybrid between two ORAM frameworks and enjoys nice properties of both.

Roughly speaking, two types of ORAM constructions have been proposed in the past, 1) those based on the hierarchical framework initially proposed by Goldreich and Ostrovsky [1]; and 2) those based on the tree-based framework initially proposed by Shi et al. [4].

Both hierarchical ORAMs and tree-based ORAMs share the common feature of having exponentially growing levels. One fundamental difference between the two frameworks is whether they impose restrictions on a block's location within a level, and how they treat shuffling of data blocks (also referred to as eviction).

Bucket ORAM adopts the level-rebuild-style data shuffling of hierarchical ORAMs, where larger levels are rebuilt exponentially less often than smaller levels. At the

same time, Bucket ORAM also places restrictions where blocks can reside in a level much as in tree-based ORAMs. In this way, Bucket ORAM's level rebuild eviction involves the client performing local operations on O(1) number of buckets at a time, and there is no need to perform global oblivious sorting on entire levels.

3.4.1 The Bucket ORAM Protocol

Bucket ORAM features level-rebuild style shuffling just like in hierarchical ORAMs, but avoids having to perform oblivious sorts globally on all blocks within a level. Instead, blocks within a level are divided into ω($logN$)-sized buckets, and the client works on four buckets at a time during the level rebuild.

Server layout. Sever storage is organized into L = O($logN$) levels where level ℓ contains 2^{ℓ} buckets for $\ell \in \{0, 1, \dots, L-1\}$. Each bucket stores Z blocks, and each block is of B bits in length. Each block is either a real block or a dummy block, and if the number of real blocks is < Z, we fill the rest of the bucket with dummies. To help achieve local shuffles, the server storage is treated as a binary tree where each bucket is a node in the tree. We get a tree structure by first labeling each bucket in level ℓ with a unique identifier $i \in \{0, 1, \dots, 2^{\ell} - 1\}$. This way, each bucket in level $\ell < L - 1$ has two distinct child buckets in level ℓ + 1, whose respective identifiers are 2i and 2i + 1.

Client Storage. Like in tree-based ORAM schemes, the client stores an O($N\ logN$)-bit position map that maps each block to a bucket in level $L - 1$. The client also stores an eviction buffer of size Z' = Z/O(1) (denoted ebuffer) containing blocks that have not been evicted yet.

Block metadata. To enable all ORAM operations, each block carries with it some metadata besides it payload. For the purpose of this section, each block's metadata include its type (either “real" or “dummy"), and for a real block, its logical address idx and path label it's mapped to. Therefore, block formats are as below:

Real Block: (“real”, data, idx, label)
Dummy Block: (“dummy”, data=_, idx=_, label=_)

Main Invariant. Like in tree-based ORAMs, every block is assigned a position label indicating a leaf node in the tree, and the mapping is stored in the position map. A block always resides along the path to its designated leaf node.

Recursion. The position map is the O($N\ logN$)-bit. The client can reduce this local storage to O(1) bits by recursively storing the position map in other smaller ORAMs using the standard recursion technique [4, 2].

Requesting a block. In the basic Bucket ORAM construction, a client requests (either to read or to update) a block like in any tree-based ORAM (*Figure 9*). To request a block with logical address idx, the client looks up the position map to identify the path where the block resides. The client then reads every block in the eviction buffer and on the corresponding path. The client is guaranteed to find the block idx in the process. Finally, the requested block (possible updated by the client) is appended to the client's local eviction buffer.

Eviction. After Z' requests, the client-side eviction buffer may be full and called an eviction routine (*Figure 10*). The goal of the eviction step is to write the blocks in the eviction buffer back to the server. In [44] is described a novel eviction routine that achieves hierarchical ORAM's level rebuild using only local reshuffle. Therefore, Bucket ORAM does not require large client storage or global oblivious sorting on a large set of blocks.

```
ORAM.Request(op, idx, data)
// Precondition: ebuffer is not full
1: label* ← UniformRandom({0,1}^(2^(L−1)))
2: label ← position[idx], position[idx] ← label*
3: block ← ⊥
4: for each block_0 ∈ {ebuffer ∪ P(label)}  do //path from leaf label to root
5:    if block_0.idx = idx then
6:       block ← block_0
7:    end if
8: end for
9: if op is "write": block.data ← data
10: block.label ← label*
11: ebuffer ← ebuffer ∪ {block}
12: return block.data
```

Figure 9: Bucket ORAM Request Algorithm

```
ORAM.Eviction(block)
// Happens every Z' requests
1: Pad ebuffer with Z − Z' or more dymmy blocks such that its size is Z
2: if AllLevels[0] is empty on server then
3:    Write back ebuffer to AllLeves[0]
4:    return
5: end if
6: Let level ← (ebuffer)
7: Let ℓ* := first empty level or L − 1 if all levels are full
8: for ℓ = 0 to ℓ* − 1 do
9:    level ← Merge(level, AllLevels[ℓ])
10: end for
11: if AllLevels[ℓ*] is empty then
12:    AllLevels[ℓ*] := level
13: else // ℓ* must be L − 1
14:    AllLevels[ℓ*] := MergeInPlace(AllLevels[ℓ*], level)
15: end if
```

Merge(level, $\overline{\text{level}}$)

// level and $\overline{\text{level}}$ each contains m buckets and resides on server
1: Initialize an empty new $\text{level}_{\text{new}}$ of size $2m$ on server
2: **for** $i \in \{0, 1, \dots, m-1\}$ **do**
3: Fetch level[i] and $\overline{\text{level}}[i]$ from server
4: $mid \leftarrow (2i+1)N/2m$
5: $S_L \leftarrow$ Left(level[i], mid) ∪ Left($\overline{\text{level}}$ [i], mid)
6: $S_R \leftarrow$ Right(level[i], mid) ∪ Right($\overline{\text{level}}$ [i], mid)
7: $\text{bucket}_L \leftarrow$ ConstructBucket(S_L), $\text{bucket}_R \leftarrow$ ConstructBucket(S_R)
8: Write back bucket_L to $\text{level}_{\text{new}}[2i]$
9: Write back bucket_R to $\text{level}_{\text{new}}[2i+1]$
10: Return (a pointer to) $\text{level}_{\text{new}}$
11: **end for**

MergeInPlace(level, $\overline{\text{level}}$)

// level and $\overline{\text{level}}$ each contains m buckets
1: Initialize an empty new $\text{level}_{\text{new}}$ of size m on server
2: **for** $i \in \{0, 1, \dots, m-1\}$ **do**
3: Fetch level[i] and $\overline{\text{level}}[i]$ from server
4: $\text{bucket}_L \leftarrow$ ConstructBucket(level[i] ∪ $\overline{\text{level}}$ [i])
5: Write back bucket to $\text{level}_{\text{new}}[i]$
6: **end for**

Left(bucket, mid)

1: Return {block ∈ bucket: (block is "real") ∧ (block.label < mid)}

Right(bucket, mid)

1: Return {block ∈ bucket: (block is "real") ∧ (block.label ≥ mid)}

Figure 10: Bucket ORAM Evict Algorithm

3.4.2 Bucket ORAM Construction with No Position Map

The standard recursion techniques [2-4, 8, 14] in the basic construction requires multiple O($logN$) interactions with the server for the online phase. In this section, we describe techniques that will allow to get rid of the client-side position map (hence recursion is not needed). In this section, the described techniques require that the server perform computation. Therefore, strictly speaking, we are in the setting of (Verifiable) Oblivious Storage as phrased by Apon et al. [28].

Intuition. In this new scheme, developed techniques so that the client can read exactly one block per level (thereby improving online bandwidth) yet be able to execute the entire request operation in a single roundtrip. For this purpose, we need each block to be one of three types, a real block, a mask block, and a dummy block. A dummy block signals an empty slot in the level data structure. Real blocks are the same as before. However, a new type of block introduced that called a mask block. Upon being rebuilt, a level will have $Z' \cdot 2^{\ell}$ mask blocks (as many as there are real blocks), and every mask block has a counter maskcnt ∈ $\{1, 2, \dots, Z' \cdot 2^{\ell}\}$. The format of different blocks are summarized below where _ denotes "don't care" fields for each type of block:

Real Block: (“real”, data, idx, label, maskcnt = _)
Mask Block: (“mask”, data =_, idx = _, label = _, maskcnt)
Dummy Block: (“dummy”, data = _, idx = _, label = _, maskcnt = _)

As in hierarchical ORAM and Ring ORAM [11], during each request the client will download the requested (real) block from a level if it is present, or a previously unread mask block otherwise. For security, the distribution of the mask blocks within the level must be identical to the distribution of real blocks in the level. And obviously each level must be rebuilt before all the mask blocks can possibly be consumed. Dummy blocks simply fill in the remaining empty spaces.

Server stores blocks in a hash table. To find the real block (or next mask block) in each level, we leverage a server-side hash table technique first proposed by Williams and Sion [45, 46]. For every (encrypted) block stored on the server, either real, mask, or dummy, the client uses a pseudo-random function to compute a hash key (denoted hkey), and reveals hkey to the server. The server builds a hash table over the blocks, such that blocks can be looked up by their hkey. Now, to read a block from a level, the client would perform a sequence of actions at the end of which produces the hkey of the block to be retrieved. Specifically, if the block to be retrieved is within the current level, the client computes the real hkey of the block. If the block is not in the current level, the client would compute the hkey for the next mask block. As in the previous paragraph, for security we guarantee that the client will ask for each hkey only once before the hash table is rebuilt. The client now reveals the hkey to the server to fetch this block. In this way, the client only fetches a single block from each level instead of Z. For security, it is important that hkey be pseudorandom and time-dependent. In other words, the same block will have a different hkey when it is written to the server again in the future.

Bloom filter. Accompanying the hash table trick, in this constructions adopted a standard Bloom Filter trick first introduced by Williams and Sion [45, 46]. To allow the client to determine whether a level contains the desired block, each level has an auxiliary (encrypted) Bloom filter. To look up a block in a level, the client looks up k locations in the the Bloom filter first to determine if the level contains the desired block. For security, if the desired block has been found in smaller levels, the client looks up k random locations in the Bloom filter. Otherwise, it looks up k real locations.

Oblivious sorts on metadata during level rebuilding. During a level rebuilding, the client performs O(1) number of oblivious sorts on metadata to achieve the following goals:

- Rebuild the encrypted Bloom filter in synchrony with the new level being rebuilt.

- Randomly distribute mask blocks to all buckets in a level, and assign unique mask counters.

Pseudo-random hash keys for blocks. Level ℓ has $Z' \cdot 2^{\ell}$ mask blocks, and every mask block has a unique mask counter denoted maskcnt $\in \{1, \ldots, Z' \cdot 2^{\ell}\}$. This guarantees that each level has at least as many mask blocks as there are real blocks. The mask counters for all mask blocks within a level must be (pseudo-)randomly distributed to buckets for security reasons. Then the every block has a hash key defined as follows:

$$\text{Hkey(block)} := \begin{cases} PRF_H(sk, T \| block.idx) & if\ real\ block \\ PRF_H(sk, T \| \text{"}mask\text{"} \| maskcnt) & if\ mask\ block \\ random\ string & o.w. \end{cases}$$

where T denotes the time step when the level has rebuilt, $PRF_{subscription}$ denotes different pseudo-random functions employed by the client, and sk denotes a client secret key that is kept confidential from the server. The level number can be uniquely inferred by the timestamp T, and therefore we do not separately include the level number in the PRF's input.

Leverage a per-level Bloom filter to compute the hkey of block to fetch. To allow the client to efficiently query whether a block is contained in a specific level, the client stores an auxiliary data structure, an encrypted Bloom filter on the server. The check if a block is within a specific level, the client computes k locations in the Bloom filter as

$$loc_i := PRF_{BF}(sk, T \| block.idx \| i) \qquad (1)$$

Access Algorithm that fetches only one block per-level

For each level ℓ, the client stores a current mask counter denoted $maskcnt_{\ell}$. When a level ℓ is rebuilt, its $maskcnt_{\ell}$ is reset to 0.
To look up a block idx, the client does the following:
Initialize found := false, Next, for each level ℓ = 0 to $L - 1$:

Look up Bloom filter:

- If not found, look up k real locations in this level's Bloom filter, where location i is computed as in Equation above (1). If all k locations are 1, conclude that the block is in this level, and set found := true.
- Else if found, lookup k random locations in this level's Bloom filter.

Compute and reveal hash key to server:

- If block idx is in level ℓ, then reveal the real hkey := $PRF_H(sk, T \| block.idx)$ to the server, where T denotes the last time level ℓ was rebuilt.
- If bloci idx is not in level ℓ, reveal a mask hkey := $PRF_H(sk, T \| block.idx \| maskcnt_{\ell})$ to the server, and then increment $maskcnt_{\ell}$.

Retrieve block:

- The server now returns the block with the specified hkey to the client (along with the block's location within the level). The online phase ends here.
- In an offline roundtrip, the client marks the block fetched invalid, by setting block.type := "dummy".

Figure 11: Improved Bucket ORAM Access Algorithm

where $1 \leq i \leq k$ and T denotes tha last time this level was rebuilt. However, the moment that a block is found in a level, for all future levels, the client looks up k random locations instead.

Putting everything together, the new algorithm for requesting a block is described in *Figure 11*. In this algorithm, the client fetches only one block per-level during a request.

Eviction/Level-rebuild. The algorithm for evicting back blocks to the server and rebuilding levels proceeds in a similar fashion as in Section 3.4.1. However, now the client has to additionally 1) obliviously rebuild metadata for this level; and 2) randomly distribute mask blocks to buckets, and assign a unique mask counter maskcnt from the range $\{1, \dots, Z' \cdot 2^{\ell}\}$ to each mask block. The random redistribution of mask blocks to buckets is required to ensure the mask blocks in the rebuilt level have the same distribution as real blocks from the server's perspective. We note that only the rebuilt level requires the above two steps, i.e., rebuilding of metadata and redistribution of mask blocks. Intermediate levels created during the cascading merge will be empty after the eviction completes and will not be touched during request operations until they themselves are rebuilt. At a high level, the new level rebuild algorithm works as follows.

Quadruplet merges on data blocks. First, the client performs quadruplet merges as in *Figure 10*. At this moment, observe that 1) the hkeys of the blocks have not been revealed to the server; and 2) the rebuilt level contains only real and dummy blocks (i.e., no mask blocks).

Assign mask counters, and randomly distribute mask blocks to buckets. When a rebuilt level is being accessed in the future, it is crucial for security that from the server's perspective, every real or mask block is assigned to a random bucket, and then a random location within the bucket. Real blocks that get merged in the level have random leaf labels whose values have not been revealed to the server earlier. And this guarantees that every real block is residing in a random bucket from the server's perspective. Mask blocks should have this same distribution over the buckets in the level. Every mask block must also be assigned a unique maskcnt from a contiguous range $\{1, \dots, Z \cdot 2^{\ell}\}$. It is described an oblivious procedure for achieving the above in *Figure 12*. This algorithm relies on O(1) oblivious sorting operations on metadata to distribute all mask blocks to random buckets. Finally, at the end of this step, hkeys of blocks are revealed sequentially to the server.

Rebuild Bloom filter. Whenever a level is being rebuilt, the client rebuilds the level's Bloom filter in synchrony with the new level. Doing so would require O(1) oblivious sorts on metadata only as shown in *Figure 13*. This algorithm is standard and a similar version was described by William and Sion [45, 46].

Randomly distribute mask blocks to buckets

Input: A level containing m buckets, where each bucket contains Z blocks, either real or dummy (there is no mask block in a newly built level after the quadruplet merge step).
Output: A new level, where real blocks reside at random locations in the same bucket. Let n_1, …,n_{2^ℓ} be random variables denoting the number of balls in each bin when we throw $Z' \cdot 2^\ell$ balls into 2^ℓ bins. Each bucket $i \in [2^\ell]$ has exactly n_i mask blocks residing at random locations, where each mask block has a unique mask counter maskcnt $\in \{1, \ldots, Z' \cdot 2^\ell\}$. The remainder empty slots in each bucket are populated with dummy blocks.

1. **Metadata array creation.** Create an array containing all blocks' metadata. Make a linear scan over the level and for each bucket i = 0 to $2^\ell - 1$, append ("dummy", _, i) for each dummy block and ("real", _, i) for each real block. Then, for j = 1 to $Z \cdot 2^\ell$: pick a random bucket i within this level, and append the metadata entry ("mask", j, r) to the array. The tuple stipulates the "mask" block with maskcnt = j will end up in bucket r. Note that the number of metadata entries assigned to each bucket is guaranteed to be $\geq Z$ at this point.

2. **O-sort metadata.** Oblivious sort the above array based on increasing order of bucket number (the last element in the tuple). For the same bucket number, place real block before mask blocks before dummy blocks.

3. **Linear scan.** Make a linear scan over the array, and for each bucket number: preserve the first Z entries and rewrite all remaining entries as ("dummy", _, ∞).

4. **O-sort metadata.** Oblivious sort the above array based on increasing order of bucket number. When sorted, preserve the first $Z \cdot 2^\ell$ entries and discard the remainder of the array.

5. **Permute blocks within each bucket.** One bucket at a time, read the next Z entries from the array (metadata for real, dummy and mask blocks) and all Z data blocks in the bucket (some of which will be real blocks). Randomly permute blocks and metadata within the bucket (on the client side), and write the bucket back.

Figure 12: Randomly distribute mask blocks to buckets during level rebuilding

Obliviously rebuild Bloom filter

Input: The metadata of the real, mask, and dummy blocks within a level, residing on the server side.
Output: A Bloom filter for this level, residing on the server side.

1. **Initialization.** Make a linear scan over the metadata. For each block:
   - If block is real: create k pairs (on server): $\{(loc_i, 1)\}_{i \in [k]}$ where loc_i denotes a location in the Bloom filter that should be set to 1.
   - Else if block is mask or dummy: create k pairs (on server): {(⊥, ⊥), …, (⊥, ⊥)}

2. **Padding.** Let m be the total number of blocks (all three types) in a level. At the end of the last step, we have an array containing $k \cdot m$ pairs. Pad this array with the pairs (1, ⊥), (2, ⊥), …, ($BFSize$, ⊥), where $BFSize$ is the Bloom filter size.

3. **O-sort array.** Oblivious sort the padded array lexigraphically where ⊥ is considered lexigraphically larger than everything else. In the sorted array, all entries (i, _) appear before (j, _) if $i > j$ where _ denotes wildcard. Further, all (i, 1) pairs appear before the (i, ⊥) pair. All (⊥, ⊥) pairs appear at the end.

4. **Deduplicate pairs.** In the linear scan, preserve only the first occurrence of each (i, _) and rewrite all other occurrences as (⊥, ⊥).

5. **O-sort array.** Oblivious sort the resulting array lexigraphically. The sorted result should be of the format.

   (1,_), (2,_), …, (m, _), (⊥, ⊥), …, (⊥, ⊥)

   where each (i, _) is either (i, 1) denoting that the i-th bit of the Bloom filter should be set, or (i, ⊥) denoting that the i-th bit of the Bloom filter should be clear.

6. **Finalize Bloom filter.** In a synchronized scan of the above array and the Bloom filter, sequentially set all bits of the Bloom filter as indicated.

Figure 13: Obliviously rebuild Bloom filter in synchrony with a newly rebuilt level

3.4.3 Security Analysis

In this sections we present the security of the Bucket ORAM protocol and show that satisfies malicious security.

Lemma 1 (Distribution of block locations within a level). From the server's perspective, after a level is rebuilt, every real and mask block resides in an independent, random bucket.

Proof. First, every real block is assigned a fresh, random leaf label when the block was last fetched. This random choice of leaf label is kept hidden from the server until the block is next requested. This leaf label places a real block in a random bucket within the level. Second, mask blocks are distributed to random buckets during the level rebuild procedure.

Lemma 2 (Every hkey is fetched only once from the server). Every hash key hkey for a real or mask block is fetched only once by the server before the level is rebuilt. Further, the request phase will not run out of mask blocks, i.e., will not attempt to read a mask block whose maskcnt is greater than the number of mask blocks within the level.

Proof. First, the mask counter for a given level is always incremented whenever a mask block is fetched, such that the next time, the client would be trying to fetch the next mask block. Second, when a real block is fetched from a level ℓ, it is logically removed from the level ℓ. Until the next time the level ℓ is rebuilt, the same block will be found in a level $\ell' < \ell$. Therefore, the next time the client seeks the same real block, it will have been found in a smaller level ℓ', and the client would be reading the next mask block from level ℓ. As mentioned earlier, each level ℓ is rebuilt with $Z' \cdot 2^{\ell}$ mask blocks. We also know that level ℓ will be rebuilt every $Z' \cdot 2^{\ell}$ time steps. Therefore, the mask blocks will not run out before the next rebuild. When the next rebuild happens, the level's mask counter is reset to 0, and all blocks within the level obtain fresh new hkeys, since the hkeys are time-dependent.

Lemma 3 (Bloom filter reads are oblivious). From the perspective of the server, the client reads k independent, (pseudo-)random locations in the Bloom filter every time.

Proof. If a block has been found in a smaller level, the client reads k fresh, random locations in a level's Bloom filter. If the block has not been found in smaller levels, the client reads k real locations computed with a pseudorandom function PRF_{BF}. Assuming security of the PRF_{BF}, we can pretend that these are random locations, and further it is important to observe that these locations have not been disclosed to the server before. In particular, observe that if the client reads k real locations in level ℓ's Bloom filter looking for a specific block idx. This means that block idx resides in a level

$\ell' \geq \ell$. At the end of the present read operation, the block idx will be relocated to a level $\ell^* \geq \ell$. Until the next time level ℓ is rebuilt, block idx will always exist in a smaller level than ℓ. This means that the client will never look for block idx in level ℓ again till the next time level ℓ is rebuilt. As a result, when the client discloses the real Bloom filter locations to the server, these locations cannot have been disclosed before.

Lemma 4 (Offline shuffling is oblivious). The level rebuilding algorithm has deterministic, predictable access patterns.

Proof. Straightforward from the description of the algorithms.

Theorem 1 (Obliviousness of the ORAM scheme). Assume that the symmetric-key encryption scheme used to encrypt data and metadata satisfies semantic security and that PRF_H and PRF_{BF} are secure pseudorandom functions. Then, the ORAM scheme described in this section satisfies semi-honest security.

Proof. Based on the set of lemmas above, it is straightforward to construct an ideal-world simulator for a semi-honest, real-world adversary. Since the symmetric-key encryption scheme is semantically secure, the simulator simulates all ciphertexts by random encryptions of 0 of appropriate length. The time step T and the occupied/empty status of each level is known by the simulator, and the access patterns of the offline shuffling is deterministic and predictable. Therefore, the simulator can easily simulate the offline shuffling. For the online request phase, the simulator simulates by reading k random locations in the Bloom filter in each level. The simulator also discloses the hkey of a random block in this level to the real-world adversary. Each block's hkey is simulated by picking a fresh random string of appropriate length. It is not hard to argue that no polynomial-time environment can distinguish the real- and the ideal-worlds.

3.4.3.1 Malicious Security

Malicious security can be obtained in Bucket ORAM scheme with standard techniques. Most of the data and metadata satisfy predictive time, i.e., the client can efficiently compute the time at which this block (or metadata) was last written to the server during a level rebuild.

With the exception of some metadata, can, for the most part, be used time- and location-aware message authentication codes to achieve malicious security (and there is no need for building a Merkle-hash tree). Upon retrieving a block from the server, the client always verifies the message authentication code, and rejects block if the verification fails.

In Bucket ORAM scheme in this section, block data can be accessed in two modes:

1. By their hkeys during the online phase of the block access algorithm.
2. By their explicit physical addresses on the server (typically containing the level number, the bucket number, and the offset within the bucket) during the offline shuffling phase.

All auxiliary metadata (including the per-level bloom filters and transient metadata created during level rebuilding, not including metadata attached to blocks) are always accessed by their explicit physical addresses. Therefore, below we discuss how to authenticate auxiliary metadata and block data separately.

Authenticating auxiliary metadata. Observe that in our scheme described in this section, all auxiliary metadata (including the per-level bloom filters and transient metadata created during level rebuilding, not including metadata attached to blocks) are accessed by their explicit addresses. Further, all metadata touched during level rebuilding is written to the server via linear scans or oblivious sorting. Both of these operations perform each write to each physical location at public and pre-determined times. Putting the above observations together, the client can attach a time- and location-sensitive message authentication code MAC($sk, T||phys\_addr||metadata$) to every auxiliary metadata chunk, where T is the time metadata was last written to the server.

Authenticating block data. Block data (including most directly attached to the block, such as idx the leaf label) can be accessed by either their hkey or their physical address on the server. Block data are authenticated also using a time- and location-aware MAC:

$$\text{MAC}(sk, T||phys\_addr||block)$$

where T denotes the last time the block was written, and phys_addr typically contains the level number, bucket number, and offset within the bucket. The only subtlety is that when the block is accessed by its hkey, the server needs to additionally return the block's phys_addr to the client, such that the client is able to verify the MAC.

Subtlety. The only exception to this “predictive time” rule is the client invalidating blocks fetched in the online phase, by setting block.type := “dummy”. Therefore, the client can employ a merkle hash tree to authenticate dummy bits attached to blocks.

Corollary. Assuming that the underlying ORAM scheme described in Section 3.4.2 has semi-honest security and that MAC is a secure message authentication scheme, then the augmented scheme with time- and location-aware message authentication codes (as described above) satisfies malicious security.

Proof. The simulator is simulating interactions with a real-world adversary $\mathcal{A}$. For every block (or metadata) the semi-honest simulator intends to send to the semi-

honest real-world adversary, now the simulator interacting with a malicious $\mathcal{A}$ additionally authenticates the block (or metadata) with time- and location-aware message authentication code. Whenever $\mathcal{A}$ returns a block (or metadata), the simulator verifies the message authentication code. If the verification fails, the simulator simply aborts. Effectively, any deviation of from the correct behavior by $\mathcal{A}$ can be detected except with negligible probability. Therefore, any deviation from the correct behavior translates to an aborting attack in the ideal world.

3.5 Ring ORAM

Ring ORAM [11] is considered the most bandwidth-efficient ORAM scheme for the small client storage setting. At the core of the construction is an ORAM that achieves *"bucket-size independent bandwidth"*, which unlocks numerous performance improvements. In practice, Ring ORAM's overall bandwidth is 2.3× to 3× better than the prior-art scheme for small client storage. Further, if memory can perform simple untrusted computation, Ring ORAM achieves constant online bandwidth (∼ 60× improvement over prior art for practical parameters). On the theory side, Ring ORAM features a tighter and significantly simpler analysis than prior art.

3.5.1 The Ring ORAM Protocol

We first describe Ring ORAM in terms of its server and client data structures.

Server storage is organized as a binary tree of buckets where each bucket has a small number of slots to hold blocks. Levels in the tree are numbered from 0 (the root) to L (inclusive, the leaves) where L = O($log\ N$) and N is the number of blocks in the ORAM. Each bucket has $Z + S$ slots and a small amount of metadata. Of these slots, up to Z slots may contain real blocks and the remaining S slots are reserved for dummy blocks.

Client storage is made up of a position map and a stash. The position map is a dictionary that maps each block in the ORAM to a random path in the ORAM tree (each path is uniquely identified by the path's leaf node). The stash buffers blocks that have not been evicted to the ORAM tree and additionally stores $Z(L + 1)$ blocks on the eviction path during an eviction operation. The position map stores N * L bits, but can be squashed to constant storage using the standard recursion technique [4].

Main invariants. Ring ORAM has two main invariants:

1. Every block is mapped to a path chosen uniformly at random in the ORAM tree. If a block α is mapped to path ℓ, block α is contained either in the stash or in some bucket along the path from the root of the tree to leaf ℓ.

2. (**Permuted buckets**) For every bucket in the tree, the physical positions of the $Z + S$ dummy and real blocks in each bucket are randomly permuted with respect to all past and future writes to that bucket.

Since a leaf uniquely determines a path in a binary tree, leaves/paths will be used interchangeably when the context is clear, and denote path ℓ as $\mathcal{P}(\ell)$.

```
Access(op, a, data*)
1: Global/persistent variables: round
2: ℓ' ← UniformRandom (0, 2^L – 1)
3: ℓ ← PositionMap[a]
4: PositionMap[a]← ℓ'
5: data ← ReadPath(ℓ, a)
6: data ← Read block a from S
7: if data := ⊥ then
8:    If block a is not found on path ℓ, it must be in Stash
9:    data ← read and remove a from Stash
10: if op := read then
11:    return data to client
12: if op := write then
13:    data ← data'
14: Stash ← round + 1 mod A
15: if round ≠ 0 then
16:    EvictPath()
17: EarlyReshuffle(ℓ)
```

Figure 14: Ring ORAM Access Algorithm

Access and Eviction Operations. The Ring ORAM access protocol is shown in *Figure 14*. Each access is broken into the following four steps:

1) **Position Map lookup** (lines 3-5): Look up the position map to learn which path ℓ the block being accessed is currently mapped to. Remap that block to a new random path ℓ'. This first step is identical to other tree-based ORAMs [2, 4], but the rest of the protocol differs substantially from previous tree-based schemes.

2) **Read Path** (lines 6-18): The ReadPath(ℓ, α) operation reads all buckets along $\mathcal{P}(\ell)$ to look for the block of interest (block α), and then reads that block into the stash. The block of interest is then updated in stash on a write, or is returned to the client on a read. We remind that both reading and writing a fata block are served by a ReadPath operation. Unlike prior tree-based schemes, this ReadPath operation only reads one block from each backet-the block of interest if found or a previously-unread dummy block otherwise. This is safe because of Invariant 2, above: each bucket is permuted randomly, so the slot being read looks random to an observer. This lowers the bandwidth overhead of ReadPath (i.e., online bandwidth) to L+1 blocks (the number of levels in the tree) or even a single block if the XOR trick is applied.

3) **Evict Path** (Lines 19-22): The EvictPath operation reads Z blocks (all the remaining real blocks, and potentially some dummy blocks) from each bucket

along a path into the stash, and then fills that path with blocks from the stash, trying to push blocks as far down towards the leaves as possible. The sole purpose of an eviction operation is to push blocks back to the ORAM tree to keep the stash occupancy low. Unlike Path ORAM, eviction in Ring ORAM selects paths in the reverse lexico-graphical order, and does not happen on every access [12]. Its rate is controlled by a public parameter A: every A ReadPath operations trigger a single EvictPath operation. This means Ring ORAM needs much fewer eviction operations than Path ORAM.

4) **Early Reshuffles** (Line 23): Finally, EarlyReshuffle on $\mathcal{P}(\ell)$, is performed to the path accessed by ReadPath. This step is crucial in maintaining blocks randomly shuffled in each bucket, which enables ReadPath to securely read only one block from each bucket.

3.5.1.1 Read Path Operation

The ReadPath operation is shown in *Figure 15*. For each bucket along the current path, ReadPath selects a single block to read from that bucket. For a given bucket, if the block of interest lives in that bucket, we read and invalidate the block of interest. Otherwise, we read and invalidate a randomly-chosen dummy block that is still valid at that point. The index of the block to read (either real or random) is returned by the GetBlockOffset. Reading a single block per bucket is crucial for bandwidth improvements. In addition to reducing online bandwidth by a factor of Z, it allows to use larger Z and A to decrease overall bandwidth. Without this, read bandwidth is proportional to Z, and the cost of larger Z on reads outweighs the benefits.

Bucket Metadata. Because the position map only tracks the path containing the block of interest, the client does not know where in each bucket to look for the block of interest. Thus, for each bucket we must store the permutation in the bucket metadata that maps each real block in the bucket to one of the $Z + S$ slots (Lines 3, GetBlockO set) as well as some additional metadata. Once we know the offset into the bucket, Line 4 reads the block in the slot, and invalidates it. We should mention that the metadata is small and independent of the block size. One important piece of metadata to mention now is a counter which tracks how many times it has been read since its last eviction (Line 8). If a bucket is read too many (S) times, it may run out of dummy blocks (i.e., all the dummy blocks have been invalidated). On future accesses, if additional dummy blocks are requested from this bucket, we cannot reread a previously invalidated dummy block: doing so reveals to the adversary that the block of interest is not in this bucket. Therefore, we need to reshuffle single buckets on-demand as soon as they are touched more than S times using EarlyReshuffle.

XOR Technique. During ReadPath operation, each block returned to the client is a dummy block except for the block of interest. This means Ring ORAM scheme can also

take advantage of the XOR technique introduced in [13] to reduce online bandwidth overhead to O(1). To be more concrete, on each access ReadPath returns L + 1 blocks in ciphertext, one from each bucket, Enc(b_0, r_0), Enc(b_1, r_1), …, Enc(b_L, r_L). **Enc** is a randomized symmetric scheme such as AES counter mode with nonce r_i. With the XOR

```
ReadPAth(ℓ, a)
1: data ← ⊥
2: for i = 0 to L do
3:    offset ← GetBlockOffset(P(ℓ, i), a)
4:    data' ← P(ℓ, i, offset)
5:    Invalidate P(ℓ, i, offset)
6:    if data' ≠ ⊥ then
7:       data ← data'
8:    P(ℓ, i).count ← P(ℓ, i).count + 1
9: return data
```

technique, ReadPath will return a *single ciphertext* – the ciphertext of all the blocks XORed together, namely Enc(b_0, r_0) ⊕ Enc(b_1, r_1) ⊕ … ⊕ Enc(b_L, r_L). The client can recover the encrypted block of interest by XORing the returned ciphertext with the encryptions of all the dummy blocks. To make computing each dummy block's encryption easy, the client can set the plaintext of all dummy blocks to a fixed value of its choosing (e.g., 0).

Figure 15: Ring ORAM ReadPath Algorithm

3.5.1.2 Evict Path Operation

The EvictPath routine is shown in *Figure 16*. As mentioned, evictions are scheduled statically: one eviction operation happens after every A reads. At a high level, an eviction operation reads all remaining real blocks on a path (in a secure fashion), and tries to push them down that path as far as possible. The leaf-to-root order in the writeback step (Lines 8) reflects that we wish to fill the deepest buckets as fully as possible (EvictPath is like a Path ORAM access where no block is accessed and therefore no block is remapped to a new path). We emphasize two unique features of Ring ORAM eviction operations. First, evictions in Ring ORAM are performed to paths in a specific order called the *reverse-lexicographic* order, first proposed by Gentry et al. [12]. The reverse-lexicographic order eviction aims to minimize the overlap between consecutive eviction paths, because (intuitively) evictions to the same bucket in consecutive accesses are less useful. This improves eviction quality and allows to reduce the frequency of eviction. Second, buckets in Ring ORAM need

```
EvictPAth()
1: Global/persistent variable G initialized to 0
2: ℓ ← G mod 2^L
3: G← G + 1
4: for i = 0 to L do
5:    Stash ← StashUReadBucket(P(ℓ, i))
6: for i = L to 0 do
7:    WriteBucket(P(ℓ, i), Stash)
8:    P(ℓ, i).count ← 0
```

to be randomly shuffled (Invariant 2), and we mostly rely on EvictPath operations to keep them shuffled. An EvictPath operation reads Z blocks from each bucket on a path into the stash, and writes out $Z + S$ blocks (only up to Z are real blocks) to each bucket, randomly permuted.

Figure 16: Ring ORAM EvictPAth Algorithm

3.5.1.3 Early Reshuffle Operation

Due to randomness, a bucket can be touched > S times by ReadPath operations before it is reshuffled by the scheduled EvictPath. If this happens, we call EarlyReshuffle on that bucket to reshuffle it before the bucket is read again. More precisely, after each ORAM access EarlyReshuffle goes over all the buckets on the read path, and reshuffles all the buckets that have been accessed more than S times by performing *ReadBucket* and *WriteBucket*. *ReadBucket* and *WriteBucket* are the same as in EvictPath: that is, *ReadBucket* reads exactly Z slots in the bucket and *WriteBucket* re-permutes and writes back $Z + S$ real/dummy blocks. Though S does not affect security, it clearly has an impact on performance (how often we shuffle, the extra cost per reshuffle, etc.).

3.5.1.4 Bucket Structure

We would like to make two remarks. First, only the *data* fields are permuted and that permutation is stored in *ptrs*. Other bucket fields do not need to be permuted because when they are needed, they will be read in their entirety. Second, *count* and *valids* are stored in plaintext. There is no need to encrypt them since the server can see which bucket is accessed (deducing count for each bucket), and which slot is accessed in each bucket (deducing *valids* for each bucket). In fact, if the server can do computation and is trusted to follow the protocol faithfully, the client can let the server update *count* and *valids*. All the other structures should be probabilistically encrypted.

Having defined the bucket structure, we can be more specific about some of the operations in earlier sections. For example, in Algorithm 2 Line 5 means reading $\mathcal{P}(\ell$,i).data[offset], and Line 6 means setting $\mathcal{P}(\ell$,i).valids[offset] to 0.

Now, we describe the helper functions in detail. GetBlockOffset reads in the *valids*, *addrs*, *ptrs* fields, and looks for the block of interest. If it finds the block of interest, meaning that the address of a still valid block matches the block of interest, it returns the permuted location of that block (stored in *ptrs*). If it does not find the block of interest, it returns the permuted location of a random valid dummy block.

ReadBucket reads all of the remaining real blocks in a bucket into the stash. For security reasons, ReadBucket always reads exactly Z blocks from that bucket. If the bucket contains less than Z valid real blocks, the remaining blocks read out are

random valid dummy blocks. Importantly, since we allow at most S reads to each bucket before reshuffling it, it is guaranteed that there are at least Z valid (real + dummy) blocks left that have not been touched since the last reshuffle.

WriteBucket evicts as many blocks as possible (up to Z) from the stash to a certain bucket. If there are z' $\leq$ Z real blocks to be evicted to that bucket, $Z + S$ – z' dummy blocks are added. The $Z + S$ blocks are then randomly shuffled based on either a truly random permutation or a Pseudo Random Permutation (PRP). The permutation is stored in the bucket field *ptrs*. Then, the function resets count to 0 and all valid bits to 1, since this bucket has just been reshuffled and no blocks have been touched. Finally, the permuted data field along with its metadata are encrypted (except *count* and *valids*) and written out to the bucket.

3.5.2 Security Analysis

Claim 1. ReadPath *leaks no information*. The path selected for reading will look random to any adversary due to Invariant 1 (leaves are chosen uniformly at random). From Invariant 2, we know that every bucket is randomly shuffled. Moreover, because we invalidate any block we read, we will never read the same slot. Thus, any sequence of reads (real or dummy) to a bucket between two shuffles is indistinguishable. Thus the adversary learns nothing during ReadPath.

Claim 2. EvictPath *leaks no information*. The path selected for eviction is chosen statically, and is public (*reverse-lexicographic* order). *ReadBucket* always reads exactly Z blocks from random slots. *WriteBucket* similarly writes $Z + S$ encrypted blocks in a data-independent fashion.

Claim 3. EarlyShuflle *leaks no information*. To which buckets EarlyShuffle operations occur is publicly known: the adversary knows how many times a bucket has been accessed since the last EvictPath to that bucket. *ReadBucket* and *WriteBucket* are secure as per observations in Claim 2.

The three subroutines of the Ring ORAM algorithm are the only operations that cause externally observable behaviors. Claims 1, 2, and 3 show that the subroutines are secure. We have so far assumed that path remapping and bucket permutation are truly random, which gives unconditional security. If pseudorandom numbers are used instead, we have computational security through similar arguments.

3.6 Comparison

3.6.1 Path ORAM vs Optimization of Path Oblivious RAM in Secure Processors

Path ORAM overhead drops by 41.8%, and SPEC benchmark execution time improves by 52.4% in relation to a baseline configuration due to the optimizations that

presented at [7]. In [7], they used DRAMSim2 [54] to simulate ORAM performance on commodity DRAM, assuming that the data ORAM is 8 GB with 50% utilization (resulting in 4 GB working set); position map ORAMs combined are less than 1 GB. So it was considered a 16 GB DRAM. DRAMSim2's default DDR3_micron configuration was used with 16-bit device width, 1024 columns per row in 8 banks, and 16384 rows per DRAM-chip. So the size of a node in the 2k-ary tree was $ch \times 128 \times 64$ bytes, where ch is the number of independent channels. The evaluation was between baseORAM and the 4 best configuration with the Overhead breakdown for 8 GB hierarchical ORAMs with 4 GB working set based on Figure 17Figure 17: Overhead breakdown for 8 GB hierarchical ORAMs with 4 GB working set. DZ3Pb12 means data ORAM uses Z=3 and position map ORAMs have 12-byte block. The final position map is smaller than 200 KB.

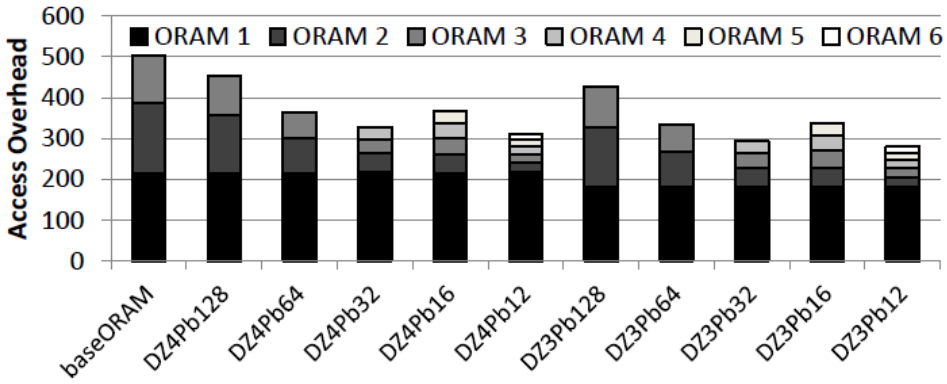

Figure 17: Overhead breakdown for 8 GB hierarchical ORAMs with 4 GB working set

Figure 18: Hierarchical ORAM latency in DRAM cycles assuming 1/2/4 channel(s)Figure 18 shows the data latency (not counting decryption latency) of hierarchical Path ORAMs using the naïve memory placement and the subtree strategy of Section 3.2, and compares these with the theoretical value, which assumes DRAM always works at its peak bandwidth. The figure shows that ORAM can benefit from multiple independent channels, because each ORAM access is turned into hundreds of DRAM accesses. But this also brings the challenge of how to keep all the independent channels busy. On average, the naïve scheme's performance becomes 20% worse than the theoretical result when there are two independent channels and 60% worse when there are four. The subtree memory placement strategy is only 6% worse than the theoretical value with two channels and 13% worse with four. The remaining overhead comes from the few row buffer misses and DRAM refresh. Even though a 12-byte position map ORAM block size has lower theoretical overheads, it is worse than the 32-byte design. We remark that it is hard to define Path ORAM's slowdown over DRAM. On one hand, DDR3 imposes a minimum ~26 (DRAM) cycles per access, making Path ORAM's latency ~30× over DRAM assuming 4 channels. On the other hand, the Path ORAM that presented to Section 3.2 consumes almost the entire bandwidth of all channels.

Its effective throughput is hundreds of times lower than DRAM's peak bandwidth (≈ access overhead). But the actual bandwidth of DRAM in real systems varies greatly and depends heavily on the applications, making the comparison harder.

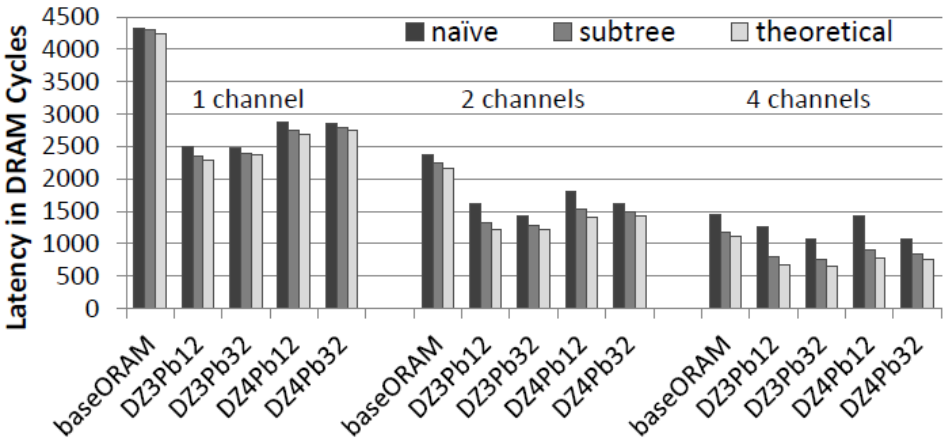

Figure 18: Hierarchical ORAM latency in DRAM cycles assuming 1/2/4 channel(s)

In order to evaluate the SPEC benchmark performance Path ORAM was connected to a processor and evaluated the optimizations over a subset of the SPEC06-int benchmarks. The processors are modeled with a cycle-level simulator based on the public domain SESC [55] simulator that uses the MIPS ISA. Instruction/memory address traces are first generated through SESC's rabbit (fast forward) mode and then fed into a timing model. Each experiment uses SPEC reference inputs, fast-forwards 1 billion instructions to get out of initialization code and then monitors performance for 3 billion instructions. It was also compared against a conventional processor that uses DRAM. Path ORAMs and DRAMs are both simulated using DRAMSim2. Figure 19 shows the SPEC benchmark running time using different Path ORAM configurations and super blocks, normalized to the insecure processor with DRAM. DZ3Pb32 reduces the average execution time by 43.9% compared with the baseline ORAM. As expected, the performance improvement is most significant on memory bound benchmarks (mcf, bzip2 and libquantum).

In that experiment, used only super blocks of size two (consisting of two blocks). On average, DZ4Pb32 with super blocks outperforms DZ3Pb32 without super blocks (the best configuration without super blocks) by 5.9%, and is 52.4% better than the baseline ORAM. There is a substantial performance gain on applications with good spatial locality (e.g., mcf) where the prefetched block is likely to be accessed subsequently. Using static super blocks with DZ3Pb32 slightly improves the performance on most benchmarks, but has worse performance on certain benchmarks because it requires too many dummy accesses, canceling the performance gain on average.

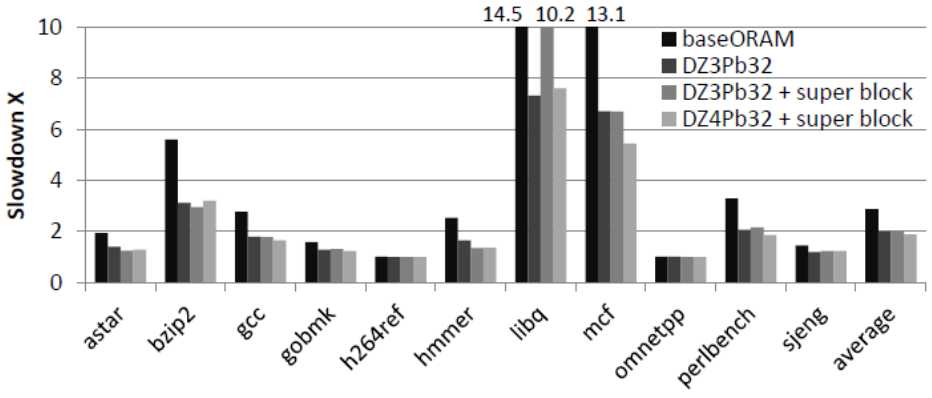

Figure 19: SPEC benchmark performance

3.6.2 Path ORAM vs Circuit ORAM

Circuit size. In *Table 1*, the circuit sizes is compared for Circuit ORAM and Path ORAM scheme. Results in this table are obtained for a 4GB dataset with the following concrete parameters: $N = 230, D = 32 bits\ and\ security\ failure\ \delta = 2^{-80}$, "Rand." stands for randomly chose eviction paths; "Det." stands for eviction with reverse-lexicographical-ordered paths. For Path ORAM [2], considered a naive implementation. For both schemes, considered two strategies for choosing the eviction path: random-order eviction and deterministic-order eviction (based on digit-reversed lexicographic order [12]). The table shows that Circuit ORAM results in 8.2× to 48.6× smaller circuit size than Path ORAM. Circuit ORAM's speedup will become even bigger when the total data size N is greater.

Type Of ORAM	Circuit ORAM		Path ORAM(naive)	
	Det.	Rand.	Det.	Rand.
Circuit size	$3.5M$	$6.6M$	$28.5M$	$170.1M$
Relative overhead	$1X$	$1.9X$	$8.2X$	$48.6X$
#AND gates	$0.97M$	$1.6M$	$9.5M$	$56.6M$
Relative overhead	$1X$	$1.6X$	$9.79X$	$58.4X$

Table 1: Comparison of Circuit ORAM and Path ORAM

Circuit ORAM also outperforms previous schemes in terms of bandwidth costs and number of accesses under wide parameter ranges. Circuit ORAM achieves the following bandwidth cost and number of accesses for a negl(N) statistical failure probability.

- Suppose the position map levels adopt a block size $D' = O(logN)$, then Circuit ORAM achieves $O(DlogN + log^3 N)\omega(1)$ bandwidth cost

and$O(log^2N)$ number of accesses. In particular, for a $D = \Omega(log^2N)$ block size, the bandwidth cost is $O(DlogN)\omega(1)$.

- Suppose all levels adopt a uniform block size of $D = \chi\, logN$, then Circuit ORAM achieves $O\big(DlogNlog_\chi N\big)\omega(1)$ bandwidth cost, and $O\big(logNlog_\chi N\big)\omega(10)$ number of accesses. Of particular interest is when $D = N^\varepsilon$ for some constant $0 < \varepsilon < 1$. In this case, Circuit ORAM achieves $O(DlogN)\omega(1)$ bandwidth cost, and $O(logN)\omega(1)$ number of accesses.

Scheme	Amortized Bandwidth Cost	Client Storage
	Tree-based ORAMs	
Binary-tree ORAM	$O(D\log^2 N + \log^4 N)\cdot\omega(1)$	$O(D)$
Path ORAM(naive circuit)	$O(D\log N + \log^3 N)$	$O(D\log N)\cdot\omega(1)$
Circuit ORAM	$\mathbf{O(D\log N + \log^3 N)\cdot\omega(1)}$	$\mathbf{O(D)}$

Table 2: Comparison of Circuit ORAM, Path ORAM and Binary-tree ORAM

3.6.3 Path ORAM vs Bucket ORAM

The cost of Oblivious RAM constructions can be characterized by several related but different metrics. Two of the most important metrics are:

- **Bandwidth blowup.** For the client to access a single block, how many blocks on average must be transmitted between the client and the server to hide the true block of intent. This metric accounts for both online and offline traffic, and hence is also referred to as overall bandwidth blowup
- **Response time or latency**. The minimum delay from a client's request till the block is retrieved

Bucket ORAM scheme gives a positive answer to the above challenge. Bucket ORAM + AHE (with the use of additively homomorphic encryption) achieves a single online roundtrip, and $O(1)$ bandwidth blowup. Without additively homomorphic encryption, Bucket ORAM has $\tilde{O}(logN)$ bandwidth blowup. We refer the reader to *Table 3* for a more detailed comparison with ORAM schemes. $\tilde{O}and\ \tilde{\Omega}$ hides $loglogN$ to poly($loglogN$) factors. Asymptotical costs listed here are for malicious security. The security of all schemes are parameterized to have negligible in N failure probability. Server I/O counts the amortized number of blocks the server touches per access.

Construction	Latency (RTs)	Client store (blocks)	BW blowup	Server I/O (blocks)	Block size (bits)
Path ORAM	$O(\log N)$	$O(\log N)\omega(1)$	$O(\log N)$	$O(\log N)$	$\Omega(\log^2 N)$
Bucket ORAM	1	$O(1)$	$\widetilde{O}(\log N)$	$\widetilde{O}(\log N)$	$\omega(\frac{\log^3 N}{\log \log N})$
Bucket ORAM + AHE	1	$O(1)$	$O(1)$	$O(\log N)$	$\widetilde{\Omega}(\log^6 N)$

Table 3: Comparison of Bucket ORAM and Path ORAM

3.6.4 Path ORAM vs Ring ORAM

In this section, it is shown how Ring ORAM improves the performance of secure processors over Path ORAM. It was evaluated a 4 GB ORAM with 64-Byte block size (matching a typical processor's cache line size). Due to the small block size, Ring ORAM had the parameters $Z = 5; A = 5; X = 2$ to reduce metadata overhead. Recursion was applied three times with 32-Byte position map block size and get a 256 KB final position map. It was evaluated the performance for SPEC-int benchmarks and two database benchmarks, and simulate 3 billion instructions for each benchmark. Assuming a flat 50-cycle DRAM latency, and compute ORAM latency with 128 bits/cycle processor memory bandwidth. It was not used a tree-top caching since it proportionally benefits both Ring ORAM and Path ORAM. Today's DRAM DIMMs cannot perform any computation, but it is not hard to imagine having simple XOR logic either inside memory, or connected to O(logN) parallel DIMMs so as not to occupy processor memory bandwidth. Thus, it is shown results with and without the XOR technique. Figure 20 shows program slowdown over an insecure DRAM. The high order bit is that using Ring ORAM with XOR results in a geometric average slowdown of 2.8× relative to an insecure system. This is a 1.5× improvement over Path ORAM. If XOR is not available, the slowdown over an insecure system is 3.2×. The experiment had also repeated with the unified ORAM recursion technique. The geometric average slowdown over an insecure system is 2.4× (2.5× without XOR).

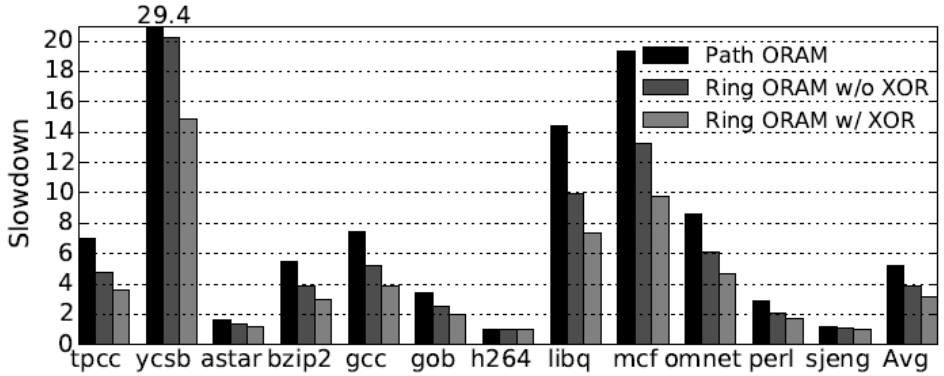

Figure 20: SPEC benchmark slowdown

4 Constant worst-case bandwidth blowup

4.1 Onion ORAM

Onion ORAM [14] is the first ORAM with constant worst-case bandwidth blowup under standard cryptographic assumptions. Onion ORAM leverages poly-logarithmic server computation to circumvent the logarithmic lower bound on ORAM bandwidth blowup. Unlike prior work, Onion ORAM does not require fully homomorphic encryption, but re-quires only certain additively homomorphic encryption schemes. Onion ORAM utilizes novel techniques to achieve security against a malicious server, without resorting to expensive and non-standard techniques.

4.1.1 Overview of Techniques

In Onion ORAM schemes, the client “guides" the server to perform ORAM accesses and evictions homomorphically by sending the server some “helper values". With these helper values, the server's main job will be to run a sub-routine called the “homomorphic select" operation (select operation for short), which can be implemented using either AHE or SWHE. We can achieve constant bandwidth blowup because helper value size is independent of data block size: when the block size sufficiently large, sending helper values does not affect the asymptotic bandwidth blowup. We now explain these ideas along with pitfalls and solutions in more detail.

Building block: homomorphic select operation. The select operation, which resembles techniques from private information retrieval (PIR) [15], takes as input m plaintext data blocks pt_1, …,pt_m and encrypted helper values which represent a user-chosen index i*. The output is an encryption of block pt_{i*}. Obviously, the helper values should not reveal i*.

All ORAM operations can be implemented using homomorphic select operations. In Onion scheme, for each ORAM operation, the client read/writes per-block metadata and creates a select vector(s) based on that metadata. The client then sends the encrypted select vector(s) to the server, who does the heavy work of performing actual computation over block contents.

Specifically, on top of tree-based ORAMs [2, 4] will be built a standard type of ORAM without server computation. Metadata for each block includes its logical address and the path it is mapped to. To request a data block, the client first reads the logic addresses of all blocks along the read path. After this step, the client knows which block to select and can run the homomorphic select protocol with the server. ORAM eviction operations require that the client sends encrypted select vectors to indicate how blocks should percolate down the ORAM tree. As explained above, each select operation adds an encryption layer to the selected block.

Achieving constant bandwidth blowup. To get constant bandwidth blowup, we must ensure that select vector bandwidth is smaller than the data block size. For this, we need several techniques. First, we will split each plaintext data block into $\mathcal{C}$ chunks pt_i = (pt_i[1], …, pt_i[$\mathcal{C}$]), where each chunk is encrypted separately, i.e., ct_i = (ct_i[1], …, ct_i[$\mathcal{C}$]) where ct_i[j] is an encryption of pt_i[j]. Crucially, each select vector can be reused for all the $\mathcal{C}$ chunks. By increasing $\mathcal{C}$, we can increase the data block size to decrease the relative bandwidth of select vectors.

Second, we require that each encryption layer adds a small additive ciphertext expansion (even a constant multiplicative expansion would be too large). Fortunately, we do have well established additively homomorphic encryption schemes that meet this requirement, such as the Damgård–Jurik cryptosystem [16]. Third, the “depth" of the homomorphic select operations has to be bounded and shallow.

Bounding the select operation depth. This issue addressed in [14] by constructing a new tree-based ORAM, which is called “bounded feedback ORAM". By “feedback", we refer to the situation where during an eviction some block α gets stuck in its current bucket b. When this happens, an eviction into b needs select operations that take both incoming blocks and block α as input, resulting in an extra layer on bucket b (on top of the layers bucket b already has). The result is that buckets will accumulate layers (with AHE) on each eviction, which grows unbounded over time.

This bounded feedback ORAM breaks the feedback loop by guaranteeing that bucket b will be empty at public times, which allows upstream blocks to move into b without feedback from blocks already in b. It turns out that breaking this feedback is not trivial: in all existing tree-based ORAM schemes [4, 2, 8], blocks can get stuck in buckets during evictions which means there is no guarantee on when buckets are empty.

Techniques for malicious security. The main idea is to rely on probabilistic checking, and to leverage an error-correcting code to amplify the probability of detection. As mentioned before, each block is divided into $\mathcal{C}$ chunks. We will have the client randomly sample security parameter $\lambda \ll \mathcal{C}$ chunks per block (the same random choice for all blocks), referred to as *verification chunks*, and use standard memory checking to ensure their authenticity and freshness. On each step, the server will perform homomorphic select operations on all $\mathcal{C}$ chunks in a block, and the client will perform the same homomorphic select operations on the λ verification chunks. In this way, whenever the server returns the client some encrypted block, the client can check whether the corresponding chunks match the verification chunks.

Unfortunately, the above scheme does not guarantee negligible failure of detection. For example, the server can simply tamper with a random chunk and hope that it's not one of the verification chunks. Clearly, the server succeeds with non-

negligible probability. The fix is to leverage an error-correcting code to encode the original $\mathcal{C}$ chunks of each block into $\mathcal{C}''$ = 2 $\mathcal{C}$ chunks, and ensure that as long as $\frac{3}{4}$ $\mathcal{C}''$ chunks are correct, the block can be correctly decoded. Therefore, the server knows a priori that it will have to tamper with at least $\frac{1}{4}$ $\mathcal{C}''$ chunks to cause any damage at all, in which case it will get caught except with negligible probability.

4.1.2 Onion ORAM Protocol (Additively Homomorphic Encryption)

In this section, we describe how to leverage an AHE scheme with additive ciphertext expansion to transform the bounded feedback ORAM into a semi-honest secure Onion ORAM scheme. Recall that each block is tagged with the following metadata: the block's logical address and the leaf it is mapped to, and that the size of the metadata is independent of the block size.

Initialization. The client runs a key generation routine for all layers of encryption, and gives all public keys to the server.

Read Path. ReadPath(ℓ, a) *Figure 21* can be done with the following steps:

1. Client downloads and decrypts the addresses of all blocks on path l, locates the block of interest a, and creates a corresponding select vector $\vec{b} \in \{0,1\}^{Z(L+1)}$.
2. Client and server run the homomorphic select sub-protocol with client's input being encryptions of each element in $\vec{b}$ and server's input being all encrypted blocks on path l. The outcome of the sub-protocol block a is sent to the client.
3. Client re-encrypts and writes back the addresses of all blocks on path l, with block a now invalidated. This removes block a from the path without revealing its location. Then, the client re-encrypts block a (possibly modified) under 1 layer, and appends it to the root bucket.

Eviction. To perform EvictAlongPath(ℓ_e), do the following for each level k from 0 to L–1:

1. Client downloads all the metadata (addresses and leaf labels) of the bucket triplet. Based on the metadata, the client determines each block's location after the bucket-triplet eviction.
2. For each slot to be written in the two child buckets:
    a. Client creates a corresponding select vector $\vec{b} \in \{0,1\}^{2Z}$.
    b. Client and server run the homomorphic select sub-protocol with the client's input being encryptions of each element in $\vec{b}$, and the server's input being the child bucket (being written to) and its parent bucket. Note that if the child bucket is empty (which is public information to the server), it conceptually has zero encryption layers.

c. Server overwrites the slot with the outcome of the homomorphic select sub-protocol.

ReadPAth(ℓ, a)

1: Read all blocks on path $\mathcal{P}(\ell)$
2: Select and return the block with address a
3: Invalidate the block with address a

Figure 21: Onion ORAM ReadPath Algorithm

EvictAlongPath (ℓ_e)

1: **for** k = 0 to L – 1 **do**
2: Read **all** blocks in $\mathcal{P}(\ell_e, k)$ and its two children
3: Move **all** blocks in $\mathcal{P}(\ell_e, k)$ and its two children
4: $\mathcal{P}(\ell_e, k)$ is empty at this point
5: **end for**

Figure 22: Onion ORAM EvictAlongPath Algorithm

4.1.3 Security Analysis

We now show that the scheme is secure against a fully malicious server who can deviate arbitrarily from the protocol. We start by describing several abstract properties of the Onion ORAM scheme from the previous section. We will call any server computation ORAM scheme satisfying these properties an abstract server computation ORAM.

Data blocks and metadata. The server storage consists of two types of data: data blocks and metadata. The server performs computation on data blocks, but never on metadata. The client reads and writes the metadata directly, so the metadata can be encrypted under any semantically secure encryption scheme.

Operations on data blocks. Each plaintext data block is divided into $\mathcal{C}$ chunks, and each chunk is separately encrypted $ct_i = (ct_i[1], \ldots, ct_i[\mathcal{C}])$. The client operates on the data blocks either by: (1) directly reading/writing an encrypted data block, or (2) instructing the server to apply a function f to form a new data block ct_i, where ct_i[j] only depends on the j-th chunk of other data blocks, i.e., $ct_i[j] = f(ct_1[j], \ldots, ct_m[j])$ for all $j \in [1..C]$. It is easy to check that the Onion ORAM scheme is instance of the above abstraction. The metadata consists of the encrypted addresses and leaf labels of each data block, as well as additional space needed to implement ORAM recursion. The data blocks are encrypted under a layered AHE scheme. Function f is a “homomorphic select operation", and is applied to each chunk.

We now describe a generic compiler that takes any “abstract server computation ORAM" that satisfies honest-but-curious security and compiles it into a “verified server computation ORAM" which is secure in the fully malicious setting.

Verifying metadata. We can use standard “memory checking" [17] schemes based on Merkle trees [18] to ensure that the client always gets the correct metadata, or aborts if the malicious server ever sends an incorrect value. A generic use of a Merkle tree would add an O($log\ N$) multiplicative overhead to the process of accessing metadata [19], which is good enough. This O($log\ N$) Integrity verification for Path Oblivious overhead can also be avoided by aligning the Merkle tree with the ORAM tree [20], or using generic authenticated data structures [21]. In any case, verifying metadata is basically free in Onion ORAM.

Challenge of verifying data blocks. Unfortunately, we cannot rely on standard memory checking to protect the encrypted data blocks when the client doesn't read/write them directly but rather instructs the server to compute on them. The problem is that a malicious server that learns some information about the client's access pattern based on whether the client aborts or not. Consider Onion ORAM for example. The malicious server wants to learn if, during the homomorphic select operation of a ORAM request, the location being selected is i. The server can perform the operation correctly except that it would replace the ciphertext at position i with some incorrect value. In this case, if the location being selected was indeed i then the client will abort since the data it receives will be incorrect, but otherwise the client will accept. This violates ORAM's privacy requirement. A more general way to see the problem is to notice that the client's abort decision above depends on the decrypted value, which depends on the secret key of the homomorphic encryption scheme. Therefore, we can no longer rely on the semantic security of the encryption scheme if the abort decision is revealed to the server. To fix this problem, we need to ensure that the client's abort decision only depends on ciphertext and not on the plaintext data.

Verifying data blocks. The solution is as follows, the client selects a random subset V consisting of λ chunk positions. This set V is kept secret from the server. The subset of chunks in positions {j : j ∈ V} of every encrypted data block are treated as additional metadata, which called the “verification chunks". Verification chunks are encrypted and memory checked in the same way as the other metadata. Whenever the client instructs the server to update an encrypted data block, the client performs the same operation himself on the verification chunks. Then, when the client reads an encrypted data block from the server, he can check the chunks in V against the ciphertexts of verification chunks. This check ensures that the server cannot modify too many chunks without getting caught. To ensure that this check is sufficient, it is applied an error-correcting code which guarantees that the server has to modify a large fraction of chunks to affect the plaintext. In more detail:

- Every plaintext data block pt = (pt [1], …, pt [$\mathcal{C}$]) is first encoded via an error-correcting code into a codeword block pt_ecc = ECC(pt) = (pt_ecc [1], …,

pt_ecc[$\mathcal{C}'$]). The error-correcting code ECC has a rate $\mathcal{C}/\mathcal{C}'$= a < 1and can efficiently recover the plaintext block if at most a δ-fraction of the codeword chunks are erroneous. For concreteness, we can use a Reed-Solomon code, and set α = $\frac{1}{2}$, δ = (1 – α)/2 = $\frac{1}{4}$. The client then uses the "abstract server computation ORAM" over the codeword blocks pt_ecc(instead of pt).

- During initialization, the client selects a secret random set V = {v_1, …, v_λ} ⊆ [$\mathcal{C}'$]. Each ciphertext data block ct_i has verification chunks $verCh_i$ = ($verCh_i$[1], …, $verCh_i$[λ]). We ensure the invariant that, during an honest execution, $verCh_i$[j] = ct_i[s_j] for j ∈ [1…λ].
- The client uses a memory checking scheme to ensure the authenticity and freshness of the metadata including the verification chunks. If the client detects a violation in metadata at any point, the client aborts (it is called $abort_0$).
- Whenever the client directly updates or instructs the server to apply the aforementioned function f on an encrypted data block ct_i, it also updates or applies the same function f on the corresponding verification chunks $verCh_i$[j] for j ∈ [1…λ], which possibly involves reading other verification chunks that are input to f.
- When the client reads an encrypted data block ct_i, it also reads $verCh_i$ and checks that $verCh_i$ [j] = ct_i[s_j] for j ∈ [1…λ] and aborts if this is not the case (it is called $abort_1$). Otherwise the client decrypts ct_i to get pt_ecc$_i$ and performs error-correction to recover pt_i. If the error-correction fails, the client aborts (it is called $abort_2$)

If the client ever aborts during any operation with abort0; abort1 or abort2, it refuses to perform any future operations.

Security Intuition. Notice that in the above scheme, the decision whether $abort_1$ occurs does not depend on any secret state of the abstract server computation ORAM scheme and therefore can be revealed to the server without sacrificing privacy. We will argue that, if $abort_1$ does not occur, then the client retrieves the correct data (so $abort_2$ will not occur) with overwhelming probability. Intuitively, the only way that a malicious server can cause the client to either retrieve the incorrect data or trigger $abort_2$ without triggering $abort_1$ is to modify at least a (by default, δ = 1/4) fraction of the chunks in an encrypted data block, but avoid modifying any of the λ chunks corresponding to the secret set V. This happens with probability at most $(1 – \delta)^\lambda$ over the random choice of V, which is negligible.

4.2 C – ORAM

Onion ORAM uses homomorphic encryption to increase the efficiency of Oblivious RAM protocols and achieves O(1) communication overhead with polylogarithmic

server computation. However, it has two drawbacks. It requires a large block size of B = Ω(log^6 N) with large constants. Moreover, while it only needs polylogarithmic computation complexity, that computation consists mostly of expensive homomorphic multiplications. C – ORAM [22] addresses these problems and reduces the required block size to Ω(log^4 N). The main idea is to replace Onion ORAM homomorphic eviction routine with a new, much cheaper permute-and-merge eviction which eliminates homomorphic multiplications and maintains the same level of security. In turn, this removes the need for layered encryption that Onion ORAM relies on and reduces both the minimum block size and server computation.

4.2.1 Overview of C – ORAM

To achieve the increased efficiency and lower block size, in C – ORAM [22] presented a novel, efficient, oblivious bucket merging technique for Onion ORAM that replaces its expensive layered encryption. Bucket merging is applied during ORAM eviction. The content of a parent node/bucket and its child node/bucket can be merged obliviously, i.e., the server does not learn any information about the load of each bucket. The idea is that the client sends a permutation Π to the server. Using this permutation, the server aligns the individual encrypted blocks of the two buckets and merges them into a destination bucket. The client chooses the permutation such that blocks containing real data in one bucket are always aligned to empty blocks in the other bucket. As each block is encrypted with additively homomorphic encryption, merging two blocks is a simple addition of ciphertexts. For the server, merging is oblivious, because, informally, any permutation Π from the client is indistinguishable from a randomly chosen permutation. For buckets of size O(z), oblivious merging evicts elements from a parent bucket to its child with O(z log z) bits of communication instead of O(γz^2) of Onion ORAM. As a result of applying this merging technique, it is only needed a constant number of PIR reads and writes for ORAM operations. As a warm up, in [22] is presented a technique allowing amortized constant communication complexity with a smaller block size B in Ω(z log z log N + γz log N). Additionally the second and main technique achieves constant worst case communication complexity with smaller block size in Ω(z log z log N + γz).

4.2.1.1 Oblivious Merging

Oblivious merging is a technique that obliviously lines up two buckets in a specific order and merges them into one bucket. Using this technique, real data elements can be evicted from a bucket to another by permuting the order of blocks of one of them and then adding additively homomorphically encrypted blocks. Oblivious merging is based on an oblivious permutation generation that takes as input the configurations of two buckets and outputs a permutation Π. A configuration of a bucket specifies which of the blocks in the bucket are real blocks and which are empty. Permutation

Π arranges blocks in such a way that there are no real data elements at the same position in the two blocks.

C-ORAM keeps Onion ORAM's main construction. That is, C – ORAM is a tree-based ORAM composed of a main tree ORAM storing the actual data and a recursive ORAM storing the position map. The position map consists of a number of ORAM trees with linearly increasing height mapping a given address to a tag. For n elements stored in the ORAM, the communication needed to access the position map is in O($log^2 N$). As with all recent tree-based ORAMS, the recursive position map's communication complexity is dominated by the block size. Let N be a power of 2. C-ORAM is a binary tree with L levels and 2L leave nodes. Each node/bucket contains $\mu \cdot z$ blocks. Here, z is the number of slots needed to hold blocks as in Onion ORAM and μ is a multiplicative constant that gives extra room in the buckets for noisy blocks, which is important for C – ORAM construction. We maintain the same relation between N, L and z as in Onion ORAM, namely $N \leq z \cdot 2^{L-1}$. Each block in a C-ORAM bucket is encrypted using an additively homomorphic encryption, e.g., Pailler's or Damgard-Jurik's cryptosystem. Also each bucket contains IND-CPA encrypted meta-information, headers, containing additional information about a bucket's contents.

4.2.1.2 Headers

Bucket headers are an important component in C – ORAM as they determine how oblivious permutations are generated. A bucket header is comprised of two parts: the first part stores for each block whether it is noisy, contains real data or is empty. The second part stores the block tags. More formally, the header is composed of two vectors $header_1$ and $header_2$. Vector $header_1$ has length $\mu \cdot z$, and each element is either noisy, empty or real. Thus, each element has a size of two bits. The total size of this vector is in O (μz). $header_2$ is a ($\mu \cdot z \times log\ N$) binary matrix. The rows represent the address of the blocks. Finally, as with all tree based ORAMS, each block in a bucket also contains the encryption of its address. That is, the address of each block is encrypted separately from the block itself.

4.2.2 C – ORAM: First Construction

In this section we present a technique allowing amortized constant communication complexity with a smaller block size.

To access an element in C-ORAM, i.e., read or write, the client first fetches the corresponding tag from the position map. This tag defines a unique path starting from the root of the ORAM tree and going to a specific leaf given by the tag. The element might reside in any bucket on this path. To find this element, is used a PIR read [23] that will be applied to each bucket. To verify whether the block exists in a bucket, the client downloads the encrypted headers of each bucket. Therewith, the client can

generate a PIR read vector retrieving the block from a bucket. To preserve the scheme's obliviousness, the client sends PIR read vectors for each bucket on the path. Once the block has been retrieved, the client can modify the block's content if required, then insert it back into the root of the C – ORAM tree using PIR write. This is the standard Path-PIR behavior to read from or write into blocks [24].

Eviction takes place after every χ = O(z) access operations. As in Onion ORAM, a path in C – ORAM is selected following deterministic reverse lexicographic order. Then, the entire root of the ORAM tree is downloaded, randomly shuffled and written back (additively homomorphically) encrypted. Finally, the eviction is performed by repeatedly applying an oblivious merge on buckets along the selected path. Any bucket belonging to this path is obliviously merged with its parent while the other child of the parent will be overwritten by a copy of the parent bucket. The former bucket on the path is called the *destination* bucket and the latter one its *sibling* bucket.

Before starting the eviction of a specific path, an invariant of the eviction process is that siblings of buckets of this path are empty, except the leaves. After the eviction, all buckets belonging to the evicted path will be empty except the leaf [14]. Note that siblings of this path, after the eviction, will not be empty anymore.

Sibling buckets, since they are simply copies of their parents, will contain blocks with tags outside the subtree of this bucket. These blocks are called noisy blocks as they do not belong into this subtree and are essentially leftover "junk". Now for correctness, in this construction, is guaranteed that the number of noisy blocks in any bucket is upper bounded. So, there will always be space for real elements in a bucket and will not overflow.

Elements in each bucket are encrypted using additively homomorphic encryption, respectively. Given two buckets B_1 and B_2, oblivious merging will permute the position of blocks in B_1 such that there are no real or noisy element at the same positions in B_1 and B_2. Consequently, if there is a real element in the i^{th} position in B_1, then for the scheme to be correct, the i^{th} position in B_2 should be empty. The following addition of elements at the same position in B_1 and B_2 will preserve the value of the real element. After χ operations, we also download the leaf bucket to delete its noisy blocks.

4.2.2.1 Details and Analysis

Let $\mathcal{P}(tag)$ denote the path starting from the root and going to the leaf identified by tag. The path is composed of L + 1 buckets including the root. $\mathcal{P}(tag, i)$ refers to the bucket at the i^{th} level of $\mathcal{P}(tag)$. For example, $\mathcal{P}(tag, 0)$ is the root bucket. $\mathcal{P}_s(tag, i)$ is the sibling of bucket $\mathcal{P}(tag, i)$. Let $[N]$ be the set of integers $\{1, \ldots, N\}$, $x \xleftarrow{\$} [N]$ uniformly sampling a random element from set $[N]$, and χ the period of eviction

which is in O(z). Identity is an empty bucket containing only encryptions of zero. *Figure 23* presents details of the access operation. An access can be either an ORAM Read or a Write operation. The only difference between the two is that a write changes the value of the block before putting it back in the root. The access operation invokes a PIR read algorithm, see *Figure 24* that obliviously retrieves a block. *Figure 25* shows the eviction where elements percolate towards their leaves using oblivious permutations, see

Figure 26.

Block size. The following asymptotic analysis will be in function of z, N, and γ. z is the size of the bucket, N the number of elements, and γ the length of the ciphertext of the additively homomorphic encryption. The communication complexity induced by an ORAM access operation comprises a PIR read operation and the eviction process (happening every $\chi \in O(z)$ accesses). The size of the bucket is $\mu \cdot z$, it is shown in security analysis section later that μ is a constant. First, the client performs PIR reads $L + 1$ times. For this, the client has to download all addresses in the path, i.e., $O(z \cdot L \cdot \log N)$ bits. Also, the client should send a logarithmic number of PIR read vectors V with size $O(\gamma \cdot z \cdot L)$ bits. Note that the computation of PIR read vectors outputs, for all but one buckets' block, encryption of zeros. Instead of sending back a logarithmic number of blocks to the client, the server only sends a single block, the summation of all the blocks output, cf. *Figure 23*. Thus, the client only retrieves a single block B. A PIR read applied to all buckets of the path induces an overhead in $O(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + B)$. For the eviction, the client downloads header_1 and the i^{th} column of header_2 and sends permutations for all buckets in the path. Thus, the overhead induced by the permutations is $O(L \cdot z \cdot \log z)$ bits. Also, after every $\chi = O(z)$ operations, the client downloads the root and one leaf, which has $O(zB)$ communication complexity. Amortized, for each operation we have $O_z(B)$ communication complexity (amortized over z). In conclusion, each access has $O_z(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + z \cdot \log(z) \cdot L + B)$ communication complexity. To have constant communication complexity in B, the block size should be $B \in \Omega(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + L \cdot z \cdot \log z) \in \Omega(\lambda \cdot \log^2 N + \gamma \cdot \lambda \cdot \log N)$. This is a consequence of $z = \Theta(\lambda)$, $\lambda \in \omega(\log n)$, and $L \in \Theta(\log N)$. Based on current attacks [25]. Therefore, $\lambda \cdot \log^2 N$ is dominated by $\gamma \cdot \lambda \cdot \log N$, and $B \in \Omega(\gamma \cdot \lambda \cdot \log N)$.

The main idea of the construction below is based on that the block size has exactly the same asymptotic as transmitted vectors V. So to improve the block size, in the next section we present a different way to access the ORAM that introduced in [22].

Access(op, adr, data, ctr, st)

1: tag = posMap(adr)
2: posMap(adr) $\xleftarrow{\$}$ [N]
3: **if** ctr = 0 mod (χ) **then**
4: Download root buckt, refresh encryption, randomize order of real elements
5: Evict(st)
6: **else**
7: **for** i = 0 **to** L **do** B = B+RIP-Read(adr, $\mathcal{P}$(tag, i)
8: **end if**
9: **if** op := write **then** set B = data
10: ctr = ctr + 1
11: Upload IND-CPA encrypted block to root $\mathcal{P}$(tag, 0)

Figure 23: C-ORAM 1st Access Algorithm

PIR-Read(addr, $\mathcal{P}$(tag, level))

1: Retrieve and decrypt address Addr of the bucket $\mathcal{P}$(tag, level)
2: **if** addr ∈ Addr **then**
3: a = Addr[addr]**if** ctr = 0 mod (χ) **then**
4: **for** i = 1 to μ · z **do**
5: if $i \neq$ a **then** $V_i \leftarrow$ ENC(0) **else**
6: $V_i \leftarrow$ ENC(1)
7: **end**
8: **else**
9: **for** i = 1 to μ · z **do** $V_i \leftarrow$ ENC(0)
10: **end if**
11: Parse bucket $\mathcal{P}$(tag, level) as (μ · z × |B|) binary matrix M
12: B = $(\sum_{i=1}^{\mu \cdot z} V_i \cdot \mathcal{M}_{1,i}, \ldots, \sum_{i=1}^{\mu \cdot z} V_i \cdot \mathcal{M}_{|B|,i})$
13: Update $header_1^{level}$ of bucket $\mathcal{P}$(tag, level)

Figure 24: C-ORAM PIR-Read Algorithm

Evict(st)

1: **for** i = 0 **to** L -1 **do**
2: Retrieve $header_1^i$ and $header_1^{i+1}$
3: Retrieve C_i and C_{i+1} respectively the ith and (i+1)th column of $header_2^i$ and $header_2^{i+1}$ of the bucket $\mathcal{P}(st, i)$ and $\mathcal{P}(st, i+1)$
4: π ← GenPerm(($header_1^i$, C_i), ($header_1^{i+1}$, C_{i+1})) generate the oblivious permutation π
5: $\mathcal{P}(st, i+1)$ = π($\mathcal{P}(st, i)$) + $\mathcal{P}(st, i+1)$
6: **if** i < L – 1 **then**
7: $\mathcal{P}_s(st, i)$) = $\mathcal{P}(st, i)$ //Copy the parent bucket into its sibling
8: **else**
9: Retrieve $header_1^{i+1}$ and C_{i+1} from the sibling leaf
10: π ← GenPerm(($header_1^i$, C_i), ($header_1^{i+1}$, C_{i+1}))
11: $\mathcal{P}(st, i+1)$ = π($\mathcal{P}(st, i)$) + $\mathcal{P}(st, i+1)$
12: **end if**
13: Update($header_1^i$) and store it with bucket $\mathcal{P}_s(st, i)$
14: Update($header_1^{i+1}$) and store it with bucket $\mathcal{P}(st, i+1)$
15: $\mathcal{P}(st, i)$ = Identity
16: **end**

Figure 25: C-ORAM Evict Algorithm

```
GenPerm(A, B)
1: Let x_1, x_2 be the number of empty and noisy slots in A
2: Let y_1, y_2 be the number of empty and noisy slots in B
3: d_1 = x_1 - y_1
4: d_2 = x_2 - y_2
5: for i = 1 to μ · z do
6:   case B[i] is full z ←$ all empty slots in A
7:   case B[i] is noisy
8:     if d_2 > 0 then
9:       z ←$ all noisy slots in A
10:      d_2 = d_2 – 1
11:    else
11:      z ←$ all noisy slots in A
12:    end
13:  case B[i] is empty
14:    if d_1 > 0 then
15:      z ←$ all non-assigned slots in A
16:      d_1 = d_1 – 1
17:    else
18:      z ←$ all full slots in A
19:    end
20:  end
21:  π(i) = z
22:  A[z] = assigned
23: end
24: return π
```

Figure 26: C-ORAM GenPerm Algorithm

4.2.3 C – ORAM: Second Construction

In this section we show how C – ORAM further reduces the block size – again by a multiplicative factor of $log\ N$ compared to the previous construction. Recall that in the previous section, the worst case involves a blowup of O(z), because during eviction the client needs to download O(z · B) bits. In this construction, the eviction remains exactly the same, and our focus will only be on ORAM access.

In 4.2.2, performed a PIR read per bucket during an access. Contrary, here is performed an oblivious merge to find out the block to retrieve. For an ORAM access to *tag*, our idea is to perform a special evict of path $\mathcal{P}(tag)$. All real elements pushed in $\mathcal{P}(tag)$ towards the leaf and then simply access the leaf bucket. So, we preserve access obliviousness and make sure that the element we want is pushed into leaf bucket *tag*.

This approach comes with several challenges. The bucket distribution must be preserved, i.e., maintaining sibling emptiness property, as guaranteed by the reverse lexicographic eviction, before evicting any path. Instead of deterministically selecting

a path for eviction, in this construction paths are chosen randomly. However, using randomized eviction, should be guaranteed empty siblings on the evicted path. By randomly evicting a path, might end to copy a bucket in its sibling which is not empty resulting therefore in a correctness flaw.

Temporarily clone the path $\mathcal{P}(tag)$ due to challenges above. The clone of $\mathcal{P}(tag)$ serves to simulate the eviction towards the leaf bucket, and we remove the clone after the access operation. We apply the oblivious merging on the bucket of this cloned path, and at the end we will have all real elements in the leaf bucket of the cloned path. Finally, we apply a PIR read to retrieve the block.

Besides, to get rid of the amortized cost and have a scheme that only requires a constant bandwidth in the worst case, we make use of a PIR write operation that will be performed during every access. In the construction of 4.2.2, we have to shuffle the root bucket since oblivious merging has to be performed on random buckets for security purposes. Moreover, we need to eliminate noisy blocks from the leaf buckets and therefore after each operations, the client downloads the evicted leaf to eliminate all noisy blocks. In the second C – ORAM construction, we are evicting after every access. Consequently, we can be certain that the root bucket is always empty after an eviction. The first PIR write operation that we perform will randomly insert the block in an empty root bucket after any access obliviously. The second use of PIR write is to delete the retrieved element from the leaf. In fact, we can also delete noisy blocks by the same tool but a PIR read is needed to retrieve first the noisy block that we will overwrite with a PIR write.

4.2.3.1 Details and Analysis

Algorithm in *Figure 27* presents the core of the second C – ORAM construction. Now, instead of performing a logarithmic number of PIR reads, we only invoke an **Evict-Clone** to read a block, cf. *Figure 28*. **Evict-Clone** uses oblivious merging of 4.2.1.1, together with one PIR read to retrieve a block. Moreover, we evict after every access. In order to eliminate noisy blocks that have been percolated to the leaf bucket, we use a PIR write to delete the noisy block, cf. *Figure 29*.

Block size. The access operation in C – ORAM is composed of scheduled path eviction, eviction in the cloned path, a PIR read, and two PIR writes. The size of the headers are negligible compared to the PIR read and write vectors. First, the eviction always involves an overhead of $O(z\,L\,log\,z)$. **Evict-Clone** performs one PIR read in addition to the regular evict. Finally, we retrieve the block of size B. Therefore, the overhead induced by these steps is $O(z\,L\,log\,z + z\,log\,N + \gamma z + B)$. Adding the two PIR writes and single PIR read operation will not change asymptotic behavior since the number of these operations is constant in N. In conclusion, to have a bandwidth that is constant in block size B, the block size should be $B \in \Omega(z \cdot L \cdot log\,z + \gamma z)$. With $z \in$

Θ(λ), λ ∈ ω($log\ N$) and L ∈ ($log\ N$), we achieve B ∈ Ω(λ [$log\ N \cdot log\ \lambda$ + γ]). In practice, γ ∈ O(λ^3), so dominates $log\ N \cdot log\ \lambda$. Therefore, block size B is B ∈ Ω(γλ). This C – ORAM construction achieves worst-case constant blow-up, it also omits inefficient PIR reads performed for ORAM access. This second construction improves the blocks size by a multiplicative factor of $log^2\ N$ compared to Onion ORAM in the worst case. As you can see, the main overhead of C-ORAM's block size comes from the size of ciphertext. Recall that γ ∈ O(λ^3). Therefore, the smaller the additively homomorphic ciphertext will get, the smaller the block size of C-ORAM will be.

Access(op, adr, data, st)

1: tag = posMap(adr)x ← position[a]
2: posMap(adr) $\xleftarrow{\$}$ [N]
3: B = Evict-Clone(adr, tag)
4: **if** op = write **then** set B = data
5: $pos_1 \xleftarrow{\$}$ [μ · z]
6: PIR-Write(pos_1, B, $\mathcal{P}$(st, 0))
7: Evict(st)
8: $pos_2 \xleftarrow{\$}$ [$header^L$]
9: N = PIR-Read(pos_2, $\mathcal{P}$(st, L))
10: PIR-Write(pos_2, -N,$\mathcal{P}$(st, L))

Figure 27: C-ORAM 2nd Access Algorithm

Evict-Clone(adr, tag)

1: Create a copy of the C-ORAM path $\mathcal{P}$(tag)
2: **for** i = 0 **to** L -1 **do**
3: Retrieve $header_1^i$ and $header_1^{i+1}$
4: Retrieve C_i and C_{i+1} respectively the i[th] and (i+1)[th] column of $header_2^i$ and $header_2^{i+1}$ of the bucket $\mathcal{P}(tag, i)$ and $\mathcal{P}(tag, i+1)$
5: π ← GenPerm(($header_1^i$, C_i), ($header_1^{i+1}$, C_{i+1}))
6: $\mathcal{P}(tag, i+1)$ = π($\mathcal{P}(tag, i)$) + $\mathcal{P}(tag, i+1)$
7: **end**
8: B = PIR-Read(adr, $\mathcal{P}(tag, L)$)
8: **for** i = 0 to L **do**
10: Update($header_1^i$) in $\mathcal{P}(tag, i)$
11: **end**

Figure 28: C-ORAM Evict-Clone Algorithm

PIR-Write(pos, block, $\mathcal{P}$(tag, level))

1: **for** i = 1 to μ · z **do**
2: if $i \neq$ pos **then** $V_i \leftarrow$ ENC(0) **else**
3: $V_i \leftarrow$ ENC(1)
4: **end**
5: Parse bucket $\mathcal{P}$(tag, level) as (μ · z × |B|) binary matrix M
6: $M_{i,j} = \mathcal{W}_i \cdot \mathcal{B}_j$
7: $\mathcal{P}$(tag, level) = M + $\mathcal{P}$(tag, level)

Figure 29: C-ORAM PIR-Write Algorithm

4.2.4 Security Analysis

The permutations generated by Algorithm Evict (*Figure 25*) are indistinguishable from random permutations. Informally, the adversary cannot gain any knowledge about the load of a particular bucket. Applying a permutation from Algorithm Evict (*Figure 25*) is equal to applying any randomly chosen permutation. We denote the adversarial permutation indistingui-shability experiment as PermG. Let $\mathcal{M}$ denote a probabilistic algorithm that generates permutations based on the configuration of two buckets and $\mathcal{A}$ a PPT adversary. Let $\mathcal{k}$ be the bucket size and $\mathcal{s}$ the security parameter. By Perm we denote the set of all possible permutations of size $\mathcal{k}$. Let $\mathcal{E}_1$ = (Gen, Enc, Dec) and $\mathcal{E}_2$ = (Gen_a, Enc_a, Dec_a) respectively denote an IND$-CPA encryption and an IND-CPA additively homomorphic encryption schemes. $\text{Perm}G^{\mathcal{A}}_{\mathcal{M},\,\mathcal{E}_1,\,\mathcal{E}_2}(\mathcal{s})$ refers to the instantiation of the experiments by algorithm $\mathcal{M}$, $\mathcal{E}_1$, $\mathcal{E}_2$ and adversary $\mathcal{A}$. The experiment $\text{Perm}G^{\mathcal{A}}_{\mathcal{M},\,\mathcal{E}_1,\,\mathcal{E}_2}(\mathcal{s})$ consists of:

- Generate two keys $\mathcal{k}_1$, $\mathcal{k}_2$ such that $\mathcal{k}_1 \xleftarrow{\$} \text{Gen}_a(1^{\mathcal{s}})$ and $\mathcal{k}_2 \xleftarrow{\$} \text{Gen}_a(1^{\mathcal{s}})$ and send $\mathcal{n}$ buckets additively homomorphic encrypted with $\text{Enc}_a(\mathcal{k}_1,.)$ associated to their headers encrypted with $\text{Enc}(\mathcal{k}_2,.)$ to the adversary $\mathcal{A}$
- The adversary $\mathcal{A}$ picks two buckets A and B, then sends the encrypted headers header(A) and header(B)
- A random bit b $\xleftarrow{\$}$ {0,1} is chosen. If b = 1, $\pi_1 \xleftarrow{\$} \mathcal{M}$(header(A), header(B)), otherwise $\pi_0 \xleftarrow{\$}$ Perm. Send π_2 to $\mathcal{A}$
- $\mathcal{A}$ has access to the oracle $\mathcal{O}_{\mathcal{M}}$ that issues permutations for any couple of headers different from those in the challenge
- $\mathcal{A}$ outputs a bit b'
- The output of the experiment is 1 if b' = b, and 0 otherwise. If $\text{Perm}G^{\mathcal{A}}_{\mathcal{M},\,\mathcal{E}_1,\,\mathcal{E}_2}(\mathcal{s}, b') = 1$, we say that the adversary $\mathcal{A}$ succeeded.

Definition 7 (Indistinguishable permutation) Algorithm $\mathcal{M}$ generates indistinguishable permutation iff for all PPT adversaries $\mathcal{A}$ and all possible configurations of buckets A and B, there exists a negligible function negl, such that

$$\Pr[\text{Perm}G^{\mathcal{A}}_{\mathcal{M},\,\mathcal{E}_1,\,\mathcal{E}_2}(\mathcal{s}, 1) = 1] - \Pr[\text{Perm}G^{\mathcal{A}}_{\mathcal{M},\,\mathcal{E}_1,\,\mathcal{E}_2}(\mathcal{s}, 0) = 1] \leq \text{negl}(\mathcal{s})$$

Theorem 2. If $\mathcal{E}_1$ is IND$-CPA secure, $\mathcal{E}_2$ IND-CPA secure, then Algorithm Evict (*Figure 25*) generates indistinguishable permutations.

4.3 Comparison

In section 4.2 shown that the homomorphic multiplications, and in fact the nesting "onion" nature of Onion ORAM solution, is not necessary. With careful application of

an oblivious merging algorithm, all movement of blocks through the tree can be done with only homomorphic addition, resulting in a more computationally efficient algorithm. This also reduced the required block size by a O($log^2 N$) factor and, allows for O(1) communication complexity in the worst case. Finally, C – ORAM scheme requires only a small storage overhead compared to Onion ORAM. For practical parameter values, C – ORAM achieves significant improvement in block size and number of homomorphic operations. *Table 4* summarizes improvements of C – ORAM when compared to Onion ORAM.

Scheme	Block size B	Simplified block size	Worst-case bandwidth	# additions	# multiplications
Onion ORAM	Ω(γλ $log^2 N$)	Ω($log^6 N$)	O(1)	Θ(Bλ $log N$)	Θ(Bλ $log N$)
C – ORAM	Ω(λ[$log N$ log λ+γ])	Ω($log^4 N$)	O(1)	Θ(Bλ $log N$)	Θ (Bλ)

Table 4: Comparison of Onion ORAM and C-ORAM

5 ObliviStore ORAM Family

5.1 ObliviStore ORAM

ObliviStore **[5]** is a high performance, distributed ORAM-based cloud data store secure in the malicious model. ObliviStore achieves high throughput by making I/O operations asynchronous. Asynchrony introduces security challenges, i.e., information leakage must be prevented not only through access patterns, but also through timing of I/O events. On **[5]** various practical optimizations are proposed which are key to achieving high performance, as well as techniques for a data center to dynamically scale up a distributed ORAM. The authors shown that with 11 trusted machines (each with a modern CPU), and 20 Solid State Drives, ObliviStore achieves a throughput of 31.5MB/s with a block size of 4KB.

5.1.1 The ObliviStore ORAM Protocol

One naive way to distribute an ORAM is to have a single trusted compute node with multiple storage partitions. However, in this case, the computation and bandwidth available at the trusted node can become a bottleneck as the ORAM scales up. In this scheme is proposed a distributed ORAM that distributes not only storage, but also computation and bandwidth.

Oblivistore ORAM consists of an oblivious load balancer and multiple ORAM nodes. The key idea is to apply the partitioning framework twice. The partitioning framework was initially proposed to reduce the worst-case shuffling cost in ORAMs **[4, 3]**, but it could leverage it to securely perform load balancing in a distributed ORAM. Specifically, each ORAM node is a “partition” to the oblivious load balancer, which relies on the partitioning framework to achieve load balancing amongst multiple ORAM nodes. Each ORAM node has several storage partitions, and relies on the partitioning framework again to store data blocks in a random storage partition with every data access. One benefit of the distributed architecture is that multiple ORAM nodes can perform shuffling in parallel.

5.1.2 Detailed Distributed ORAM Construction

To access a block, the oblivious load balancer first looks up its position map, and determines which ORAM node is responsible for this block. The load balancer than passes the request to this corresponding ORAM node. Each ORAM node implements a smaller ORAM consisting of multiple storage partitions. Upon obtaining the requested block, the ORAM node passes the result back to the oblivious load balancer. The oblivious load balancer now temporarily places the block in its eviction caches. With every data access, the oblivious load balancer chooses *v* random ORAM nodes

and evicts one block (possibly real or dummy) to each of them, through an ORAM write operation.

Each ORAM node also implements the shuffling functionalities. In particular, the ORAM nodes can be regarded as a parallel processors capable of performing reshuffling in parallel. The oblivious load balancer need not implement any shuffling functionalities, since it does not directly manage storage partitions. Hence, even though the load balancer is a central point, its functionality is very light-weight in comparison with ORAM nodes which are in charge of performing actual cryptographic and shuffling work.

Notice that each ORAM node may not be assigned an equal amount of storage capacity. In this case, the probability of accessing or evicting to an ORAM node is proportional to the amount of its storage capacity. For ease of explanation, we assume that each storage partition is of equal size, and that each ORAM node may have different number of partitions – although in reality, could be supported partitions of uneven sizes in a similar fashion.

5.1.3 Dynamic Scaling Up

Adding compute nodes. When a new ORAM node processor is being added to the system (without additional storage), the new ORAM node processor registers itself with the load balancer. The load balancer now requests existing ORAM nodes to hand over some of their existing their partitions to be handled by the new processor. To do this, the ORAM nodes also need to hand over part of their local metadata to the new processor, including part of the position maps, eviction caches, and partition states. The load balancer also needs to update its local metadata accordingly to reflect the fact that the new processor is now handling the reassigned partitions

Adding compute nodes and storage. The more difficult case is when both new processor and storage are being added to the system. One naive idea is for the ORAM system to immediately start using the new storage as one or more additional partitions, and allow evictions to go to the new partitions with some probability. However, doing so would result in information leakage. Particularly, when the client is reading the new partition for data, it is likely reading a block that has been recently accessed and evicted to this partition. In ORAM Scheme [5] is proposed a new algorithm for handling addition of new ORAM nodes, including processor and storage. When a new ORAM node joins, the oblivious load balancer and the new ORAM node jointly build up new storage partitions. At any point of time, only one storage partition is being built. Building up a new storage partition involves:

- ***Random block migration phase.*** The load balancer selects random blocks from existing partitions, and migrates them to the new partition. The new

partition being built is first cached in the load balancer's local trusted memory, and it will be sequentially written out to disk when it is ready. This requires about O(N/D) amount of local memory, where N is the total storage capacity, and D is the number of ORAM nodes. During the block migration phase, if a requested block resides within the new partition, the load balancer fetches the block locally, and issues a dummy read to a random existing partition (by contacting the corresponding ORAM node). Blocks are only evicted to existing partitions until the new partition is fully ready.

- ***Marking partition as ready*.** At some point, enough blocks would have been migrated to the new partition. Now the load balances sequentially writes the new partition out to disk, and marks this partition as ready.
- ***Expanding the address space.*** The above two steps migrate existing blocks to the newly introduced partition, but do not expand the capacity of the ORAM. We need to perform an extra step to expand ORAM's address space. Similarly, the challenge is how to do this securely. Suppose the old address space is [1, N], and the new address space after adding a partition is [1, N'], where $N' > N$. One naive idea is to randomly add each block in the delta address space [N + 1, N'] to a random partition. However, if the above is not an atomic operation, and added blocks become immediately accessible, this can create an information leakage. For example, after the first block from address space [N + 1, N'] has been added, at this time, if a data access request wishes to fetch the block added, it would definitely visit the partition where the block was added. To address this issue, the algorithm first assigns each block from address space [N + 1, N'] to a random partition – however, at this point, these blocks are not accessible yet. Once all blocks from address space [N + 1, N'] have been assigned, the load balancer notifies all ORAM nodes, and at this point, these additional blocks become fully accessible.

Initially, a new ORAM node will have 0 active partitions. Then, as new storage partitions get built, its number of active partitions gradually increases. Suppose that at some point of time, each existing ORAM node has c_1, c_2, ..., c_{m-1} partitions respectively, and the newly joined ORAM node has c_m active partitions, while one more partition is being built. Suppose all partitions are of equal capacity, then the probability of evicting to each active partition should be equal. In other words, the probability of evicting to the i'-th ORAM (where $i \in [m]$) node is proportional to c_i.

The remaining question is when to stop the migration and mark the new partition as active. This can be done as follows. Before starting to build a new partition, the oblivious load balancer samples a random integer from the binomial distribution $k \xleftarrow{\$} B(N,\rho)$, where N is the total capacity of the ORAM, and $\rho = \frac{1}{P+1}$, where P denotes the

total number of active partitions across all ORAM nodes. The goal is now to migrate k blocks to the new partition before marking it as active. However, during the block migration phase, blocks can be fetched from the new partition but not evicted back to it. These blocks fetched from the new partition during normal data accesses are discounted from the total number of blocks migrated. The full node join algorithm is in **[6]**.

5.1.4 Security Analysis

Definition 8 (Oblivious accesses and scheduling). Let seq_0 and seq_1 denote two data access sequences of the same length and with the same timing:

$$seq_0 := [(blockid_1, t_1), (blockid_2, t_2), \ldots, (blockid_m, t_m)]$$
$$seq_1 := [(blockid'_1, t_1), (blockid'_2, t_2), \ldots, (blockid'_m, t_m)]$$

Define the following game with an adversary who is in control of the network and the storage server:

- The client flips a random coin b.
- Now the client runs distributed asynchronous ORAM is algorithm and plays access sequence seq_b with the adversary.
- The adversary observes the resulting event sequence and outputs a guess b' of b.

We say that an asynchronous ORAM is secure, if for any polynomial-time adversary, for any two sequences seq_0 and seq_1 of the same length and timing, $|Pr[b' = b] - \frac{1}{2}| \leq negl(\lambda)$, where λ is a security parameter, and negl is a negligible function.

Theorem 3. ObliviStore (asynchronous) ORAM construction satisfies the security notion described in Definition 8 above.

Both the physical addresses accessed and the sequence of events observed by the server are independent of the data access sequence. In [6] is shown that an adversary can perform a perfect simulation of the scheduler without knowledge of the data request sequence. Specifically, both the timing of I/O events and the physical addresses accessed in the simulation are indistinguishable from those in the real world.

5.2 Burst ORAM

Burst ORAM [13] is an oblivious cloud storage system that achieves both practical response times and low total bandwidth consumption for bursty work-loads by reducing online bandwidth costs and aggressively rescheduling shuffling work to delay the bulk of the IO until the idle periods. In this schema, authors focus on reducing

effective IO by reducing online IO and delaying offline IO. This approach achieves to satisfy bursts of requests quickly, and delay most IO until idle periods. Moreover, it allows many bursts to be satisfied with nearly a 1× effective bandwidth cost. That is, during the burst, one block is transferred for every block requested. After the burst extra IO executed to catch up on shuffling and prepare for future requests. Before we proceed in details of this protocol we provide a short outline of Burst ORAM techniques, challenges and what is considered as burst.

Bursts. Intuitively, a burst is a period of frequent block requests from the client preceded and followed by relatively idle periods. Many real-world workloads exhibit bursty patterns (e.g. [26, 27]). Often, bursts are not discrete events, such as when multiple network file system users are operating concurrently. Thus Burst ORAM handles bursts fluidly: the more requests issued at once, the more Burst ORAM tries to delay offline IO until idle periods.

Challenges. When building a burst-friendly ORAM system there are several challenges. The first is ensuring security maintenance. A naive approach to reducing online IO may mark requests as satisfied before enough blocks are read from the server, leaking information about the requested block's identity. The second challenge is ensuring that we maximally utilize client storage and available bandwidth while avoiding deadlock. An excessively aggressive strategy that delays too much IO may use so much client space that we run out of room to shuffle. It may also underutilize available bandwidth, increasing response times. On the other hand, an overly conservative strategy may under-utilize client space or perform shuffling too early, delaying online IO and increasing response times.

Techniques. Burst ORAM addresses the challenges above by combining several novel techniques. It is introduced a new XOR technique for reducing online bandwidth cost to nearly 1×, Prioritizing online IO and delaying offline/ shuffle IO until client memory is nearly full. Also, Burst ORAM prioritizes efficient shuffle jobs in order to delay the bulk of the shuffle IO even further, ensuring that is minimized effective IO during long bursts. Last but not least, available client space is used to cache small levels locally to reduce shuffle IO.

5.2.1 The Burst ORAM Protocol

Burst ORAM achieves low response time by prioritizing online IO over shuffle IO. That is, shuffle IO suppressed during bursts, delaying it until idle periods. Requests are satisfied once online IO finishes, so prioritizing online IO allows to satisfy all requests before any shuffle IO starts, keeping response times low even for later requests. During the burst, requests are processed by fetching blocks from the server, but since shuffling is suppressed, no blocks are uploaded. Thus, we must resume shuffling once client storage fills. When available bandwidths are large and bursts are short, the

response time saved by prioritizing online IO is limited, as most IO needed for the burst can be issued in parallel. However, when bandwidth is limited or bursts are long, the savings can be substantial. With shuffle IO delayed until idle times, online IO dominates the effective IO, becoming the bottleneck during bursts. Here it comes the new XOR technique, which introduced in [13] and reduces Online IO.

5.2.1.1 XOR Technique

The XOR technique allows the Burst ORAM server to combine the O($log\ N$) blocks fetched during a request into a single block that is returned to the client, reducing the online bandwidth cost to O(1). If only the desired block was fetched, its identity would be revealed to the server. Instead, XOR all the blocks together and return the result. Since there is at most one real block among the O($log\ N$) returned, the client can locally reconstruct the dummy block values and XOR them with the returned block to recover the encrypted real block.

In Burst ORAM, as in ObliviStore, each request needs to retrieve a block from a single partition, which is a simplified hierarchical ORAM resembling those in [1]. The hierarchy contains $L \approx \frac{1}{2}\ log_2\ N$ levels with real-block capacities 1, 2, 4, ..., 2^{L-1} respectively. To retrieve a requested block, the client must fetch exactly one block from each of the L levels. The XOR technique requires that the client be able to reconstruct dummy blocks, and that dummies remain indistinguishable from real blocks. This property is achieved by encrypting a real block b residing in partition p, level ℓ, and offset off as $AES_{sk_{p,\ell}}(off\ ||\mathrm{B})$. An encrypted dummy block residing in partition p, level ℓ, and offset off as $AES_{sk_{p,\ell}}(off)$. The key $sk_{p,\ell}$ is specific to partition p and level ℓ, and is randomized every time ℓ is rebuilt.

For simplicity, we start by considering the case without early shuffle reads. In this case, exactly one of the L blocks requested is the encryption of a real block, and the rest are encryptions of dummy blocks. The server XORs all L encrypted blocks together into a single block X_Q that it returns to the client. The client knows which blocks are dummies, and knows p, ℓ, off for each block, so it reconstructs all the encrypted dummy blocks and XORs them with X_Q to obtain the encrypted requested/real block.

Early Shuffle Reads. An early shuffle read occurs when we need to read from a level with no more than half its original blocks remaining. Since such early shuffle reads may be real blocks, they cannot be included in the XOR. Fortunately, the number of blocks in a level is public, so the server already knows which levels will cause early shuffle reads. Thus, the server simply returns early shuffle reads individually, then XORs the remaining blocks, leaking no information about the access sequence. Since each early shuffle read block must be transferred individually, early shuffle reads increase online IO. Fortunately, early shuffle reads are rare, even while shuffling is

suppressed during bursts, so the online bandwidth cost stays under 2× and near 1× in practice.

5.2.1.2 Scheduling and Reducing Shuffle IO

In this section, we show how Burst ORAM schedules shuffle IO so that jobs that free the most client space using the least shuffle IO are prioritized. Thus, at all times, Burst ORAM issues only the minimum amount of effective IO needed to continue the burst, keeping response times lower for longer. We also show how Burst ORAM reduces overall IO by locally caching the smallest levels from each partition. We start by defining shuffle jobs.

In Burst ORAM, as in ObliviStore, shuffle IO is divided into per-partition shuffle jobs. Each job represents the work needed to shuffle a partition p and upload blocks evicted to p. A shuffle job is defined by five entities:

- A partition p to which the job belongs
- Blocks evicted to but not yet returned to p
- Levels to read blocks from
- Levels to write blocks to
- Blocks already read from p (early shuffle reads)

Each shuffle job moves through three phases:

Creation Phase. We create a shuffle job for p when a block is evicted to p following a request. Every job starts out inactive, meaning we have not started work on it. If another block is evicted to p, we update the sets of eviction blocks and read/write levels in p's inactive job. When Burst ORAM activates a job, it moves the job to the *Read Phase*, freezing the eviction blocks and read/write levels. Subsequent evictions to p will create a new inactive shuffle job. At any time, there is at most one active and one inactive shuffle job for each partition.

Read Phase. Once a shuffle job is activated, we begin fetching all blocks still on the server that need to be shuffled. That is, all previously unread blocks from all the job's read levels. Once all such blocks are fetched, they are shuffled with all blocks evicted to p and any early shuffle reads from the read levels. Shuffling consists of adding/ removing dummies, pseudo-randomly permuting the blocks, and then re-encrypting each block. Once shuffling completes, we move the job to the *Write Phase*.

Write Phase. Once a job is shuffled we begin storing all shuffled blocks to the job's write levels on the server. Once all writes finish, the job is marked complete, and Burst ORAM is free to activate p's inactive job, if any.

5.2.1.2.1 Prioritizing Efficient Jobs

Since executing shuffle IO delays the online IO needed to satisfy requests, the response time could be reduced by doing as little shuffling as is needed to free up space. The hope is that the bulk could be delayed of the shuffling until an idle period, so that it does not interfere with pending requests.

By the time client space fills, there will be many partitions with inactive shuffle jobs. Since we can choose jobs in any order, we can minimize the up-front shuffling work by prioritizing the most efficient shuffle jobs: those that free up the most client space per unit of shuffle IO. According to [13] the space freed by completing a job for partition p is the number of blocks evicted to p plus the number of early shuffle reads from the job's read levels. Thus, they define in [13] shuffle job efficiency as follows:

$$Job\ Efficiency = \frac{\#\ Evictions + \#\ Early\ Shuffle\ Reads}{\#\ Blocks\ to\ Read + \#\ Blocks\ to\ Write}$$

Job efficiencies vary substantially. Most jobs start with 1 eviction and 0 early shuffle reads, so their relative efficiencies are determined strictly by the sizes of the job's read and write levels. If the partition's bottom level is empty, no levels need be read, and only the bottom must be written, for an overall IO of 2 an efficiency of 0.5. If instead the bottom 4 levels are occupied, all 4 levels must be read, and the 5th level written, for a total of roughly 15 reads and 32 writes, yielding a much lower efficiency of just over 0.02. Both jobs free equal amounts of space, but the higher-efficiency job uses less IO.

Since small levels are written more often than large ones, efficient jobs are common. Further, by delaying an unusually inefficient job, it is given time to accumulate more evictions. While such a job will also accumulate more IO, the added write levels are generally small, so the job's efficiency tends to improve with time. Thus, prioritizing efficient jobs reduces shuffle IO during the burst, thereby reducing response times.

Unlike Burst ORAM, ObliviStore [5] does not use client space to delay shuffling, so there are fewer shuffle jobs to choose from at any one time. Thus, job scheduling is less important and jobs are chosen in creation order. Since ObliviStore is concerned with throughput, not response times, it has no incentive to prioritize efficient jobs.

5.2.1.2.2 Level Caching

Burst ORAM spends a lot of time accessing small levels. If client space used to locally cache the smallest levels of each partition, could be eliminated the shuffle IO associated with those levels entirely. Since levels are shuffled with a frequency inversely proportional to their size, each is responsible for roughly the same fraction of shuffle IO. Thus, shuffle IO could be greatly reduced by caching even a few levels

from each partition. Further, since caching a level eliminates its early shuffle reads, which are common for small levels, caching can also reduce online IO.

In Burst ORAM, cached only as many levels as are guaranteed to fit in the worst case. More precisely, it is identified the maximum number such that the client could store all real blocks from the smallest levels of every partition even if all were full simultaneously. Levels are cached by only updating an inactive job when the number of evictions is such that all the job's write levels have index at least . Since each level is only occupied half the time, caching levels consumes at most half of the client's space on average, leaving the rest for requested blocks.

5.2.2 Security Analysis

The server knows public information such as the values of each semaphore and the start and end times of each request. The server also knows the level configuration of each partition and the size and phase of each shuffle job, including which encrypted blocks have been read from and written to the server. The server must not learn the contents of any encrypted block, or anything about which plaintext block is being requested. Thus, the server may not know the location of a given plaintext block, or even the prior location of any previously requested encrypted block.

All of Burst ORAM's publicly visible actions are, or appear to the server to be, independent of the client's sensitive data access sequence. Since Burst ORAM treats the server as a simple block store, the publicly visible actions consist entirely of deciding when to transfer which blocks. Intuitively, Burst ORAM must ensure that each action taken is both deterministic and dependent only on public information, or appears random to the server. Equivalently, the schema must be able to generate a sequence of encrypted block transfers that appears indistinguishable from the actions of Burst ORAM using only public information. We now show how each Burst ORAM component meets these criteria.

ORAM Main and Client Security. ORAM Main (Figure 30) chooses whether to advance the Requester or the Shuffler, and depends on the size of the request queue and the Local Space semaphore. Since the number of pending requests and the semaphores are public, ORAM Main is deterministic and based only on public information. For each eviction, the choice of partition is made randomly, and exactly one block will always be evicted. Thus, every action in Figure 30 is either truly random or based on public information, and is trivial to simulate.

Requester Security. The Requester (*Figure 31*) must first identify the partition containing a desired block. Since the block was assigned to the partition randomly and this is the first time it is being retrieved since it was assigned, the choice of partition appears random to the server. Within each partition, the requester deterministically

retrieves one block from each occupied level. The choice from each level appears random, since blocks were randomly permuted when the level was created. The Requester singles out early shuffle reads and returns them individually. The identity of levels that return early shuffle reads is public, since it depends on the number of blocks in the level. The remaining blocks are deterministically combined using XOR into a single returned block. Finally, the request is marked satisfied only after all blocks have been returned, so request completion time depends only on public information. The Requester's behavior can be simulated using only public information by randomly choosing a partition and randomly selecting one block from each occupied level. Blocks from levels with at most half their original blocks remaining should be returned individually, and all others combined using XOR and returned. Once all blocks have been returned, the request is marked satisfied.

Shuffler Security. As in ObliviStore, Shuffler (Figure 32: Burst ORAM Shuffler Algorithm) operations depend on public semaphores. Job efficiency, which used for prioritizing jobs, depends on the number of blocks to be read and written to perform shuffling, as well as the number of early shuffle reads and blocks already evicted (not assigned). The identity of early shuffle read levels and the number of evictions is public. Further, the number of reads and writes depends only on the partition's level configuration. Thus, job efficiency and job order depend only on public information. Since the Shuffler's actions are either truly random (e.g. permuting blocks) or depend only on public information (i.e. semaphores), it is trivial to simulate.

Client Space. Since fetched blocks are assigned randomly to partitions, but evicted using an independent process, the number of blocks awaiting eviction may grow. The precise number of such blocks may leak information about where blocks were assigned, so it must be kept secret, and the client must allocate a fixed amount of space dedicated to storing such blocks. ObliviStore [5] relies on a probabilistic bound on overflow space provided in [2]. Since Burst ORAM uses ObliviStore's assignment and eviction processes, the bound holds for Burst ORAM as well. Level caching uses space controlled by the Local Space semaphore, so it depends only on public information.

```
Client and ORAM Main()
1: function ClientRead(b)
2:   Append b to RequestQueue
3:   On RequestCallBack(D(b)), return D(b)
4: procedure Write(b, d)
5:   Append b to RequestQueue
6:   On RequestCallBack(D(b)), write d to D(b)
7: procedure ORAM Main
8:   RequestMade ← false
9:   if RequestQueue ≠ ∅ then
10:    b ← Peek(RequestQueue)
11:    if Fetch(b) then      ▷ Request Issued
12:      RequestMade ← true
13:      Pop(RequestQueue)
14:      MakeEvictions
15:  if  RequestMade = false then
16:    TryShuffleWork
17: procedure MakeEvictions
18:  PendingEvictions = PendingEvictions + v
19:  while PendingEvictions ≥ 1 do
20:    p ← random partition
21:    Evict new dummy or assigned real block to p
22:    Vp = Vp + 1
23:    if shuffling p only writes levels ≥ λ then
24:      Jp ← p's  inactive job ▷ Create if needed
25:      V_Jp ← Vp
26:      if p has no active job then
27:        NJQ = NJQ ∪ Jp
28:    PendingEvictions = PendingEvictions – 1
```

Figure 30: Burst ORAM Client and ORAM Main Algorithm

```
Requester()
1: function Fetch(b)
2:   P(b), L(b) ← position map lookup on b
3:   Q = ∅, C = ∅
4:   for level ℓ ∈ P(b) do
5:     if ℓ is non-empty then
6:       b_ℓ ← b if ℓ = L(b)
7:       b_ℓ ← ID of next dummy in ℓ if ℓ ≠ L(b)
8:       if ℓ more than half full then
9:         Q ← Q ∪ S(b_ℓ)          ▷ Standard read
10:      else
11:        C ← C ∪ S(b_ℓ)          ▷ Early shuffle read
12:  Ret ← | C | + MAX(|Q|,1)     ▷ #blocks to return
13:  if Not TryDec(Local Space, Ret) then
14:    return false  ▷ Not enough space for blocks
15:  Dec(Concurrent IO, Ret)
16:  Issue async, request for (C, Q) to server
17:  When done, server calls:
18:    FetchCallBack(E(C, XOR, of E(Q))
19:  return true
20: procedure FetchCallBack({E(ci)},XQ)
21:  INC(Concurrent IO, 1)
22:  if b ∈ Q then
23:    X'_Q ← ⊕{E(q_i)| S(q_i) ∈ Q, q_i ≠ b}  ▷ Subtraction block, computed locally
24:    E(b) ← X_Q ⊕ X'_Q
25:  if b ∈ C then
26:    E(b) ← E(c_i) where c_i = b
27:  D(b) ← decrypt E(b)
28:  Assign b for eviction to random partition
29:  RequestCallBack(D(b))
```

Figure 31: Burst ORAM Requester Algorithm

Shuffler()

1: **procedure** TryShuffleWork
2: **if** NOT TryDec(Concurrent IO, 1) **then**
3: **return**
4: ReadIssued, WriteIssued ← false
5: **if** All reads for jobs in RJQ issued **then**
6: TryActive ⊳ Try to add job to RJQ
7: **if** $J_p \in$ RJQ has not issued read b_r **then**
8: **if** TryDec(Shuffle Buffer, 1) **then**
9: Issue assync.request for $S(b_R)$
10: When done: ReadCallBack($E(b_R)$)
11: ReadIssued ← true
12: **if** !ReadIssued and $J_p \in$ WJQ has write b_W **then**
13: Write E(b_W) to server
14: When done, call WriteCallBack($S(b_W)$)
15: WriteIssued ← true
16: **if** Not ReadIssued and Not WriteIssued **then**
17: INC(Concurrent IO, 1) ⊳ No shuffle work
18: **procedure** TryActive
19: **if** NJQ ≠ ∅ **then**
20: J_p ← Peek(NJQ) ⊳ Most efficient job
21: **if** TryDec(Shuffle Bufferm $V_{J_p} + A_{J_p}$) **then**
22: Mark J_p active ⊳ V_{J_p} frozen
23: INC(Local Space, $V_{J_p} + A_{J_p}$)
24: Move J_p from NJQ to RJQ
25: **procedure** ReadCallBack($E(b_R)$)
26: INC(Concurrent IO, 1)
27: Decrypt $E(b_R)$, place $D(b_R)$ in Shuffle Buffer
28: **if** all writes in J_p have finished **then**
29: Mark J_p complete
30: Remove J_p from WJQ
31: Update $C_p \leftarrow C_p + V_{J_p}, V_p \leftarrow V_p - V_{J_p}$
32: Add p's inactive job, if any, to NJQ

Figure 32: Burst ORAM Shuffler Algorithm

5.3 CURIOUS ORAM

CURIOUS ORAM [35] based upon a new set of metrics for evaluating ORAM designs, focusing more on latency, monetary expense, outsourcing ratio, elasticity and reliability. Among all existing designs, ObliviStore, which is built on partitioning the main ORAM into a set of smaller server-side ORAMs, turns out to be the most promising one. However, ObliviStore except of the privacy weakness in its implementation, it is overly complicated due to some of its specific performance optimization (e.g., background shuffling). CURIOUS is characterized by a set of fixed-size small ORAMs, offering a large constant outsourcing ratio, convenience for supporting asynchronous operations and the capability to expand and shrink its cloud-side storage. It has been built to ensure oblivious data access when serving multiple requests concurrently, and adopt a simpler eviction strategy, making it easier to implement. Also importantly, unlike ObliviStore, which is tied to a layered RAM scheme [5, 3], CURIOUS allows its underlying small, fixed-size ORAMs to be easily replaced. As a result, its performance will be continuously improved whenever a new design of such a building-block ORAM is available. For applications easily supported by ORAM, both ObliviStore and CURIOUS perform comparably. However, for demanding applications that stressed ORAM, CURIOUS significantly outperforms ObliviStore in response time (only its 25%), despite doubling the network traffic. In all cases, CURIOUS incurred lower monetary expense than (1/2 ∼ 2/3 of) ObliviStore.

5.3.1 The CURIOUS ORAM Protocol

We describe the design of CURIOUS, a modular partition-based framework which despite being asymptotically worse (in terms of bandwidth overhead) is able to outperform ObliviStore in both monetary expense and response time.

CURIOUS utilizes many small constant-size ORAMs (called subORAMs, or partition ORAMs), and uses existing remote storage services in a black-box way. At a high level, the framework consists of a position map, an eviction cache (both stored locally), and a collection of *m* subORAMs (whose state is kept locally, but whose storage is outsourced to the cloud). CURIOUS is modular: it cleanly separates the modules so that modules (e.g., partitions) can be improved upon independently, and specific modules may be replaced by others in order to suit a specific application scenario.

To process a request for block x, CURIOUS uses the position map to find which subORAM contains x. It then calls the subORAM module to both retrieve x and evict one or more (possibly dummy) blocks to that subORAM. Once retrieved, x is put into the eviction cache, and associated with a random subORAM. This ensures x will be

evicted at a random time, preventing the cloud from learning information about the requests from the subORAM access sequences.

Asynchronicity and concurrency. Oblivious processing of concurrent requests is challenging, and can compromise security or correctness when done incorrectly. To illustrate this, consider the sequence of requests *get, put, get*, all for the same block x. If an asynchronous scheme processes these requests sequentially, whereas other requests would be processed concurrently, it becomes vulnerable to certain attacks [35]. However, naively processing the three requests concurrently can compromise correctness or security, too. Indeed, naive processing would, for each request, lookup the position of x (i.e., which subORAM) using the position map, and then it would retrieve x. Now the cloud observes three concurrent requests to the same subORAM, an event that would happen only with probability $1/m^3$ for m subORAMs (e.g., $m = 2^{10}$), if the three requests were independent. In addition, it is necessary to ensure correctness, i.e., the last request (*get*) must return the data written to the block by the 2nd request (*put*), and consistency, e.g., a request should not unexpectedly override something written by a concurrent request. To address these requirements, CURIOUS leverages modularity and adopts a simple concurrency model: the event that any two requests are run concurrently is statistically independent of their requests' parameters (i.e., type and block). To ensure this, CURIOUS uses a sequential scheduling process that detects whether two requests are “in conflict" (e.g., they access the same block). By keeping track of pending requests, the framework can make such conflicts oblivious to the cloud (i.e., it appears as if such conflicting requests are of any two random requests).

Construction. Figure 33 describes the modular construction of CURIOUS. The interface includes *scheduleGet* and *schedulePut*, both of which are asynchronous (i.e., the call returns immediately, but the callback is invoked upon completing the request).

To process requests concurrently as well as obliviously, CURIOUS ensures that the event that any two requests are processed concurrently is statistically independent of those requests. Each request will operate on a random subORAM so two requests are only competing (i.e., must be executed sequentially) if they operate on the same subORAM. When a request operates on a subORAM, it gets a lock on it and only accesses blocks of that subORAM.

Two competing requests that scheduled sequentially, though the request processing (i.e., accessing the subORAM) can be asynchronous and concurrent. The idea is that when scheduling a request, the framework will mark the targeted block as “in transit", indicating that a request is in the process of retrieving that block. Subsequent requests for the same block are aware of the fact that the block is already being retrieved so they perform a dummy access (to a random subORAM) to hide (to the cloud) the fact

that the two requests targeted the same block. Upon finishing the first request, the block is put in the cache, and also delivered to each concurrent request targeting the same block. This prevents the kind of leaks uncovered in this Section, at the cost of disallowing requests to concurrently operate on the same subORAM.

To address correctness, i.e., it must appear (to the application) as if requests are processed sequentially, CURIOUS maintains a version id in the header of each block. This allows the framework to ensure that every *get* always retrieves the correct version of the data, even in presence of concurrent puts to the same block.

CURIOUS ORAM Framework

Local state:

1. Position map, storing for each block k: a pair (s, p), where s is the subORAM, and p the position within that subORAM.
2. Eviction cache, storing pairs (k, s), where k is a block and s is the index of the associated subORAM.
3. For each of the m subORAMs: their state.

sheduleGet(k, callback)

1. Call schedule(k, $\perp$, callback)

shedulePut(k, v,callback)

1. Call schedule(k, v, callback)

lookupPos(k)

1. Lookup (s, p) for block k
2. If block k is in cache, set s and p uniformly at random

evictBlocks(s)

1. Pick and remove c cache entries of the form (k, s)
2. If there are less than c such entries, add dummy blocks

addToCache(k)

1. Pick s, the index of the subORAM, uniformly at random
2. Store (k, s) in the cache

schedule(k, v,callback)

1. Call lookupPos(k) to get (s, p)
2. Call evictBlocks(s) to get e, te set of blocks to evict
3. Call s.retrieveBlockAndEvict(p, e)
4. If $v \neq \perp$, then overwrite then content of the block k with v
5. If block k is newly retrieved, call addToCache(k)
6. Call callback

Figure 33: CURIOUS ORAM Framewotk

Eviction. When a block is added to the eviction cache (e.g., as a result of a request) it becomes associated with a uniformly randomly chosen subORAM, following the same way as to ObliviStore. This random choice is integral to ensure obliviousness. CURIOUS performs evictions right after each subORAM access, i.e., a constant number of blocks associated with that subORAM (in the eviction cache) are re-written to it. Dummy blocks are used for padding, if needed. The subORAM module decides how many blocks are evicted. To the cloud, the eviction process is statistically independent of the requests, because all it sees is a fixed number of blocks (some of which may be

dummy) evicted to the same subORAM that was just accessed. This is guaranteed if the subORAM's module eviction process (in terms of what the cloud sees) is also independent of which blocks are evicted.

SubORAM. A subORAM module defines a single function: *retrieveBlockAndEvict*, which retrieves a block (given its position information) and evicts a list of blocks in one operation. The module is a tree-based ORAM, which resembles PathORAM [2]. The construction, shown in Figure 34, makes use of a b -ary tree (for any $b \geq 2$) whose nodes are buckets containing z b blocks, for a small integer z. The security of this subORAM design can be easily derived from that of PathORAM. Namely, blocks written to a subORAM are associated with a uniformly random leaf, hence a random path accessed per ORAM read/write. Note that, there is no stash associated with the subORAM, instead when a path overflows, we add the over own blocks to the client cache. We can choose values of b and z, such that the outsource ratio remains satisfactory. Additionally, in order to exploit the download/upload asymmetry, we can deterministically re-write only the first half of the path some of the time, e.g., for the first out of every two re-writes, so as to lower the average number of uploaded nodes per request (at the cost of lowering the outsource ratio, since blocks are more likely to overflow).

subORAM

Storage organization:

1. A b-ary tree of depth d where each node has a capacity of z blocks.

retrieveBlockAndEvict(p, e)

1. Retrieve path ending at leaf p from storage
2. For each block $\in e$, pick a random leaf ℓ
3. For each block $\in e$, update the position map with its new leaf ℓ
4. Rewrite path p, pushing blocks down to the leaf as far as possible

Figure 34: CURIOUS ORAM - subORAM design

5.3.2 Security

Part of the challenge to secure an asynchronous scheme involves timing. There are two ways in which timing may leak information: (1) through the application running on top of ORAM whose requests have input-dependent timing patterns, and (2) due to a weakness in the ORAM design itself. Like ObliviStore, CURIOUS only addresses the latter. The former is not meant to be prevented by ORAM, which was not designed to hide application timing. Here is adopted the security definition of ObliviStore (Definition 8 in Section 5.1.4), which roughly says that for any two applications with the same timing pattern, the ORAM's timing and accesses must be statistically indistinguishable. To prove the security of CURIOUS must shown that what is observed by the cloud provider is statistically independent of the requests (i.e., type, blocks, and timing). Consider a CURIOUS instance of a fixed capacity with *m* subORAMs. The cloud sees: (1) the timing of operations to the storage, (2) the

sequence of subORAMs accessed, and (3) the exact operations to each subORAM's raw storage. CURIOUS' concurrency model takes care of (1). SubORAM design takes care of (3) due to the statistical independence of the requests. This leaves (2) to be dealt with here.

Theorem 4. For any sequence of t requests, the sequence of subORAMs accessed by CURIOUS is statistically independent of the type and target block of those requests.

Proof. Consider an arbitrary request; there are three possibilities for the targeted block: (1) it has not been requested before; (2) it has been requested before and is in the cache; or (3) it has been requested before and is not in cache. Since during initialization blocks are randomly assigned to a subORAM, for (1), from the point of view of the cloud, a uniformly random subORAM will be accessed. For (2), the block is in the cache, so a uniformly random subORAM will be accessed. Finally for (3), the block is not in the cache, so it must have been evicted earlier, when a uniformly random subORAM was visited.

5.4 Comparison

5.4.1 ObliviStore ORAM vs Burst ORAM

For ObliviStore and Burst comparison two experiments are presented to [13] the Endless Burst Experiment and NetApp Workload Experiment.

Endless Burst Experiment. For the endless burst experiments, a 32TB ORAM was used with $N = 2^{33}$ 4KB blocks and 100GB client space. 2^{33} requests were issued at once, then start satisfying requests in order using each scheme. The bandwidth costs of each request was recorded, averaged over requests with similar indexes and over three trials. *Figure 35* and *Figure 36* show online and effective costs, respectively. The insecure baseline is not shown, since its online, effective, and overall bandwidth costs are all 1. *Figure 35* shows that Burst ORAM maintains 5X– 6X lower online cost than ObliviStore for bursts of all lengths. When Burst ORAM starts to delay shuffling, it incurs earlier shuffle reads, increasing online cost, but stays well under 2X on average. Burst ORAM effective costs can be near 1X because writes associated with requests are not performed until blocks are shuffled. Burst ORAM defers shuffling, so its effective cost stays close to its online cost until client space fills, while ObliviStore starts shuffling immediately, so its effective cost stays constant (Figure 41). Thus, response times for short bursts will be substantially lower in Burst ORAM than in ObliviStore. Eventually, client space fills completely, and even Burst ORAM must shuffle continuously to keep up with incoming requests. This behavior is seen at the far right of Figure 41, where each scheme's effective cost converges to its overall cost. Burst ORAM's XOR technique results in slightly higher overall cost than ObliviStore's level compression, so Burst ORAM is slightly less efficient for very long bursts. Without local level caching,

Burst ORAM spends much more time shuffling the smallest levels, yielding the poor performance of Burst ORAM No Level Caching. If shuffle jobs are started in arbitrary order, as for Burst ORAM No Prioritization, the amount of shuffling per request quickly increases, pushing effective cost toward overall cost. However, by prioritizing efficient shuffle jobs as in Burst ORAM proper, more shuffling can be deferred, keeping effective costs lower for longer, and maintaining shorter response times.

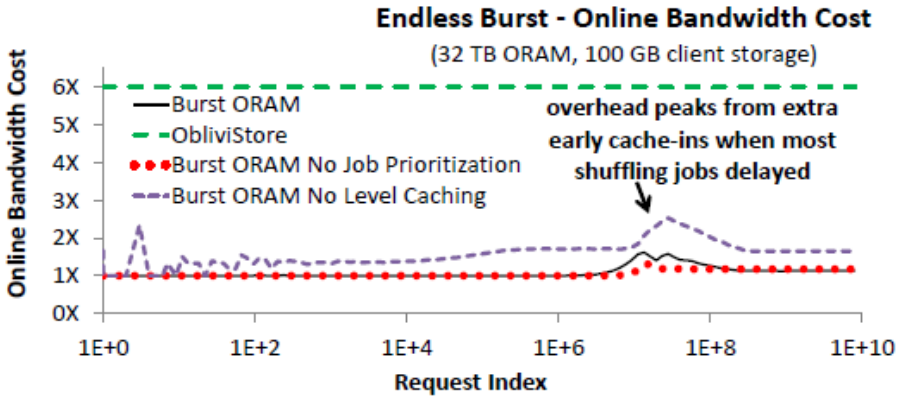

Figure 35: Endless Burst – Online Bandwidth Cost

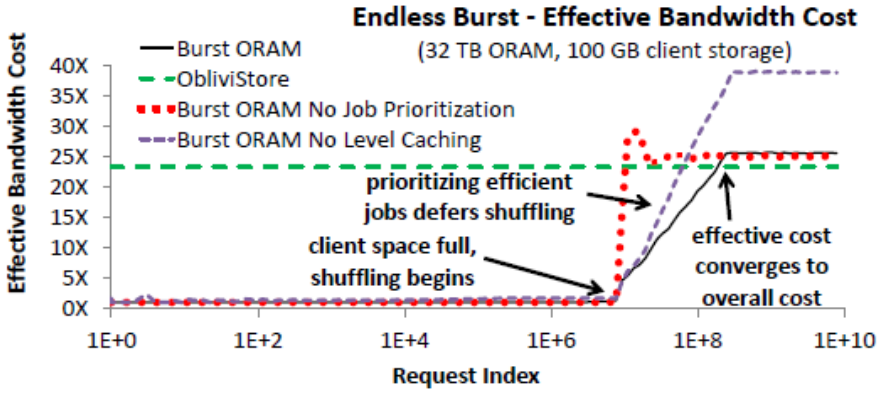

Figure 36: Endless Burst – Effective Bandwidth Cost

NetApp Workload Experiment. The NetApp experiments show how each scheme performs on a realistic, bursty workload. Burst ORAM exploits the bursty request patterns, minimizing online IO and delaying shuffle IO to achieve near-optimal response times far lower than ObliviStore's. Level caching keeps Burst ORAM's overall bandwidth costs low. *Figure 37* shows 99.9-percentile response times for several schemes running the 15-day NetApp workload for varying bandwidths. All experiments assume a 50ms network latency. For most bandwidths, Burst ORAM response times are orders of magnitude lower than those of ObliviStore and comparable to those of the insecure baseline. Shuffle prioritization and level caching noticeably reduce response times for bandwidths under 1Gbps.

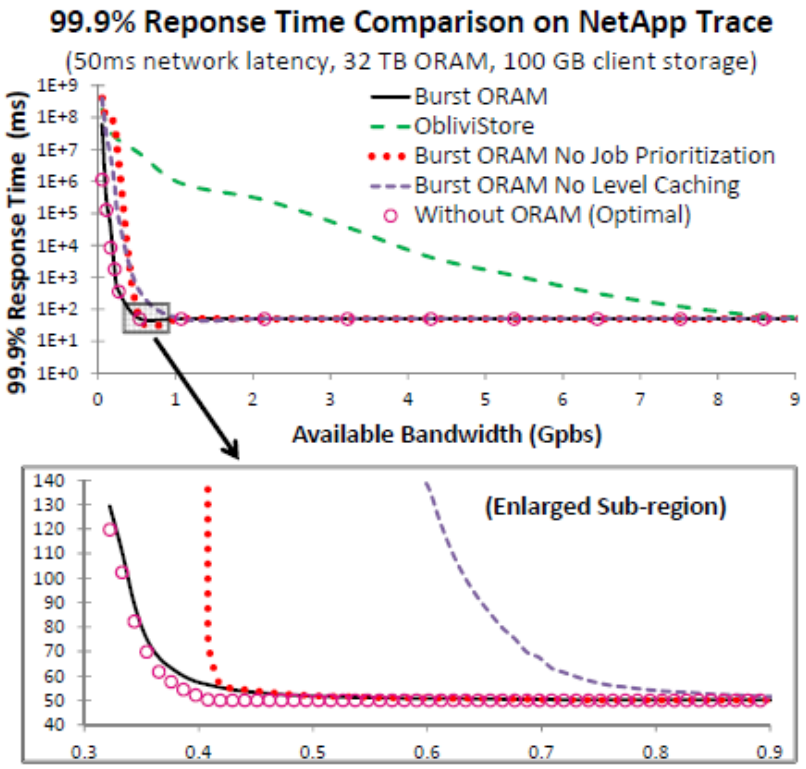

Figure 37: 99.9% Reponse Time Comparison on NetApp Trace[1]

Figure 38 compares p-percentile response times for p values of 90%, 99%, and 99.9%. It gives absolute p-percentile response times for the insecure baseline, and differences between the insecure baseline and Burst ORAM p-percentile response times (Burst ORAM overhead). When baseline response times are low, Burst ORAM response times are also low across multiple p.

Figure 39 shows the overall bandwidth costs incurred by each scheme running the NetApp workload at 400Mbps. Costs for other bandwidths are similar. Burst ORAM clearly achieves an online cost several times lower than ObliviStore's. Level caching reduces Burst ORAM's overall cost from 42X to 29X. Burst ORAM's higher cost is due to a combination of factors needed to achieve short response times. First, Burst ORAM uses the XOR technique, which is less efficient overall than ObliviStore's mutually exclusive level compression. Second, Burst ORAM handles smaller jobs first. Such jobs are more efficient in the short-term, but since they frequently write blocks to small

[1] (Top) Burst ORAM achieves short response times in bandwidth-constrained settings. Since ObliviStore has high effective cost, it requires more available client-server bandwidth to achieve short response times. (Bottom) Burst ORAM response times are comparable to those of the insecure (without ORAM) scheme.

levels, they create more future shuffle work. In ObliviStore, such jobs are often delayed during a large job, so fewer levels are created, reducing overall cost.

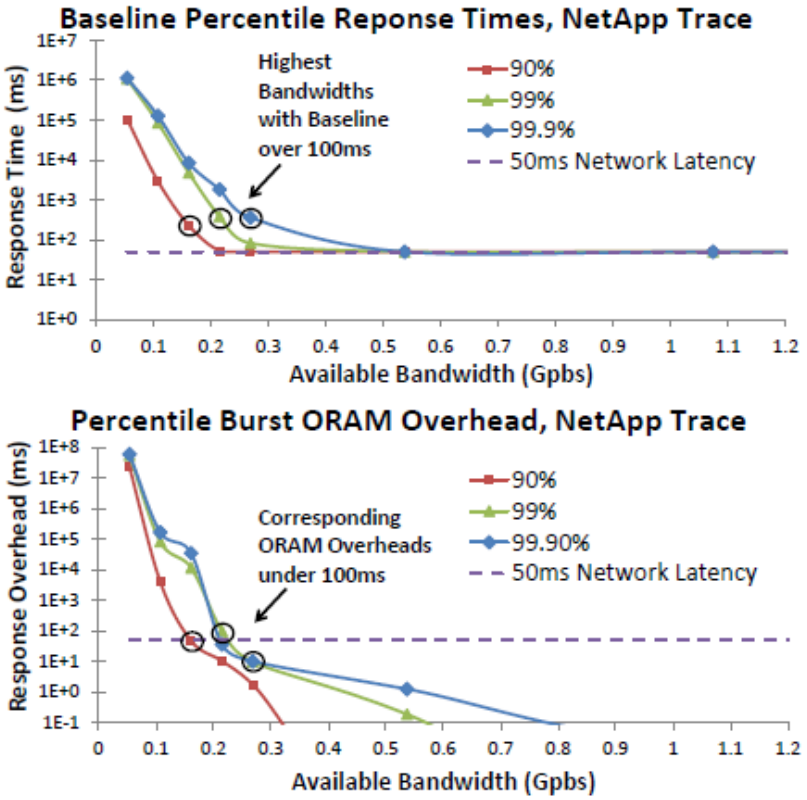

Figure 38: Comparison of Burst ORAM and Baseline[2]

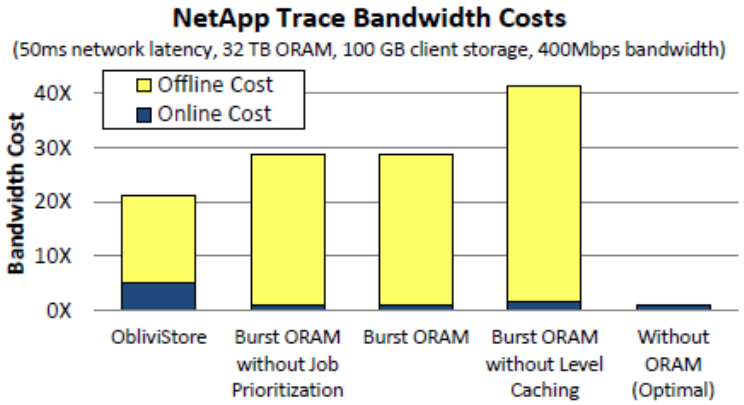

Figure 39: NetApp Trace Bandwidth Costs

[2] (Top) Insecure baseline (no ORAM) p-percentile response times for various p. (Bottom) Overhead (difference) between insecure baseline and Burst ORAM's p-percentile response times. Marked nodes show that when baseline p-percentile response times are < 100ms, Burst ORAM overhead is also < 100ms.

5.4.2 ObliviStore ORAM vs CURIOUS ORAM

CURIOUS was evaluated against ObliviStore using the following experimental setup.

Experiment settings. The experiments were conducted from a Linux server on a university network. The machine ran an ORAM client to interact with S3; the S3 buckets were placed on the US EAST1 Standard (North Virginia) Amazon S3 region. This region has the lowest round-trip time with the ORAM client. The bandwidth between the client and S3 was 50 MB/s downstream and 10 MB/s upstream2. In those particular experiments, an ORAM always started running in a warmed-up state (after $O(n)$ requests were processed where n is the capacity of the ORAM instance).

The application traces were replayed, with capacity 256MB and block size 16KB, in all cases. For a fair comparison, the number of sub-ORAMs of CURIOUS were set such that either schemes have roughly the same outsource ratio (or ObliviStore has the advantage). The results, displayed in *Comparison of CURIOUS and ObliviStoreTable 5*, show that CURIOUS is better in supporting the selected applications, i.e., its slowdown is either the same or less than ObliviStore, despite the more than doubled bandwidth usage in some cases (e.g., for fileserver, the bandwidth usage of tree-based CURIOUS is 4500.4 KB/req vs. only 2159:8 KB/req for ObliviStore). Further, it is shown that the monetary expense incurred by CURIOUS is between 1/2 and 2/3 that of ObliviStore. In terms of applications, both varmail and webproxy are easily supported by both schemes, but CURIOUS has slightly higher response time (e.g., 266 ms vs. 200ms for webproxy). This is due to the background shuffling of ObliviStore which, for less demanding applications, is beneficial because the shuffling cost is not paid upfront. For such applications, minimizing the response time below a certain threshold may not be required; instead in such cases, monetary expenses may outweigh small differences in response times. For these applications, CURIOUS' operating monetary cost is almost half of ObliviStore. For the demanding applications (i.e., webserver and fileserver) which stressed ORAMs, CURIOUS is a better fit than ObliviStore. Indeed, due to background shuffling, and high upload cost, ObliviStore experienced high response times and larger slowdown. Take webserver as an example, the response time is almost 4 times that of CURIOUS (i.e., 7.950 sec/req for ObliviStore vs. only 2.004 sec/req for CURIOUS) but the slowdown is comparable; both schemes are close to being able to fully support the application. For fileserver, though neither scheme is even close to satisfying the demands of this application, nevertheless CURIOUS significantly outperformed ObliviStore both in response time and slowdown.

		varmail	webproxy	webserver	fileserver
	Metric	16 KB	16 KB	16 KB	16 KB
ObliviStore	Bandwidth usage	623.0	590.8	454.9	2159.8
	Response time	0.196	0.200	7.950	13.531
	Slowdown	1.000	1.000	1.152	3.767
	Monetary cost	0.153	0.144	0.113	0.548
	Outsource Ratio	171.2	99.5	16.1	18.5
CURIOUS tree-based	Bandwidth usage	1025.0	988.7	852.5	4500.4
	Response time	0.270	0.266	2.004	2.114
	Slowdown	1.000	1.000	1.114	2.702
	Monetary cost	0.081	0.078	0.067	0.355
	Outsource Ratio	209.7	111.3	25.6	18.7

Table 5: Comparison of CURIOUS and ObliviStore

6 Applied ORAM Schemes

6.1 ObliviSync

ObliviSync [50] is an oblivious cloud storage system that specifically targets one of the most widely-used personal cloud storage paradigms: synchronization and backup services, popular examples of which are Dropbox, iCloud Drive, and Google Drive. This solution is asymptotically optimal and practically efficient, with a small constant overhead of approximately 4x compared with non-private file storage, depending only on the total data size and parameters chosen according to the usage rate, and not on the number or size of individual files. This construction also offers protection against timing-channel attacks, which has not been previously considered in ORAM protocols. In [50] built and evaluated a full implementation of ObliviSync that supports multiple simultaneous read-only clients and a single concurrent read/write client whose edits automatically and seamlessly propagate to the readers. It has been shown that the system functions under high workloads, with realistic file size distributions, and with small additional latency (as compared to a baseline encrypted file system) when paired with Dropbox as the synchronization service. The main goal in [50] was to present an efficient solution for oblivious storage on a personal cloud synchronization/backup provider such as (but not limited to) Dropbox or Google Drive.

6.1.1 ObliviSync Setting Overview

The setting consists of an untrusted cloud provider and one or more clients which backup data to the cloud provider. If there are multiple clients, the cloud provider propagates changes made by one client to all other clients, so that they each have the same version of the filesystem. Even if “Dropbox” is used as a shorthand for the scenario, the solution is not specific to Dropbox and will work with any similar system. This setting used because:

1. It is one of the most popular consumer cloud services used today, and is often colloquially synonymous with the term “cloud”.
2. The interface for Dropbox and similar storage providers is “agnostic,” in that it will allow you to store any data as long as you put it in the designated synchronization directory. This allows for one solution that works seamlessly with all providers.
3. Synchronization and backup services do not require that the ORAM hide a user’s read accesses, only the writes. This is because (by default) every client stores a complete local copy of their data, which is synchronized and backed up via communication of changes to/from the cloud provider.

Write-Only ORAM. The third aspect of the setting above (i.e., it doesn't need to hide read accesses) is crucial to the efficiency of ObliviSync system. Each client already has a copy of the database, so when they read from it they do not need to interact with the cloud provider at all. If a client writes to the database, the changes are automatically propagated to the other clients with no requests necessary on their part. Therefore, the ORAM protocol only needs to hide the write accesses done by the clients and not the reads. This is important because [51] have shown that write-only ORAM can be achieved with optimal asymptotic communication overhead of O(1). In practice, write-only ORAM requires only a small constant overhead of 3-6x compared to much higher overheads for fully-functional ORAM schemes, which asymptotically are Ω(logN). In [51] is presented a detailed description of the write-only ORAM, in which ObliviSync scheme based.

6.1.2 ObliviSync Scheme

ObliviSync system uses the idea of write-only ORAM on top of any file backup or synchronization tool in order to give multiple clients simultaneous updated access to the same virtual filesystem, without revealing anything at all to the cloud service that is performing the synchronization itself, even if the cloud service is corrupted to become an honest-but-curious adversary. Write-only ORAM is ideal for this setting because each client stores an entire copy of the data, so that only the changes (write operations) are revealed to the synchronization service and thus only the write operations need to be performed obliviously.

Improvements over write-only ORAM. Compared to the previous write-only ORAM construction [51], the authors in [50] made significant advances and improvements to fit this emergent application space:

- **Usability:** Users interact with the system as though it is a normal system folder. All the encryption and synchronization happens automatically and unobtrusively.
- **Flexibility:** A real filesystem is supported and innovative methods are used to handle variable-sized files and changing client roles (read/write vs. read-only) to support multiple users.
- **Strong obliviousness:** The design of OblivicSync system not only provides obliviousness in the traditional sense, but also protects against timing channel attacks. It also conceals the total number of write operations, a stronger guarantee than previous ORAM protocols.
- **Performance:** The system well matches the needs of real file systems and matches the services provided by current cloud synchronization providers. It can also be tuned to different settings based on the desired communication rate and delay in synchronization.

Basic architecture. The high-level design of ObliviSync is presented in Figure 40. There are two types of clients in the system: a read/write client (ObliviSync-RW) and a read-only client (ObliviSync-RO). At any given time, there can be any number of ObliviSync-RO's active as well as zero or one ObliviSync-RW clients. It is noted that a given device may work as a read-only client in one period of time and as a write-only client in other periods of time. Both clients consist of an actual backend folder as well as a virtual frontend folder, with a FUSE client running in the background to seamlessly translate the encrypted data in the backend to the user's view in the frontend virtual filesystem.

The system relies on existing cloud synchronization tools to keep all clients' backend directories fully synchronized. This directory consists of encrypted files that are treated as generic storage blocks, and embedded within these storage blocks is a file system structure loosely based on i-node style file systems which allows for variable-sized files to be split and packed into fixed-size units. Using a shared private key (which could be derived from a password) the job of both clients ObliviSync-RO and ObliviSync-RW is to decrypt and efficiently fetch data from these encrypted files in order to serve ordinary read operations from the client operating in the frontend directory.

The ObliviSync-RW client, which will be the only client able to change the backend files, has additional responsibilities: (1) to maintain the file system encoding embedded within the blocks, and (2) to perform updates to the blocks in an oblivious manner using ObliviSync ORAM.

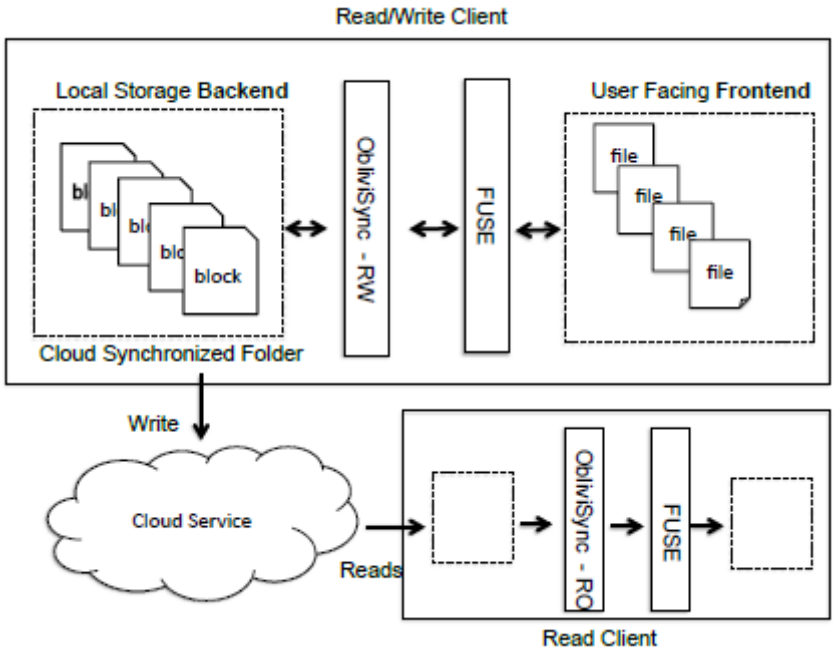

Figure 40: ObliviSync high-level design

User transparency with FUSE mount. From the user's perspective, however, the interaction with the frontend directory occurs as if interacting with any files on the host system. This is possible because the FUSE mount (file system in user space) interface displays the embedded file system within the backend blocks to the user as if it were any other file system mount. Under the covers, though, the ObliviSync-RO or ObliviSync-RW clients are using the backend directory files in order to serve all data requests by the client, and the ObliviSync-RW client is additionally monitoring for file changes/creations in the FUSE mount and propagating those changes to the backend.

Strong obliviousness through buffered writes. In order to maintain obliviousness, these updates are not immediately written to the backend filesystem by the ObliviSync-RW client. Instead, the process maintains a buffer of writes that are staged to be committed. At regular timed intervals, random blocks from the backend are loaded, repacked with as much data from the buffer as possible, and then re-encrypted and written back to the backend folder. From there, the user's chosen file synchronization or backup service will do its work to propagate the changes to any read-only clients. Moreover, even when there are no updates in the buffer, the client pushes dummy updates by rewriting the chosen blocks with random data. In this way, as the number of blocks written at each step is fixed, and these writes (either real or dummy) occur at regular timed intervals, an adversary operating at the network layer is unable to determine anything about the file contents or access patterns. Without dummy updates, for example, the adversary can make a reasonable guess about the size of the files that the client writes; continued updates without pause is likely to indicate that the client is writing a large file. Note that in some cases, revealing whether a client stores large files (e.g., movies) may be sensitive. The full source code of ObliviSyn implementation is available on GitHub [52].

6.1.3 Security Analysis

Security definitions are presented below, before the detailed security analysis of the system.

Notation. Here, the parameter L is the maximum number of bytes that may be modified, and t is the latest time that is allowed. Also, recall the parameters B: block pair size, N: number of backend block pairs, and k: drip rate.

Definition 9. $((L,t)$-$\boldsymbol{fsequence})$. A sequence of non-read operations for a block filesystem is a (L,t)-$\boldsymbol{fsequence}$ if the total number of bytes to be modified in the filesystem metadata and file data is at most L, and the last operation takes place before or at time t.

Definition 10. (**Write-only strong obliviousness**) Let L and t be the parameters for $fsequence$. A block filesystem is write-only strongly-oblivious with running time T, if for any two (L,t)-$fsequence$ P_0 and P_1, it holds that:

- The filesystem finishes all the tasks in each $fsequence$ within time T with probability $2 - neg(\lambda)$, where λ is the security parameter.
- The access pattern of P_0 is computationally indistinguishable to that of P_1.

Time to write all files. The **Theorem 5** below shows the relationship between the number of sync operation, the drip rate, and the size of the buffer. Specifically, it shows that, with high probability, a buffer with size s is completely cleared and synced to the backend after $O(\frac{s}{Bk})$ sync operations. This is optimal up to constant factors, since only Bk bytes are actually written during each sync.

Theorem 5. For a running ObliviSync-RW client with parameters B, N, k as above, let m be the total size (in bytes) of all non-stale data currently stored in the backend, and let s be the total size (in bytes) of pending write operations in the buffer, and suppose that $m + s \leq NB/4$. Then the expected number of sync operations until the buffer is entirely cleared is at most $4s/(Bk)$. Moreover, the probability that the buffer is not entirely cleared after at least $\frac{48s}{Bk} + 18r$ sync operations is at most $\exp(-r)$.

The proof of Theorem 5 is presented in [50].

Theorem 6. Let λ be the security parameter. Consider ObliviSync-RW with parameters B, N, k as above, and with drip time t. For any L and t as fsequence parameters, ObliviSync-RW is strongly-secure write-only filesystem with running time $T = t + \frac{48Lt}{Bk} + 18\lambda t$.

The proof of Theorem 6 is presented in [50].

6.1.4 Evaluation

The ObliviSync was evaluated against the following properties:

a. Throughput with fixed-size files
   - **Bandwidth overhead: 2x until 25% of the load.** With drip rate 3 (the solid line for k = 3), it takes about ~120 epochs on average to sync 25% of the frontend files. Note that the number of bytes that would be transferred to the cloud storage during 120 epochs is 120 · (k + 1) · B = 480 MB, and 25% of the frontend files amounts to 250 MB. So, the experiment shows that the system needs only 2x bandwidth overhead, when the front-end files occupies at most 25% of the total cloud storage, with the parameters chosen in this experiment. This is

better performance than what is shown in Theorem 5, which provably guarantees 4x bandwidth overhead.

- **Linear costs until 33% of the load.** The inflection point, between linear and super-linear, is particularly interesting. Apparent immediately is the fact that the inflection point is well beyond the 25% theoretic bound; even for a drip rate of k = 3, it manages to get at least 1/3 full before super-linear tendencies take over. Further, notice that for higher drip rates, the inflection point occurs for higher percentage of fullness for the backend.

b. Throughput with variable-size files
   - **Good performance for variable-size files.** After three runs, the average number of epochs needed to synchronize the two file loads is the same, 100 epochs. This clearly shows that ObliviSync systems is dependent on the total number of bytes to synchronize and not the size of the individual files.

c. Latency
   - **About 1 epoch to sync, even for high fill rates.** First, for lower fill rates, the time to complete a single file synchronization is roughly one epoch. At higher fill rates, it starts to take more epochs, on average, to sync a single file; however, even for the most conservative k = 3, it only takes at most 5 epochs even for very high fill rates. For more aggressive drip rates, k = 9,12 the impact of higher filler rates is diminished, still only requiring about 2 epochs to synchronize a single file.

d. The size of pending writes buffer
   - **Reasonable buffer size: at most 2 MB.** Clearly, as the fill rate increases, the amount of uncommitted data in the buffer increases; however, the relationship is not strictly linear. For example, with 20% full and 50% full, we see only a small difference in the buffer size for this extreme thrashing rate. At a fill rate of 75%, however, there is a noticeable performance degradation. Because most of the blocks selected at each epoch are either full or do not have enough space, due to fragmentation, the buffer cannot always be cleared at a rate sufficient to keep up with incoming writes. Thus, the size of the buffer doubles in comparison with the other workloads.

6.1.4.1 Functionality with Dropbox backend

Here, ObliviSync performance is measured on a real cloud synchronization service, namely Dropbox. Additionally, it is provided a baseline comparison of the overhead of ObliviSync, and so similar experiments were performed in [50] using EncFS [53] as the data protection mechanism.

Throughput over Dropbox. For both EncFS and ObliviSync, the interest is in a large number of files, namely 20% full or ∼200MB, and then was measured how long it took for the buffer to clear and all files to become available. Like before, a read and write computer is used, and the difference in the local and remote propagation delays of file synchronization is measured. For EncFS on the write computer, the propogation delay for all the files is nominal with files appearing nearly immediately. On the read computer, there is a propagation delay associated with Dropbox remote synchronization, and all files are accessible within 100 seconds. For ObliviSync on the write computer, a very similar throughput trend-line as in the prior experiments. In total, it takes just under 800 seconds (or 80 epochs) for all the files to synchronize. Interestingly, on the read computer, the propagation delay is relatively small, with respect to the overall delay, and files are accessible within an additional epoch or two. In total, these results clearly demonstrate that ObliviSync is functional and efficient to use over cloud synchronization services like Dropbox.

Latency over Dropbox. In the EncFS upon writing the file immediately it becomes available to write computer. However on the read computer, it takes a little under 5 seconds for the synchronization with Dropbox to complete for the same file to be accessible. This measurement forms a baseline of performance for the rate of DropBox synchronization without ObliviSync. For ObliviSync, on the write computer, an expected performance metric of just under 10 seconds for each file to be visible to the read mount. The reason it is under 10 seconds and not exactly 10 seconds, as the setting of the drip time, is that a write occurring between epoch timers will take less than an epoch to sync. The propagation rate to the read computer takes a similar time as that of EncFS (∼ 5 seconds); however, there is higher variance as more files need to be transferred by the Dropbox service per epoch (namely 4 = k +1 with the superblock). Still, this added variance is within 3x in terms of epochs: it takes at most 30 seconds for a file to sync (or 3 epochs of waiting), which is very reasonable considering the built-in overhead of the system.

6.2 Tiny ORAM

Tiny ORAM [33] is a hardware ORAM with small client storage, integrity verification, or encryption units. With these attributes, Tiny ORAM can be used by a single-chip secure processor to obfuscate its execution to an adversary watching the chip's I/O pins. As a proof of concept, it has been evaluated as the on-chip memory controller of a 25 core processor. Tiny ORAM design takes up 1/36-th the area (1 mm^2 of silicon in 32 nm technology) of the chip, which is roughly equivalent to the area of a single core, and consumes an estimated 112 mW at a 1 GHz clock frequency. With a 128 bits/cycle channel to main memory (roughly equivalent to 2 DRAM channels), Tiny ORAM can complete a 1 GByte non-recursive ORAM lookup for a 512 bit block in ∼

1275 processor cycles (An insecure DRAM access for a 512 bit block takes an average 58 processor cycles).

In the secure processor setting, the only implementation-level treatment of ORAM is a system called Phantom, by Maas et al. [47]. In [33] authors addressed the challenges had left open by the Phantom design. In this chapter, we will present a complete silicon tape-out of Tiny ORAM design - the first for any type of ORAM - and integrate it with general purpose processor cores to create the first single-chip secure processor able to hide its access pattern to main memory.

6.2.1 Design Challenges

In building Hardware ORAM, there are two major challenges areas where new ideas were proposed in [33].

Challenge 1: Position Map Management. The first challenge for hardware ORAM controllers is that they need to store and manage the Position Map (PosMap for short). Recall from prior chapters: the PosMap is a key-value store that maps data blocks to random locations in external memory. Hence, the PosMap's size is proportional to the number of data blocks (e.g., cache lines) in main memory and can be hundreds of MegaBytes in size. This is too large to fit in a processor's on-chip memory.

To more efficiently manage the PosMap (Challenge 1), the following mechanisms were proposed in [33].

1. The **PosMap Lookaside Buffer**, or PLB for short, a mechanism that significantly reduces the memory bandwidth overhead of Recursive ORAMs depending on underlying program address locality.
2. A way to compress the PosMap, which reduces the cost of recursion and improves the PLB's effectiveness.
3. A new ORAM integrity scheme, called PosMap MAC or PMMAC for short, which is extremely efficient in practice and is asymptotically optimal.

With the PLB and PosMap compression, PosMap-related memory bandwidth overhead reduced by 95%, overall ORAM bandwidth overhead reduced by 37% and SPEC performance improved by 1.27×. As a standalone scheme, PMMAC reduces the amount of hashing needed for integrity checking by ≥ 68× relative to prior schemes. Using PosMap compression and PMMAC as a combined scheme, an integrity checking mechanism for ORAM increases performance overhead by only 7%.

Challenge 2: Throughput with Large Memory Bandwidth. The second challenge in designing ORAM in hardware is exactly how to maximize data throughput for a given memory (e.g., DRAM) bandwidth. For a given memory bandwidth, the factor limiting

data throughput should be the memory. Yet, as shown by the Phantom design, this may not be the case because of other factors (such as processor area constraints, etc).

To improve design throughput for high memory bandwidths (Challenge 2), the following mechanisms were proposed in [33].

1. A **subtree** layout scheme to improve memory bandwidth of tree ORAMs implemented over DRAM.
2. A **bit-based stash management** scheme to enable small block sizes. When implemented in hardware, Tiny ORAM scheme removes the block size bottleneck in the Phantom design.
3. A new ORAM scheme called **RAW ORAM**, derived from Ring ORAM [11], to reduce the required encryption engine bandwidth.

The subtree layout scheme ensures that over 90% of available DRAM bandwidth is actually used by Tiny ORAM. The stash management scheme prevents a performance bottleneck in Phantom when the block size is small, and allows Tiny ORAM to support any reasonable block size (e.g., from 64-4096 Bytes). In particular: with a 64 Byte block size, Tiny ORAM improves access latency by $\geq 40\times$ in the best case compared to Phantom. On the other hand, RAW ORAM reduces the number of encryption units by $\sim 3\times$ while maintaining comparable bandwidth to the original design.

6.2.2 Frontend

In this section we present mechanisms to optimize the PosMap. The techniques in this section only impact the Frontend and can be applied to any Position-based ORAM Backend (such as [3, 4, 12]).

6.2.2.1 PosMap Lookaside Buffer

Considering Recursive ORAM as a multi-level page table for ORAM, a natural optimization is to cache PosMap blockes so that LLC accesses exhibiting program address locality require less PosMap ORAM accesses on average. This idea is the essence of the PosMap Lookaside Buffer, or PLB, whose name obviousliy originates from the Translation Lookaside Buffer (TLB) in conventional systems. Unfortunately, unless case is taken, this idea totally breaks the security of ORAM. In this section, fixes of the security holes are presented.

PLB Caches. The blocks in PosMap ORAMs contain a set of leaf labels for consecutive blocks in the next ORAM. Given this fact, some PosMap ORAM lookups can be eliminated by adding a hardware cache to the ORAM Frontend calles the PLB. Suppose the LLC requests block a_0 at some point. PosMap ORAM block needs from $ORam_i$ for a_0 has address $a_i = a_0 / X^i$. If this PosMap block is in the PLB when block a_0 is requested, the ORAM controller has the leaf needed to lookup $ORam_{i-1}$, and can skip $ORam_i$ and

all the smaller PosMap ORAMs. Otherwise, block a_i is retrieved from $ORam_i$ and added to the PLB. When block a_i is added to the PLB, another block may have to be evicted in which case it is appended to the stash of the corresponding ORAM. A minor but important detail is that a_i may be valid address for blocks in multiple PosMap ORAMs; to disambiguate blocks in the PLB, block a_i is stored with the tag $i \parallel a_i$ where $\parallel$ denotes bit concatenation.

PLB (In)security. Unfortunately, since each PosMap ORAM is stored in a different physical ORAM tree and PLB hits/misses correlate directly to a program's access pattern, the PosMap ORAM access sequence leaks the program's access pattern. To show how this breaks security, consider two example programs in a system with one PosMap ORAM $ORam_1$ (whose blocks store $X = 4$ leaves) and a Data ORAM $ORam_0$. Program A unit strides through memory (e.g., touches a, a + 1, a + 2,…). Program B scans memory with a stride of X (e.g., touches a, a + X, a + 2 X,…). For simplicity, both programs make the same number of memory accesses. Without the PLB, both programs generate the same access sequence, namely: 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, … where 0 denotes an access to $ORam_0$, and 1 denotes an access to $ORam_1$. However, with the PLB, the adversary sees the following access sequences (0 denotes an access to $ORam_0$ on a PLB hit):

Program A: 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, …

Program B: 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, …

Program B constantly misses in the PLB and needs to access $ORam_1$ on every access. Clearly, the adversary can tell program A apart from program B in the PLB-enabled system.

Security Fix: Unified ORAM tree. To hide PosMap access sequence, Recursive ORAM should be changed such that all PosMap ORAMs and the Data ORAM store blocks in the same physical tree which will be denoted $ORam_U$. Organizationally, the PLB and on-chip PosMap become the new Path ORAM Frontend, which interacts with a single ORAM Backend. Security-wise, both programs from the previous section access only $ORam_U$ with the PLB and the adversary cannot tell them apart.

TLB vs. PLB. While a traditional TLB caches single address transactions, the PLB caches entire PosMap blocks. The address locality exploited by both structures, however, is the same.

6.2.2.2 PosMap Compression

In this section is shown how to compress the PosMap using pseudorandom functions (PRFs). The high level goal is to store more leaves per PosMap block, thereby reducing the number of the Recursive PosMaps. However, this scheme by itself does not dramatically improve performance.

Main Idea. Following previous notation, suppose each PosMap block contains X leaf labels for the next ORAM. For example, some PosMap block contains leaf labels for the blocks with addresses {a, a + 1, …, a + X − 1}. With the compressed PosMap scheme, the PosMap block's contents are replaced with an α-bit group counter (GC) and X β-bit individual counters (IC):

With this format, the current leaf label can be computed for block a + j through $PRF_K(a + j \parallel GC \parallel IC_j) \bmod 2^L$. Note that with this technique, the on-chip PosMap is unchanged and still stores an uncompressed leaf per entry.

Block Remap. For PRFK() to generate a uniform random sequence of leaves, it must be ensured that each $GC \parallel IC_j$ strictly increases (i.e., the PRFK() must never see the same input twice). This was achieved by the following modified remapping operation:

When remapping block a + j, the ORAM controller first increments its individual counter IC_j. If the individual counter rolls over (becomes zero again), the ORAM controller will increment the group counter GC. This changes the leaf label for all the blocks in the group, so we have to read each block through the Backend, reset its individual counter and remap it to the updated path given by $PRF_K(a + j \parallel GC + 1 \parallel 0) \bmod 2^L$. In the worst-case where the program always requests the same block in a group, it is necessary to reset X individual counters in the group every 2^β accesses. This reset operation is very expensive for baseline Recursive ORAM. In that case, the ORAM controller must make X full Recursive ORAM accesses to reset the individual counters in a certain PosMap ORAM block. Otherwise, it reveals that individual counters in a certain PosMap ORAM block. Otherwise, it reveals that individual counters have overflown in that certain ORAM, which is related to the access pattern. On the other hand, using a single Unified ORAM tree as is done to support the PLB reduces this to X accesses to $ORam_U$.

System Impact and the PLB. The compressed PosMap format can be used with or without a PLB and, like the PLB, does not require changes to the Backend. That is, PosMap blocks are stored in their compressed format inside the PLB and ORAM tree/Backend. Uncompressed leaves are generated using the PRF on-demand by the Frontend. Each block stored in the Backend or ORAM tree is still stored alongside its uncompressed leaf label (a one-time cost per block), to facilitate ORAM evictions.

Benefit of Compressed Format. The scheme that proposed in [33] compresses the PosMap block by setting, α, β and X such that $\alpha/X + \beta < L$, implying that the (amortized) bits needed to store each leaf has decreased. A larger X means a fewer number of PosMap ORAMs are needed. Further, this scheme improves the PLB's hit rate since more blocks are associated with a given PosMap block. For concreteness, suppose the ORAM block size in bits is B = 512. The compressed PosMap scheme enables $X' = 32$ by setting α = 64 and β = 14, regardless of ORAM tree depth L. In this configuration, the worst case block remap overhead is $X'/2^{\beta} = .2\%$. By comparison, the original PosMap representation only achieves $X = 16$ for ORAM tree depths of $L = 17$ to $L = 32$.

6.2.2.3 PosMap MAC

In this section is described a novel and simple integrity verification scheme for ORAM called PosMAp MAC or PMMAC, that is facilitated by PosMap compression technique from the previous section. PMMAC achieves asymptotic improvements in hash bandwidth over prior schemes and is easy to implement in hardware.

Main Idea and Non-Recursive PMMAC. Clearly, any memory system including ORAM that requires integrity verification can implement the replay-resistant MAC scheme by storing per-block counters in a tamper-proof memory. Unfortunately, the size of this memory is even larger than the original ORAM PosMap making the scheme untenable. If PosMap entries are represented as non-repeating counters, as is the case with the compressed PosMap (Section 6.2.2.2), the replay-resistant MAC scheme can be implemented without additional counter storage.

Firstly, described PMMAC without recursion and with simple/flat counters per-block to illustrate ideas. Suppose block α which has data d has access count c. Then, the on-chip PosMap entry for block α is c and we generate the leaf l for block α through $l = PRF_K(a\|c)\ mod 2^L$. Block α is written to the Backend as the tuple (h, d) where

$$h = MAC_K(a\|c\|d)$$

When block α is read, the Backend returns $(h\star, d\star)$ and PMMAC performs the following check to verify authenticity/freshness:

$$\text{Assert } h\star == MAC_K(a\|c\|d\star)$$

where $\star$ denotes values that may have been tampered with. After the assertion is checked, c is incremented for the returned block.

Security follows if it is infeasible to tamper with block counters and no counter value for a given block is ever repeated. The first condition is clearly satisfied because the

counters are stored on-chip. The second condition is satisfied by making each counter wide enough to not overflow (e.g. 64 bits wide).

PMMAC requires no change to the ORAM Backend because the MAC is treated as extra bits appended to the original data block. As with PosMap compression, the leaf currently associated with each block in the stash/ORAM tree is stored in its original (uncompressed) format.

Adding Recursion and PosMap Compression. To support recursion, PosMap blocks (including on-chip PosMap entries) may contain either a flat (64 bits) or compressed counter (Section 6.2.2.2) per next-level PosMap or Data ORAM block. As in the non-Recursive ORAM case, all leaves are generated via a PRF. The intuition for security is that the tamper-proof counters in the on-chip PosMap form the root of trust and then recursively, the PosMap blocks become the root of trust for the next level PosMap or Data ORAM blocks. Note that in the compressed scheme, and the components of each counter are already sized so that each block's count never repeats/overflows. It is given a formal analysis for security with Recursive ORAM in the next Section 6.2.2.4.

For realistic parameters, the scheme that uses flat counters in PosMap blocks incurs additional levels of recursion. For example, using B = 512 and 64 bit counters we have $X = B/64 = 8$. Importantly, with the compressed PosMap scheme we can derive each block counter can be derived from $BG\ and\ IC_j$ (Section 6.2.2.2) without adding levels of recursion or extra counter storage.

Key Advantage: Hash Bandwidth and Parallelism. Combined with PosMap compression, the overheads for PMMAC are the bits added to each block to store MACs and the cost to perform cryptographic hashes on blocks. The extra bits per block are relatively low-overhead – the ORAM block size is usually 64-128 Bytes and each MAC may be 80-128 bits depending on the security parameter. To perform a non-Recursive ORAM access (i.e., read/write a single path), Path ORAM reads/writes $O(logN)$ blocks from external memory. Merkle tree constructions [20, 28] need to integrity verify all the blocks on the path to check/update the root hash. Crucially, PMMAC construction only needs to integrity verify (check and update) 1 block – namely the block of interest – per access, achieving an asympotic reduction in hash bandwidth.

To give some concrete numbers, assume $Z = 4$ block slots per ORAM tree bucket following [2, 47]. Then, there are $Z * (L + 1)$ blocks per path in ORAM tree, and this construction reduces hash bandwidth by $68\times$ for $L = 16$ and by $132\times$ for $L = 32$. It is not included the cost of reading sibling hashes for the Merkle tree for simplicity.

Integrity verifying only a single block also prevents a serialization bottleneck present in Merkle tree schemes. Consider the scheme from [20], a scheme optimized for Path

ORAM. Each hash in the Merkle tree node must be recomputed based on the contents of the corresponding ORAM tree bucket and its child hashes, and is therefore fundamentally sequential. If this process cannot keep up with memory bandwidth, it will be the system's performance bottleneck.

Adding Encryption: Subtle Attacks and Defenses. Up to this point PMMAC has been discussed in the context of providing integrity only. ORAM must also apply a probabilistic encryption scheme (assuming AES counter mode as done in [7]) to all data stored in the ORAM tree. In this section is shown how the encryption scheme of [7] breaks under active adversaries because the adversary is able to replay the one-time pads used for encryption. ([7] presented an integrity very cation scheme based on Merkle trees to prevent such attacks.) It is shown how PMMAC doesn't prevent this attack by default and then provide a fix that applies to PMMAC.

Firstly is shown that the scheme used by [7] for reference: Each bucket in the ORAM tree contains, in addition to Z encrypted blocks, a seed used for encryption (the BucketSeed) that is stored in plaintext. (BucketSeed is synonymous to the “counter" in AES counter mode.) If the Backend reads some bucket whose seed is BucketSeed, the bucket will be re-encrypted and written back to the ORAM tree using the one-time pad (OTP) $AES_K(BucketID \| BucketSeed + 1 \| i)$, where i is the current chunk of the bucket being encrypted.

The above encryption scheme breaks privacy under PMMAC because PMMAC doesn't integrity verify BucketSeed. For a bucket currently encrypted with the pad $P = AES_K(BucketID \| BucketSeed \| i)$, suppose the adversary replaces the plaintext bucket seed to BucketSeed – 1. This modification will cause the contents of that bucket to decrypt to garbage, but won't trigger an integrity violation under PMMAC unless bucket BucketID contains the block of interest for the current access. If an integrity violation is not triggered, due to the replay of BucketSeed, that bucket will next be encrypted using the same one-time pad P again.

Replaying one-time pads obviously causes security problems. If a bucket re-encrypted with the same pad P contains plaintext data D at some point and D' at another point, the adversary learns $D \oplus D'$. If D is known to the adversary, the adversary immediately learns D' (i.e., the plaintext contents of the bucket).

The fix for this problem is relatively simple: To encrypt chunk i of a bucket about to be written to DRAM, the pad $P = AES_K(GlobalSeed \| i)$ will be used, where $GlobalSeed$ is now a single monotonically increasing counter stored in the ORAM controller in a dedicated register (this is similar to the global counter scheme in [48]). When a bucket is encrypted, the current $GlobalSeed$ is written out alongside the bucket as before and $GlobalSeed$ (in the ORAM controller) is incremented. Now it's

easy to see that each bucket will always be encrypted with a fresh OTP which defeats the above attack.

6.2.2.4 Security Analysis

We now give a security analysis for the PLB and PMMAC schemes.

6.2.2.4.1 PosMap Lookaside Buffer

It is given a proof sketch that PLB+Unified ORAM tree construction achieves satisfies **Definition 2**. To do this, is used the fact that the PLB interacts with a normal Path ORAM Backend. The following observations will be used to argue security:

Observation 1. If all leaf labels l_i used in {read, write, readrmv} calls to Backend are random and independent of other l_j for $i \neq j$, the Backend achieves the security of the original Path ORAM.

Observation 2. If the append is always preceded by a readrmv, stash overflow probability does not increase (since the net stash occurancy is unchanged after both operations).

Theorem 7. The PLB+Unified ORAM tree scheme reduces to the security of the ORAM Backend.

You could find the detailed Proof in [33]. Of course, the PLB may further influence the ORAM trace length (the number of calls to Access for a given Z) by filtering out some calls to Backend for PosMap blocks. Now the trace length is determined by, and thus reveals, the sum of LLC misses and PLB misses. The processor cache and the PLB are both on-chip and outside the ORAM Backend, so adding a PLB is the same (security-wise) to adding more processor cache: in both cases, only the total number of ORAM accesses leaks. By comparison, using a PLB without a Unified ORAM tree leaks the set of PosMap ORAMs needed on every Recursive ORAM access, which makes leakage grow linearly with the trace length.

6.2.2.4.2 PosMap MAC (Integrity)

It is shown that breaking the integrity verification scheme is as hard as breaking the underlying MAC.

Observation 3. If the first $k-1$ address and counter pairs $(a_i,\ c_i)$' s the Frontend receives have not been tampered with, then the Frontend seeds a MAC using a unique $(a_k,\ c_k)$, i.e., $(a_i,\ c_i) \neq (a_k,\ c_k)$ for $1 \leq i \leq k$. This further implies $(a_i,\ c_i) \neq (a_j,\ c_j)$ for all $1 \leq i < j \leq k$.

This property can be seen directly from the algorithm description, with or without the PLB and/or PosMap compression. For every a, we have a dedicated counter, sourced from the on-chip PosMap or the PLB, that increments on each access. If we use PosMap compression, each block counter will either increment (on a normal access) or jump to the next multiple of the group counter in the event of a group remap operation. Thus, each address and counter pair will be different from previous ones. The Observation 3 is used to be proven the security of the integrity scheme.

Theorem 8. Breaking the PMMAC scheme is as hard as breaking the underlying MAC scheme.

You could find the detailed Proof in [33].

6.2.2.4.3 PosMap MAC (Privacy)

The system's privacy guarantees require certain assumptions under PMMAC because PMMAC is an authenticate-then-encrypt scheme [49]. Since the integrity verifier only check the block of interest returned to the Frontend, other (tampered) data on the ORAM tree path will be written to the stash and later be written back to the ORAM tree. For example, if the adversary tampers with the block-of-interest's address bits, the Backend won't recognize the block and won't be able to send any data to the integrity verifier (clearly an error). The adversary may also coerce a stash overflow by replacing dummy blocks with real blocks or duplicate blocks along a path. To address these cases, certain assumptions are necessary about how the Backend will possibly behave in the presence of tampered data. It is required a correct implementation of the ORAM Backend to have the following property:

Property 1. If the Backend makes an ORAM access, it only reveals to the adversary (a) the leaf send by the Frontend for that access and (b) a fixed amount of encrypted data to be written back to the ORAM tree.

If **Property 1** is satisfied, it is straightforward to see that any memory request address trace generated by the Backend is indistinguishable from other traces of the same length. That is, the Frontend receives tamper-proof responses (by Theorem 8) and therefore produces independent and random leaves. Further, the global seed scheme trivially guarantees that the data written back to memory gets a fresh pad.

If **Property 1** is satisfied, the system can still leak the ORAM request trace length; i.e., when an integrity violation is detected, or when the Backend enters an illegal state. Conceptually, an integrity violation generates an exception that can be handled by the processor. When that exception is generated and how it is handled can leak some privacy. For example, depending on how the adversary tampered with memory, the violation may be detected immediately or after some period of time depending on

whether the tampered bits were of interest to the Frontend. Quantifying this leakage is outside the scope.

6.2.3 Backend

Now several mechanisms are presented to improve the ORAM Backend's throughput when memory bandwidth is high. The techniques in this section only impact the Backend and can be applied regardless of optimizations from the previous section 6.2.2.

The issue for Tree ORAMs (like Path ORAM) implemented over DRAM is that to be secure, ORAM accesses inherently have low spatial locality in memory. Yet, achievable throughput in DRAM depends on spatial locality: bad spatial locality means more DRAM row buffer misses which means time delay between consecutive accesses. Indeed, when naively storing the Path ORAM tree into an array, two consecutive buckets along the same path hardly have any locality, and it can be expected that row buffer hit rate would be low. The following technique that introduced in [33] can improve Path ORAM's performance on DRAM.

Each subtree is packed with k levels together, and are treated as the nodes of a new tree, a $2k$ -ary tree with $\left\lceil \frac{L+1}{k} \right\rceil$ levels. Figure 44 is an example with k = 2. It was adopted the address mapping scheme in which adjacent addresses first differ in channels, then columns, then banks, and lastly rows. The node size was set of the new tree to be the row buffer size times the number of channels, which together with the original bucket size determines k.

Performance impact. With commercial DRAM DIMMs, k = 6 or k = 7 is possible which allows the ORAM to maintain 90 – 95% of peak possible DRAM bandwidth for every parameterization. Without the technique, achievable bandwidth may be < 50% depending on the data block size, recursion scheme used, number of DRAM channels, and other parameters. It should be noted that Phantom was able to achieve 94% of peak DRAM bandwidth [47] without the subtree packing technique as there was sufficient spatial locality given their large 4 KByte block size.

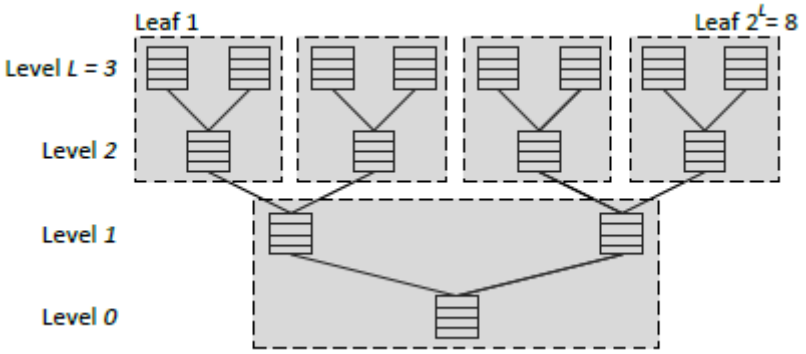

Figure 41: Illustration of subtree locality

6.2.3.1.1 Stash Management

Deciding where to evict each block in the stash is a challenge for Path ORAM hardware designs. Conceptually, this operation tries to push each block in the stash as deep (towards the leaves) into the ORAM tree as possible while keeping to the invariant that blocks can only live on the path to their assigned leaf.

Phantom constructs an FPGA-optimized heap sort on the stash [47]. Unfortunately, this approach creates a performance bottleneck because the initial step of sorting the stash takes multiple cycles per block. For example, in a Phantom design, adding a block to the heap takes 11 cycle. If the ORAM block size and memory bandwidth is such that writing a block to memory takes less than 11 cycles, system performance is bottlenecked by the heap-sort-based eviction logic and not by memory bandwidth.

In [33] proposed a new and simple stash eviction algorithm based on bit-level hardware tricks that takes a single cycle to evict a block and can be implemented efficiently in FPGA logic. This eliminates the above performance overhead for any practical block size and memory bandwidth.

PushToLeaf With Bit Tricks. The PushToLeaf() routine, is shown in Figure 42: PushToLeaf Algorithm. PushToLeaf($Stash, l$) is run once during each ORAM access and populates an array of pointers occ. Stash can be thought of as a single-ported RAM that stores data blocks and their metadata. Once populated, $occ[i]$ points to the block in Stash that will be written back to the $i\text{-}th$ position along $P(l)$. Thus, to complete the ORAM eviction, a hardware state machine sends each block given by $Stash[occ[i]]$ for $i = 0, \dots, (L+1) * Z - 1$ to be encrypted and written to external memory.

Notations. Suppose l is the current leaf being accessed. The leaves are represented as L-bit words which are read right-to-left: the $i\text{-}th$ bit indicates whether path l traverses the $i\text{-}th$ bucket's left child (0) or right child (1). On Line 3, each entry of $occ\ is\ set\ to\ \perp$, to indicate that the eviction path is initially empty. Occupied is an $L+1$ entry memory that records the number of real blocks that have been pushed back to each bucket so far.

Bit operation-based stash scan. $2C$ stands for two's complement arithmetic

1: **Inputs:** the current leaf l being accessed
2: **function** PushToLeaf($Stash, l$)
3: $occ \leftarrow \{\perp\ for\ i = 0, \dots, (L+1) * Z - 1\}$
4: Occupied $\leftarrow \{0\ for\ i = 0, \dots, L\}$
5: **for** $i \leftarrow 0\ to\ T + L * Z - 1$ **do**
6: $(a, l_i, D) \leftarrow Stash[i]$ //Leaf assigned to $i\text{-}th$ block
7: $level \leftarrow PushBack(l, l_i, Occupied)$
8: **if** $a \neq \perp\ and\ level > -1$ ***then***
9: $offset \leftarrow level * Z + Occupied[level]$
10: $occ[offset] \leftarrow i$
11: $Occupied[level] \leftarrow Occupied[level] + 1$
12: **end if**
13: **end for**
14: **end function**
15: **function** PushBack($l, l', Occupied$)
16: $t_1 \leftarrow (l \oplus l' \| 0)$ //Bitwise XOR
17: $t_2 \leftarrow t_1\ \&\ -t_1$ //Bitwise AND, $2C$ negation
18: $t_3 \leftarrow t_2 - 1$ //$2C$ sutraction
19: $full \leftarrow \{(Occupied[1] = Z) for\ i = 0\ to\ L\}$
20: $t_4 \leftarrow t_3\ \&\ \sim full$ //Bitwise AND/negation
21: $t_5 \leftarrow reverse(t_4)$ //Bitwise reverse
22: $t_6 \leftarrow t_5\ \&\ -t_5$
23: $t_7 \leftarrow reverse(t_6)$
24: ***if*** $t_7 = 0$ ***then***
25: **return** -1 //Block is stuck in stash
26: ***end if***
27: **return** $log_2(t_7)$ //Note: t_7 must be one-hot
28: **end function**

Figure 42: PushToLeaf Algorithm

PushBack(). The core operation in this proposal is the PushBack() subroutine, which takes as input the path l we are evicting to, the path l' a block in the stash is mapped to, and outputs which level on path l that block should get written back to.

Security. While the stash eviction procedure is highly-optimized for hardware implementation, it is algorithmically equivalent to the original stash eviction procedure with Path ORAM. Thus, security follows from the original Path ORAM analysis.

Hardware Implementation and Pipelining. The algorithm above runs $T + (L+1)Z$ iterations of PushBack per ORAM access, where T is the stash size not counting the path length. In hardware, the Algorithm in *Figure 42* is pipelined in three respects to hide its latency.

6.2.3.1.2 Reducing Encryption Bandwidth

Another serious problem for ORAM design is the area needed for encryption units. All data touched by ORAM must get decrypted and re-encrypted to preserve privacy.

Encryption bandwidth hence scales with memory bandwidth and quickly becomes the area bottleneck. To address this problem in [33] it was proposed a new ORAM design, which is called RAW ORAM, optimized to minimize encryption bandwidth at the algorithmic and engineering level.

RAW ORAM Algorithm. RAW ORAM is based on Ring ORAM [11] and splits ORAM Backend operations into two flavors: **ReadPath** and **EvictPath** accesses. **ReadPath** operations perform the minimal amount of work needed to service a client processor's read/write requests (i.e., last level cache misses/writebacks) and EvictPath accesses perform evictions (to empty the stash) in the background. To reduce the number of encryption units needed by ORAM, in [33] **ReadPath** accesses were optimized to only decrypt the minimal amount of data needed to retrieve the block of interest, as opposed to the entire path. **EvictPath** accesses require more encryption/decryption, but occur less frequently.

Parameter A. Like Ring ORAM [11], RAW ORAM uses the parameter A, set at system boot time. For a given A, RAW ORAM obeys a strict schedule that the ORAM controller performs one EvictPath access after every A reads.

Security. The security analysis is very similar (and simpler, even) to that in Ring ORAM [11]. ReadPath accesses always read paths in the ORAM tree at random, just like Path ORAM [2]. Further, EvictPath accesses occur at predetermined times and are to predictable/data-independent paths.

Performance and Area Characteristics. Assume for simplicity that the bucket header is the same size as a data block. Then, each ReadPath access reads $(L+1)Z$ blocks on the path, but only decrypts 1 block; it also reads/writes and decrypts/re-encrypts the $L+1$ headers/blocks. An EvictPath reads/writes and decrypts/re-encrypts all the $(L+1)(Z+1)$ blocks on a path. Thus, in RAW ORAM the relative memory bandwidth per bucket is $Z+2+\frac{2(Z+1)}{A}$, and the relative encryption bandwidth per bucket is roughly $1+\frac{2(Z+1)}{A}$. In Figure 43, virtualized the relative memory and encryption bandwidth of RAW ORAM with different parameter settings that have been shown [11] to give negligible stash overflow probability. Based on this $Z=5, A=5$ $(Z5A5)$ is a good trade-off as it achieves 6% memory bandwidth improvemtns and $\sim 3\times$ encryption reduction over Path ORAM.

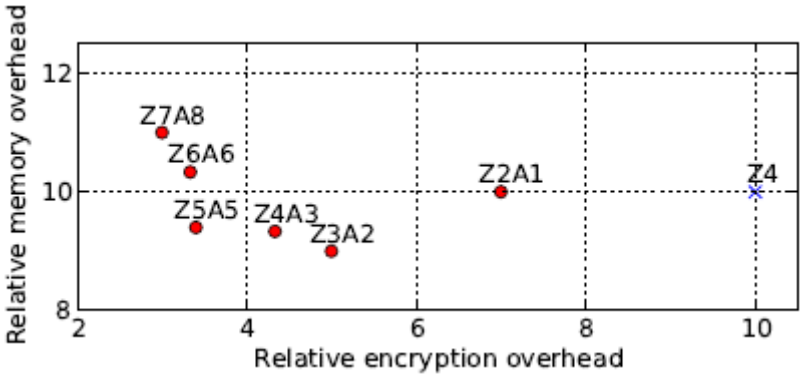

Figure 43: The relative memory and encryption bandwidth overhead of RAW ORAM

Exploiting Path Eviction Predictability. Despite RAW ORAM's theoretic area savings for encryption units, careful engineering is needed to prevent that savings from turning into performance loss. The problem is that by reducing encryption units (i.e., AES) to provide “just enough" bandwidth for ReadPath accesses, it is forced to wait during EvictPath accesses for that reduced number of AES units to finish decrypting/re-encrypting the entire path. Further, since all AES IVs are stored externally with each bucket, the AES units can't start working on a new EvictPath until that access starts.

To remove the above bottleneck while maintaining the AES unit reduction, authors in [33] made the following key observation: Since EvictPath operations occur in a predictable, fixed order, they can determine exactly how many times any bucket along any path has been written in the past.

Using eviction predictability, could be pre-computed the AES-CTR initialization vector IV_1. Simply put, this means the AES units can do all decryption/encryption work for EvictPath accesses “in the background" during concurrent ReadPath accesses. To decrypt the $i\text{-}th$ 128-bit ciphertext chunk of the bucket with unique ID BucketID at level j in the tree, it is XOR with the following mask: $AES_K(g_j \| BucktID \| i)$ where g_j is the bucket eviction count defined above. Correspondingly, re-encryption of that chunk is done by generating a new mask where the write count has been incremented by 1. With this scheme, g_j takes the place of IV_1 and since g_j can be derived internally, so it is not necessary to store it externally.

On both **ReadPath** and **EvictPath** operations, must be decrypted the program addresses and valid bits of all blocks in each bucket. For this could be applied the global counter scheme from Section PosMap MAC6.2.2.3 or used the mask as in Ren et al. [7], namely $AES_K(IV_2 \| BucktID \| i)$, where IV_2 is stored externally as part of each bucket's header.

At the implementation level, an AES core was time-multiplexed between generating masks for IV_1 and IV_2. The AES core prioritizes IV_2 operations; when the core is not servicing IV_2 requests, it generates masks for IV_1 in the background and stores them in a FIFO.

6.2.4 Evaluation (FPGA Prototype)

Now is described a hardware prototype of Tiny ORAM on a Virtex-7 VC707 FPGA board and analyze its area and performance characteristics. The main reason for hardware prototyping is to tape-out in ASIC. With that in mind, the FPGA evaluation has two primary objectives. First, to compare against Phantom (which was optimized for FPGA) in as apples-to-apples a comparison as possible. Second, to demonstrate that the design is working under `high memory bandwidth' conditions.

The entire design (as well as the extension to ASIC) is open source at http:// kwonalbert.github.io/oram.

6.2.4.1.1 Metrics and Baselines

The entire design was written in plain Verilog and was synthesized using the Xilinx Vivado flow (version 2013.4). Performance was measured as the latency (in FPGA cycles or real time) between when an FPGA user design requests a block and Tiny ORAM returns that block. Area is calculated in terms of FPGA lookup-tables (LUT), flip-flops (FF) and Block RAM (BRAM), and is measured post place-and-route (i.e., represents final hardware area numbers). For the rest of the section BRAM is counted in terms of 36 Kbit BRAM.

Tiny ORAM is compared with two baselines shown in *Table 6*. The first one is Phantom [47], which normalized to Tiny ORAM capacity and the 512 bits/cycle DRAM bandwidth of this particular VC707 board. Further, Phantom's tree top caching is disabled. Phantom's performance/area numbers are taken/approximated from the figures in their paper. The second baseline is a basic Path ORAM with the stash management technique that described above, to show the area saving of RAW ORAM.

Design	Phantom	Path ORAM	RAW ORAM
Parameters			
Z, A	4, N/A	4, N/A	5, 5
Block size	4 KByte	64 Byte	64 Byte
# of tiny aes cores	4	4	1
Performance (cycles)			
Access 64 B	$\sim$ 12000	270	276
Access 4 KB	$\sim$ 12000	17280	17664
ORAM Backend Area			
LUT (%)	$\sim$ 6000 + 11460	18977 (7%)	14427 (5%)
FF (%)	not reported	16442 (3%)	11359 (2%)
BRAM (%)	$\sim$ 172 + 344	357 (34%)	129 (13%)
Total Area (Backend+Frontend)			
LUT (%)	$\sim$ 10000 + 11460	22775 (8%)	18381 (6%)
FF (%)	not reported	18252 (3%)	13298 (2%)
BRAM (%)	$\sim$ 235 + 344	371 (36%)	146 (14%)

Table 6: Comparison of Tiny ORAM and two Baselines

6.2.4.1.2 Implementation

Organization. The design was built hierarchically as three main components: the Frontend, stash (Backend) and AES units used to decrypt/re-encrypt paths (Backend). We evaluate both Path ORAM and RAW ORAM Backend designs (Section 6.2.3.1.1). The Path ORAM Backend is similar to the Phantom Backend.

Unlike Phantom, this design does not have a DRAM buffer (see [47]). If such a structure is needed it should be much smaller than that in Phantom (<10 Kbytes as opposed to hundreds of KBytes) due to the 64 Byte block size.

Parameterization. Both of the designs (Path ORAM and RAW ORAM) use B = 512 bits per block and L = 20 levels. The choice of B = 512 (64 Bytes) shows that Tiny ORAM can run even very small block sizes without imposing hardware performance bottlenecks. There is a constraint to set L = 20 because this setting fills the VC707's 1 GByte DRAM DIMM.

The Frontend which was evaluated is P_X16. Also, it was not evaluated the cost of integrity (PMMAC) in the FPGA prototype as integrity was not considered by the Phantom design and does not impact memory throughput.

Clock regions. The DRAM controller on the VC707 board runs at 200 MHz and transfers 512 bits/cycle. To ensure that DRAM is Tiny ORAM's bottleneck, the design's timing was optimized to run at 200 MHz.

DRAM controller. The interface is a DDR3 DRAM through a stock Xilinx on-chip DRAM controller with 512 bits/cycle throughput. From when a read request is presented to

the DRAM controller, it takes ~30 FPGA cycles to return data for that read (i.e., without ORAM). The DRAM controller pipelines requests. That is, if two reads are issued in consecutive cycles, two 512 bit responses arrive in cycle 30 and 31. As mentioned before, the subtree layout scheme allows to achieve near-optimal DRAM bandwidth.

Encryption. "Tiny aes" is used, a pipelined AES core that is freely downloadable from http://opencores.org/. Tiny aes has a 21 cycle latency and produces 128 bits of output per cycle. One tiny aes core costs 2865/3585 FPGA LUT/FF and 86 BRAM. To implement the time-multiplexing scheme from Section 6.2.3.1.2, is simply added state to track whether tiny aes's output (during each cycle) corresponds to IV_1 or IV_2.

Given the DRAM bandwidth, RAW ORAM requires 1.5 (has to be rounded to 2) tiny aes cores to completely hide mask generation for EvictPath accesses at 200 MHz. To reduce area further, the design was optimized to run tiny aes and associated control logic at 300 MHz. Thus, the final design requires only a single tiny aes core. Basic Path ORAM would require 3 tiny aes cores clocked at 300 MHz, which matches the 3× AES saving in the analysis from Section 6.2.3.1.2. The tiny aes clock was not optimized for basic Path ORAM, and used 4 of them running at 200 MHz.

6.2.4.1.3 Access Latency Comparison

For the rest of the FPGA evaluation, all access latencies are averages when running on a live hardware prototype. *Table 6* gives a summary of results. RAW ORAM Backend can finish an access in 276 cycles ($1.4\mu s$) on average. This is very close to basic Path ORAM; it is not got the 6% theoretical performance improvement because of the slightly more complicated control logic of RAW ORAM.

After normalizing to the DRAM bandwidth and ORAM capacity that presented in Section 6.2, Phantom should be able to fetch a 4 KByte block in $\sim 60\mu s$. This shows the large speedup potential for small blocks. Suppose the program running has bad data locality (i.e., even though Phantom fetches 4 KBytes, only 64 Bytes are touched by the program). In this case, Tiny ORAM using a 64 Byte block size improves ORAM latency by 40× relative to Phantom with a 4 KByte block size. Phantom was run at 150 MHz: if optimized to run at 200 MHz like the current design, the improvement is $\sim 60\times$. Even with perfect locality where the entire 4 KByte data is needed, using a 64 Byte block size introduces only $1.5 - 2\times$ slowdown relative to the 4 KByte design.

6.2.4.1.4 Hardware Area Comparison

In *Table 4*, is shown that the RAW ORAM Backend requires only a small percentage of the FPGA's total area. The slightly larger control logic in RAW ORAM dampens the area reduction from AES saving. Despite this, RAW ORAM achieves an $\geq 2\times$ reduction in

BRAM usage relative to Path ORAM. Note that Phantom [47] did not implement encryption: this area was extrapolated by adding 4 tiny aes cores to their design and estimate a BRAM savings of 4× relative to RAW ORAM.

6.2.4.1.5 Full System Evaluation

Now evaluate a complete ORAM controller by connecting RAW ORAM Backend to the optimized ORAM Frontend. For completeness, a baseline Recursive Path ORAM is implemented and evaluated. To our knowledge, authors in [33] implemented the first form of Recursive ORAM in hardware. They call configurations with the repsective optimized Frontend “Freecursive" to distinguish them from the baseline Frontend. For $L = 20$, 2 PosMap ORAMs were added, to attain a small on-chip position map (< 8 KB).

Figure 44 shows the average memory access latency of several real SPEC06-int benchmarks. Due to optimizations from Section 6.2.2, performance depends on program locality. For this reason, also two synthetic traces were evaluated: scan which has perfect locality and rand which has no locality. Two extreme benchmarks: libq is known to have good locality, and on average the ORAM controller can access 64 Bytes in 490 cycles. Sjeng has bad (almost zero) locality and fetching a 64 Byte block requires ~950 cycles ($4.75\ \mu s$ at 200 MHz). Benchmarks like sjeng reinforce the need for small blocks: setting a larger ORAM block size will strictly decrease system performance since the additional data in larger blocks won't be used.

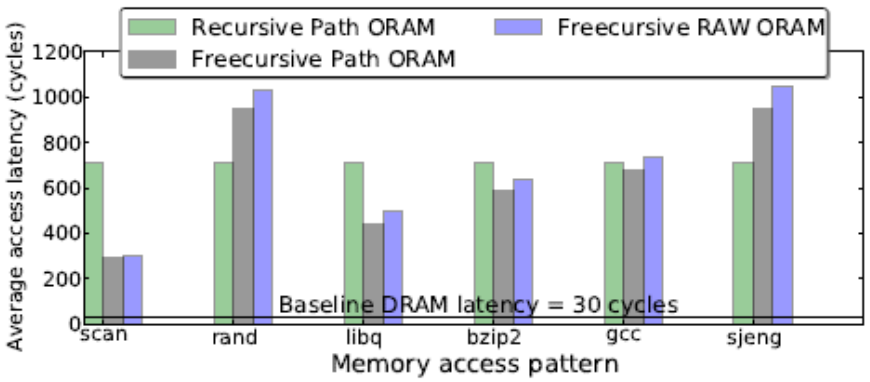

Figure 44: Evaluation between Path ORAM and RAW ORAM

7 Conclusion

Security of data storage is a huge problem in nearly all aspects of the Internet connected world. Consider several ubiquitous settings: outsourced storage, computation outsourcing and the Internet of Things (IoT). The underlying problem is inherent in how programs are written today: to be performant, program control flow and memory access behavior depends on the sensitive information we wish to hide. This thesis studies a cryptographic primitive called Oblivious RAM (ORAM), which provably eliminates all information leakage in memory access patterns [1, 3]. ORAM schemes that are presented make both theoretical and practical contributions. Those schemes are categorized based on schemes characteristics. The 4 main categories are Path ORAM Family, Constant worst-case bandwidth blowup Family, ObliviStore Family, and Applied ORAM schemes. On Chapter 3, Path ORAM [2] and several ORAM schemes, which were based on Path ORAM are presented along with a comparison between them. On Chapter 4, we present ORAM schemes, which achieve Constant worst-case Bandwidth blowup also, we present a comparison between them. The ORAM schemes are Onion ORAM [14] and C – ORAM [22]. On Chapter 5, ObliviStore ORAM [5] and ORAM schemes, which were based on ObliviStore ORAM are presented along with a comparison between them. On Chapter 6, we present two applied ORAM schemes (ObliviSync [50] and Tiny ORAM [33]) that could be used in *real world*. ObliviSync is an oblivious cloud storage system that specifically targets one of the most widely-used personal cloud storage paradigms: synchronization and backup services, popular examples of which are Dropbox, iCloud Drive, and Google Drive. Tiny ORAM is a hardware ORAM with small client storage, integrity verification, or encryption units.

8 Bibliography

[1] O. Goldreich and R. Ostrovsky. Software protection and simulation on Oblivious RAMs. *Journal of the ACM,* 1996.

[2] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu and Srinivas Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. *Cryptology ePrint Archive,* Report2013/180.

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

[4] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O($logN^3$) worst-case cost. In ASIACRYPT, pages 197{214, 2011.

[5] STEFANOV, E., AND SHI, E. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (2013).*

[6] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. Technical report.

[7] REN, L., YU, X., FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Design space exploration and optimization of path oblivious ram in secure processors. In ISCA (2013)

[8] Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Preprint, 2014.

[9] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with O ($\log^2 n$) overhead. CoRR, abs/1307.3699, 2013.

[10] Oblivious Network RAM. D Dachman-Soled, C Liu, C Papamanthou, E Shi, U Vishkin IACR Cryptology ePrint Archive 2015, 73

[11] Constants Count: Practical Improvements to Oblivious RAM. Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. In Usenix Security Symposium, 2015.

[12] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In PETS, 2013.

[13] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In USENIX security, 2014.

[14] DEVADAS, S., VAN DIJK, M., FLETCHER, C. W., REN, L., SHI, E., AND WICHS, D. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, 2015. http://eprint.iacr.org/2015/005.

[15] H. Lipmaa. An Oblivious Transfer protocol with log-squared communication. In ISC, 2005.

[16] Ivan Damgard and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In PKC, 2001.

[17] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In FOCS, 1991.

[18] Ralph C. Merkle. Protocols for public key cryptography. In Oakland, 1980.

[19] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In FAST, 2013.

[20] Ling Ren, Christopher Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for Path Oblivious-RAM. In HPEC, 2013.

[21] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In POPL, 2014.

[22] Tarik Moataz , Travis Mayberry , Erik-Oliver Blass, Constant Communication ORAM with Small Blocksize, Proceedings of the 22nd ACMSIGSAC Conference on Computer and Communications Security, October 12-16, 2015

[23] E. Kushilevitz and R. Ostrovsky. Replication is not Needed: Single Database, Computationally-Private Information Retrieval. In Proceedings of Foundations of Computer Science, pages 364–373, Miami Beach, USA, 1997.

[24] T. Mayberry, E.-O. Blass, and A.H. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In Proceedings of the Network and Distributed System Security Symposium, San Diego, USA, 2014.

[25] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In Proceedings of Information Security Conference, pages 314–328, Singapore, 2005.

[26] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multidimensional trace analysis. In Proc. ACM SOSP (2011).

[27] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In Proc. USENIX ATC (2008), USENIX Association, pp. 213–226.

[28] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In PKC. 2014.

[29] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In FOCS, 2001.

[30] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan. Resizable tree-based oblivious ram. Financial Crypto, 2015.

[31] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In ACM Conference on Computer and Communications Security (CCS), 2014.

[32] Ran Canetti. Security and composition of multiparty cryptographic protocols. Journal of Cryptology, 2000.

[33] FLETCHER, C., REN, L., KWON, A., VAN DIJK, M., STEFANOV, E., SERPANOS, D., AND DEVADAS, S. A low-latency, low-area hardware oblivious ram controller. In FCCM (2015)

[34] O. Goldreich. Towards a theory of software protection and simulation on Oblivious RAMs. In STOC, 1987.

[35] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS, 2015.

[36] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. Commun. ACM, 52(5):91{98, May 2009.

[37] Andrew “bunnie" Huang. Hacking the xbox: An introduction to reverse engineering. 2003.

[38] Markus G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp. IEEE Trans. Comput., 47(10):1153{1157, October 1998.

[39] Sean Gallagher. Your usb cable, the spy: Inside the nsas catalog of surveillance magic. Ars Technica.

[40] Rafal Wojtczuk and Alexander Tereshkin. Attacking intel bios. Blackhat, 2009.

[41] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[42] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, and S. Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. IACR Cryptology ePrint Archive, 2014:431, 2014.

[43] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014. http://eprint.iacr.org/

[44] Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. CW Fletcher, M Naveed, L Ren, E Shi, E Stefanov - IACR Cryptology ePrint Archive, 2015

[45] P. Williams and R. Sion. Single round access privacy on outsourced storage. In CCS, 2012.

[46] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In CCS, 2008.

[47] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In CCS, 2013.

[48] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In MICRO, 2007.

[49] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In CRYPTO, 2001.

[50] ObliviSync: Practical Oblivious File Backup and Synchronization, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. RocheIACR Cryptology ePrint Archive2016.

[51] A. B. Downey. The structural cause of file size distributions. In Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on, pages 361–370, 2001.

[52] ObliviSync github repository. https://github.com/oblivisync/oblivisync.

[53] EncFS. https://vgough.github.io/encfs/

[54] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “Dramsim2: A cycle accurate memory system simulator,” Computer Architecture Letters, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[55] J. Renau, “Sesc: Superescalar simulator,” university of illinois urbana-champaign ECE department, Tech. Rep., 2002. [Online]. Available: http://sesc.sourceforge.net/index.html

[56] Dan Hubbard and Michael Sutton. Top threats to cloud computing v1. 0. Cloud Security Alliance, 2010.

[57] Ewen Macaskill and Gabriel Dance. The nsa files: Decoded. The Guardian, 2013.

[58] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thome, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Beguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In CCS, 2015.

[59] IBM. Ibm 4765 description. Technical report, 2011.

[60] David Grawrock. The Intel Safer Computing Initiative: Building Blocks for Trusted Computing. Intel Press, 2006.

[61] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In CCS, 2009.

[62] Intel. Software guard extensions programming reference. Intel, 2013.

[63] Amazon. Amazon simple storage service developer's guide. Amazon, 2006.

[64] Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In NDSS, 2012.

[65] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In ASPLOS, 2004.

[66] Alon Itai and Michael Slavkin. Detecting data structures from traces. In Workshop on Approaches and Applications of Inductive Programming, 2007.

[67] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In CCS, 2012.

[68] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure processor architecture for encrypted computation on untrusted programs. In STC, 2012.

[69] R. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. Foundations of Secure Computation, 1978.

[70] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In IEEE Symposium on Security and Privacy, 2015.

[71] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In CCS, 2012.

[72] Seny Kamara. How to search on encrypted data: Oblivious rams (part 4). Blog post, 2013.

[73] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. pages 513-524, 2012.

[74] David Cash, Alptekin Kupcu, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In Advances in Cryptology-EUROCRYPT 2013, pages 279-295. Springer, 2013.

[75] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In EUROCRYPT, 2013.