Pre-Print version 2015-11-22

A Practical Oblivious Map Data Structure with Secure Deletion and History Independence

Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi

United States Naval Academy
`[roche,aviv,choi]@usna.edu`

22nd November 2015

Abstract

We present a new oblivious RAM that supports variable-sized storage blocks (vORAM), which is the first ORAM to allow varying block sizes without trivial padding. We also present a new history-independent data structure (a HIRB tree) that can be stored within a vORAM. Together, this construction provides an efficient and practical oblivious data structure (ODS) for a key/value map, and goes further to provide an additional privacy guarantee as compared to prior ODS maps: even upon client compromise, deleted data and the history of old operations remain hidden to the attacker. We implement and measure the performance of our system using Amazon Web Services, and the single-operation time for a realistic database (up to 2^{18} entries) is less than 1 second. This represents a 100x speed-up compared to the current best oblivious map data structure (which provides neither secure deletion nor history independence) by Wang et al. (CCS 14).

1 Introduction

1.1 Motivation

Increasingly, organizations and individuals are storing large amounts of data in remote, shared cloud servers. For sensitive data, it is important to protect the privacy not only of the data itself but also of the access to the metadata that may contain *which* records have been accessed and *when*, thereby revealing properties of the underlying data, even if that data is encrypted. There are multiple points of potential information leakage in this setting: an adversary could observe network communication between the client and server; an adversary could compromise the cloud itself, observing the data stored at the server, possibly including mirrored copies or backups; an adversary could observe the computations performed by the remote server; the adversary may compromise the locally-stored client data; or, finally, the adversary may compromise the data in multiple ways, e.g., a complete compromise of both the remotely stored cloud storage and locally-stored client storage[1].

While a complete compromise will inevitably reveal private data, we seek data storage mechanisms which maximize privacy while maintaining reasonable, practical efficiency, at any level of compromise. For generality, we assume a computationally-limited server which may only store and retrieve blocks of raw data, and we focus on the most basic (and perhaps most important) data structure: a key/value map.

Oblivious RAM (ORAM). With a computationally-limited server, the *access pattern* of client-server communication reveals the entire history of the remote data store. This access pattern, even if the actual data is

[1] We assume an honest-but-curious server throughout, and leave achieving an ODS with malicious servers as an open problem.

encrypted, may leak sensitive information about the underlying stored data, such as keyword search queries or encryption keys [12, 20, 42].

A *generic* solution to protect against access pattern leakage is oblivious RAM (ORAM) [13], which obscures the operation being performed (read/write), the address on which it operates, and the contents of the underlying data. Any program (with the possible necessity of some padding) can be executed using an ORAM to hide the access patterns to the underlying data.

A great number of ORAM schemes have been recently proposed, most aiming to improve the efficiency as it relates to the recursive index structure, which is typically required to store the hidden locations of items within the ORAM (for example [11, 16, 21, 24, 36, 37] and references therein). However, an important aspect overlooked by previous work is the *size* of data items themselves. The vORAM construction we propose provides an affirmative answer to the following question:

> *Can an oblivious RAM hide the size of varying-sized items, with greater efficiency than that achieved by trivial padding?*

Oblivious data structure (ODS). Recently, Wang et al. [40] showed that it is possible to provide obliviousness more efficiently if the specifics of the target program are considered. In particular, among other results, Wang et al. achieved an oblivious data structure (ODS) scheme for a key-value map, by constructing an AVL tree *on a non-recursive ORAM without using the position map*. Their scheme requires $\tilde{O}(\log n)$ ORAM blocks of client storage, where n is the maximum number of allowable data items. More importantly, due to lack of position map lookups, the scheme requires only $O(\log^2 n)$ blocks of communication bandwidth, which constituted roughly an $O(\log n)$-multiplicative improvement in communication bandwidth over the generic ORAM solution. We will briefly explain "the pointer-based technique" they introduced to eliminate the position map in Section 1.3.

The practicality of oblivious data structures are challenging, however, owing to the combination of inefficiencies in the data structures compounded with that of the underlying ORAM. In our experimental results presented in Section 6, and Table 1 specifically, we found that the AVL ODS suffers greatly from a high round complexity, and also that the per-operation bandwidth exceeds the total database size (and hence a trivial alternative implementation) until the number of entries exceeds 1 million.

Similar observations for ORAMs more generally were made recently by Bindschaedler et al. [4], who examined existing ORAM alternatives in a realistic cloud setting, and found many theoretical results lacking in practice. We ask a related question for ODS, and answer it in the affirmative with our HIRB data structure stored in vORAM:

> *Can an oblivious map data structure be made practically useful in the cloud setting?*

Catastrophic attack. In the cloud storage scenario, obliviousness will protect the client's privacy from any observer of network traffic or from the cloud server itself. However, if the attacker compromises the client and obtains critical information such as the encryption keys used in the ODS, all the sensitive information stored in the cloud will simply be revealed to the attacker.

We call this scenario a *catastrophic attack*, and it is important to stress that this attack is quite realistic. The client machine may be stolen or hacked, or it may even be legally seized due to a subpoena.

Considering the increasing incidence of high-profile catastrophic attacks in practice (e.g., [1, 19]), and that even government agencies such the CIA are turning to third-party cloud storage providers [23], it is important to provide some level of privacy in this attack scenario. Given this reality, we ask and answer the following additional question:

> *Can we provide any privacy guarantee even under a catastrophic attack?*

Specifically, our vORAM+HIRB construction will provide strong security for deleted data, as well as a weaker (yet optimal) security for the history of past operations, after complete client compromise.

1.2 Security Requirements

Motivated by the goals outlined previously, we aim to construct a cloud database system that provides the following two security properties:

- Obliviousness: The system should hide both the data and the access patterns from an observer of all client-server communication (i.e., be an ODS).
- Secure Deletion and History Independence: The system, in the face of a catastrophic attack, should ensure that no previously deleted data, the fact that previous data existed, or the order in which extant data has been accessed, is revealed to an attacker.

Additionally, we require that the system be practically useful, meaning it should be *more efficient (w.r.t. communication cost, access time, and round complexity) than previous ODS schemes*, even those that do not necessarily provide secure deletion nor history independence.

Each required security notion has individually been the focus of numerous recent research efforts (see Section 2). To the best of our knowledge, however, *there is no previous work that considers all the properties simultaneously.* We aim at combining the security properties from *obliviousness*, *secure deletion*, and *history independence* into a new, unified system for secure remote cloud storage. The previous ODS schemes do not provide history-independence nor secure deletion and are inefficient for even small data stores. Previous mechanisms providing secure deletion or history independence are more efficient, but do not hide the access pattern in remote cloud storage (i.e., do not provide obliviousness). And unfortunately, the specific requirements of these constructions means they cannot trivially be combined in a straightforward way.

To better understand the necessity of each of the security requirements, consider each in kind.

Obliviousness: The network traffic to a remote server reveals to an attacker, or to the server itself, which raw blocks are being read and written. Even if the block contents are encrypted, an attacker may be able to infer sensitive information from this access pattern itself. Like previous ODS schemes, our system will ensure this is not the case; the server-level access pattern reveals nothing about the underlying data operations that the user is performing.

History independence: By inspecting the internal structure of the currently existing data in the cloud after a catastrophic attack, the attacker may still be able to infer information about which items were recently accessed or the likely prior existence of a record even if that record was previously deleted [2]. However, if an ODS scheme provided *perfect history independence*, the catastrophic attacker cannot infer which sequence of operations was applied, among all the sequences that could have resulted in the current set of the data items. Interestingly, we show that it is impossible to achieve perfect history independence in our setting with a computationally-limited server; nonetheless, providing ℓ-history independence is still desirable, where only the most recent ℓ operations are revealed but nothing else.

Secure deletion: Given that only bounded history independence is possible, the privacy of deleted data must be considered. It is desirable that the catastrophic attacker should not be able to guess information about deleted data. In practice, data deleted from persistent media, such as hard disk drives, is easily recoverable through standard forensic tools. In the cloud setting, the problem is compounded because there is normally no direct control of how and where data is stored on physical disks, or backed up and duplicated in servers around the globe. We follow a similar approach as [34], where secure deletion is accomplished by re-encrypting and deleting the old encryption key from local, erasable memory such as RAM.

1.3 Our Work

Pointer-based technique. Wang et al. [40] designed an ODS scheme for map by storing an AVL tree on top of the non-recursive Path ORAM [37] using the pointer-based technique, in which the ORAM position tags act as pointers, and the pointer to each node in the AVL tree is stored in its parent node. With this technique, when the parent node is fetched, the position tags of its children are immediately obtained. Therefore, *the position map lookups are no more necessary*.

Similarly, in our ODS scheme, we will overlay a data structure on a non-recursive ORAM using a pointer-based technique for building the data structure.

We stress that the non-recursive Path ORAM still remains the best choice when we would like to embed our data structure in an ORAM with the pointer-based technique, in spite of all the recent improvements on ORAM techniques. This is mainly because all ORAM improvement techniques consider the setting where an ORAM runs in a stand-alone fashion, unlike our setting where the ORAM actions, in particular with position map lookups, depend on the upper-layer data structure. In particular, with the non-recursive Path ORAM, each ORAM operation takes only a *single round* of communication between the client and server, since there is no position map lookup; moreover, each operation transfers $O(\log n)$ blocks where the size of each block can be arbitrarily small up to $\Omega(\log n)$. To compare the non-recursive Path ORAM with the most recent stand-alone ORAMs, each operation of the constant communication ORAM [29] transfers $O(1)$ blocks each of which should be of size $\Omega(\log^4 n)$, and it additionally uses computation-intensive homomorphic encryptions. For Ring ORAM [35], it still refers to the position map, and although its online stage may be comparable to the non-recursive Path ORAM, it still has the additional offline stage. The non-recursive version of these ORAMs has essentially the same efficiency as the non-recursive Path ORAM.

Impracticality of existing data structures. Unfortunately, no current data structure exists that can meet our security and efficiency requirements:

- It should be a *rooted tree.* This is necessary, since we would like to use the pointer-based technique. Because the positions are randomly re-selected on any access to that node, the tree structure is important in order to avoid dangling references to old pointers.
- The height of the tree should be $O(\log n)$ *in the worst case*. To achieve obliviousness, all operations must execute with the same running time, which implies all operations will be padded to some upper bound that is dependent on the height of the tree.
- The data structure itself should be *(strongly) history-independent*, meaning the organization of nodes depends only on the current contents, and not the order of operations which led to the current state. As a negative example, consider an AVL tree, which is not history independent. Inserting the records A, B, C, D in that order; or B, C, D, A in that order; or A, B, C, D, E and then deleting E; will each result in a different state of the data structure, thereby revealing (under a catastrophic attack) information on the insertion order and previous deletions.

To the best of our knowledge, there is no data structure satisfying all of the above conditions. Most tree-based solutions, including AVL trees and B-trees, are not history independent. Treaps and B-treaps are rooted trees with history independence, but they have linear height in the worst case. Skip-lists and B-Skip-lists are history independent and tree-like, but technically they are not rooted trees and thereby not amenable to the pointer-based technique. That is, Skip-lists and B-Skip-lists have multiple incoming links, requiring linear updates in the ORAM to maintain the pointers and position tags in the worst case.

HIRB. We developed a new data structure, called a HIRB tree (history independent, randomized B-tree), that satisfies all the aforementioned requirements. Conceptually, it is a *fixed height* B-tree such that when each item is inserted, the level in HIRB tree is determined by $\log_\beta n$ trials of (pseudorandom) biased coin flipping where β is the block factor. The tree may split or merge depending on the situation, but *it never rotates*. The fixed height of the tree, i.e. $H = 1 + \log_\beta n$, is very beneficial for efficiency. In particular, *every operation* visits at most $2H$ nodes, which greatly saves on padding costs, compared to the ODS scheme of [40] where each AVL tree operation must be padded up to visiting $3 \cdot 1.44 \cdot \lg n$ nodes.

The HIRB is described more carefully in Section 5, with full details in the appendix.

vORAM. One challenge with HIRB trees is that number of items that each tree node contains are variable, and in the unlucky case, it may become too large for an ORAM bucket to store.

This challenge is overcome by introducing vORAM (ORAM with variable-size blocks). The design of vORAM is based on the non-recursive version of Path ORAM where the bucket size remains fixed, but each bucket may contain as many variable-size blocks (or parts of blocks) as the bucket space allows. Blocks may also be stored across multiple buckets (in the same path).

We observe that the irregularity of the HIRB node sizes can be smoothed over $O(\log n)$ buckets from the vORAM root to an vORAM leaf, and we prove that the stash size on the client can still be small $\tilde{O}(\log n)$ with high probability. We note that vORAM is *the first ORAM that deals with variable size blocks*, and may be of independent interest.

The vORAM is described carefully in Section 4, and the full details are provided in the appendix.

Secure deletion. Finally, for secure deletion, a parent vORAM bucket contains the encryption keys of both children. When a bucket is modified, it is encrypted with a fresh key; then the encryption keys in the parent is accordingly modified, which recursively affects all its ancestors. However, we stress that in each vORAM operation, *leaf-to-root refreshing takes place anyway, and adding this mechanism is bandwidth-free.* Additionally, instead of using the label of each item directly in HIRB, we use the hash of the label. This way, we can remove the dependency between the item location in HIRB and its label (with security proven in the random oracle model).

Imperfect history independence. Our approach does not provide perfect history independence. Although the data structure in the vORAM is history independent, the vORAM is not. Indeed, in any tree-based or hierarchical ORAM, the items near the root have been more likely recently accessed as compared to items near the leaves. The catastrophic adversary can observe all the ORAM structure, and such leakage breaks perfect history independence. We show a formal lower bound for the amount of leakage in Section 3.

Experiments and efficiency of our scheme. In order to empirically measure the performance of our construction, we first performed an analysis to determine the smallest constant factor overhead to achieve high performance with negligible likelihood of failure. Following this, we implemented our system in the cloud with Amazon Web Services as the cloud provider and compared it to alternatives that provide some, but not all of the desired security properties. To the best of our knowledge, *there has been no previous work that implements and tests any ODS system in the actual cloud setting*. As argued in Bindschaedler et al. [4], who independently compared various ORAM systems in the cloud, it is important to see how systems work in the actual intended setting. As comparison points, we compare our system with the following implementations:

- ORAM+AVL: We reimplemented the ODS map by Wang et al. [40] that provides obliviousness but not secure deletion nor history independence.
- SD-B-Tree: We implemented a remotely stored block-level, encrypted B-Tree (as recommend by the secure deletion community [34]) that provides secure deletion but not history independence nor obliviousness.
- Naive approach: We implemented a naive approach that achieves all the security properties by transferring and re-encrypting the entire database on each access.

In all cases the remotely stored B-Tree is the fastest as it requires the least amount of communication cost (no obliviousness). For similar reasons, vORAM+HIRB is much faster than the baseline as the number of items grows (starting from 2^{14} items), since the baseline requires communication that is linear in the number of items. We also describe a number of optimizations (such as concurrent connections and caching) that enables vORAM+HIRB to be competitive with the baseline even when storing as few as 2^9 items. It should be noted, without optimizations, the access time is on the order of a few seconds, and with optimizations, access times are less than one second.

Surprisingly, however, the vORAM+HIRB is *20x faster than ORAM+AVL*, irrespective of the number of items, even though ORAM+AVL does not support history independence or secure deletion. We believe this is mainly because vORAM+HIRB requires much smaller round complexity. Two factors drive the round complexity improvement:

Much smaller height: While each AVL tree node contains only one item, each HIRB node contains β items on average, and is able to take advantage of slightly larger buckets which optimize the bandwidth to remote cloud storage by storing the same amount of data in trees with smaller height.

Much less padding: AVL tree operations sometimes get complicated with balancing and rotations, due to which each operation should be padded up to $3 \cdot 1.44 \lg n$ node accesses. However, HIRB operations are simple, do not require rotations, and thus, each operation accesses at most $2 \log_\beta n$ nodes.

Although the Path-ORAM bucket for ORAM+AVL is four times smaller than the vORAM bucket in our implementation, it affects bandwidth but not the round complexity. The fully optimized vORAM+HIRB protocol is about *100x faster than ORAM+AVL*. We describe the details of our experiments in Section 6.

Summary of our contributions. To summarize, the contributions of this paper are:

- New security definitions of history independence and secure deletion under a catastrophic attack.
- The design and analysis of an oblivious RAM with variable size blocks, the vORAM;
- The design and analysis of a new history independent and randomized data structure, the HIRB tree;
- A lower bound on history independence for any ORAM construction with sub-linear bandwidth;
- Improvements to the performance of mapped data structures stored in ORAMs;
- An empirical measurement of the settings and performance of the vORAM in the actual cloud setting;
- The implementation and measurement of the vORAM+HIRB system in the actual cloud setting.

2 Related Work

We discuss related work in oblivious data structures, history independence, and secure deletion. Our system builds upon these prior results and combines the security properties into a unified system.

ORAM and oblivious data structures. ORAM protects the access pattern from an observer such that it is impossible to determine which operation is occurring, and on which item. The seminal work on the topic is by Goldreich and Ostrovsky [13], and since then, many works have focused on improving efficiency of ORAM in both the space, time, and communication cost complexities (for example [11, 16, 21, 24, 36, 37] just to name a few; see the references therein).

There have been works addressing individual oblivious data structures to accomplish specific tasks, such as priority queues [39], stacks and queues [27], and graph algorithms [5]. Recently, Wang et al. [40] achieved oblivious data structures (ODS) for maps, priority queues, stacks, and queues much more efficiently than previous works or naive implementation of the data structures on top of ORAM.

Our vORAM construction builds upon the non-recursive Path ORAM [40] and allows *variable sized* data items to be spread across multiple ORAM buckets. Although our original motivation was to store differing-sized B-tree nodes from the HIRB, there may be wider applicability to any context where the size (as well as contents and access patterns) to data needs to be hidden.

Interestingly, based on our experimental results, we believe the ability of vORAM to store partial blocks in each bucket may even improve the performance of ORAM when storing uniformly-sized items. However, we will not consider this further in the current investigation.

History independence. History independence of data structures requires that the current organization of the data within the structure reveals nothing about the prior operations thereon. Micciancio [26] first considered history independence in the context of 2-3 trees, and the notions of history independence were formally developed in [8, 17, 30]. The notion of *strong history independence* [30] holds if for any two sequences of operations, the distributions of the memory representations are identical at all time-points that yield the same storage content. Moreover, a data structure is strongly history independent if and only if it has a *unique representation* [17]. There have been uniquely-represented constructions for hash functions [6, 31] and variants of a B-tree (a B-treap [14], and a B-skip-list [15]). We adopt the notion of unique representation for history independence when developing our history independent, randomized B-tree, or HIRB tree.

We note that history independence of these data structures considers a setting where a single party runs some algorithms on a single storage medium, which doesn't correctly capture the actual cloud setting where client and server have separate storage, execute protocols, and exchange messages to maintain the data structures. Therefore, we extend the existing history independence and give a new, augmented notion of history independence for the cloud setting with a catastrophic attack.

Independently, the recent work of [3] also considers a limited notion of history independence, called Δ-history independence, parameterized with a function Δ that describes the leakage. Our definition of history independence has a similar notion, where the leakage function Δ captures the number of recent operations which may be revealed in a catastrophic attack.

Secure deletion. Secure deletion means that data deleted cannot be recovered, even by the original owner. It has been studied in many contexts [33], but here we focus on the cloud setting, where the user has little or no control over the physical media or redundant duplication or backup copies of data. In particular, we build upon secure deletion techniques from the applied cryptography community. The approach is to encrypt all data stored in the cloud with encryption keys stored locally in erasable memory, so that deleting the keys will securely delete the remote data by rendering it non-decryptable.

Boneh and Lipton [7] were the first to use encryption to securely remove files in a system with backup tapes. The challenge since was to more effectively manage encrypted content and the processes of re-encryption and erasing decryption keys. For example, Di Crescenzo et al. [10] showed a more efficient method for secure deletion using a tree structure applied in the setting of a large non-erasable persistent medium and a small erasable medium. Several works considered secure deletion mechanisms for a versioning file system [32], an inverted index in a write-once-read-many compliance storage [28], and a B-tree (and generally a mangrove) [34].

3 Preliminaries

We assume that readers are familiar with security notions of standard cryptographic primitives [22]. Let λ denote the security parameter.

Modeling data structures. Following the approach from the secure deletion literature, we use two storage types: *erasable memory* and *persistent storage*. Contents deleted from erasable memory are non-recoverable, while the contents in persistent storage cannot be fully erased. We assume the size of erasable memory is small while the persistent storage has a much larger capacity. This mimics the cloud computing

$\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}_1,\mathcal{A}_2}(\mathcal{D},\lambda,n,h,b)$
$\mathsf{acc}_0 \leftarrow \mathcal{D}.\mathsf{Init}(1^\lambda,n)$;
$(\overrightarrow{\mathsf{op}}^{(0)},\overrightarrow{\mathsf{op}}^{(1)},\textsc{st}) \leftarrow \mathcal{A}_1(1^\lambda,\mathsf{acc}_0)$;
$\overrightarrow{\mathsf{acc}} \leftarrow \mathcal{D}.\overrightarrow{\mathsf{op}}^{(b)}()$;
if $h=1$:
  return $\mathcal{A}_2(\textsc{st},\overrightarrow{\mathsf{acc}},\mathcal{D}.\mathsf{em})$;
else
  return $\mathcal{A}_2(\textsc{st},\overrightarrow{\mathsf{acc}})$;

$\mathsf{EXP}^{\mathsf{sdel}}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3}(\mathcal{D},\lambda,n,b)$
$\mathsf{acc}_0 \leftarrow \mathcal{D}.\mathsf{Init}(1^\lambda,n)$;
$d_0 \leftarrow \mathcal{A}_1(1^\lambda,0)$;
$d_1 \leftarrow \mathcal{A}_1(1^\lambda,1)$;
$(\overrightarrow{\mathsf{op}}_{d_0,d_1},S) \leftarrow \mathcal{A}_2(\mathsf{acc}_0,d_0,d_1)$;
$\overrightarrow{\mathsf{acc}} \leftarrow \mathcal{D}.(\overrightarrow{\mathsf{op}}_{d_0,d_1} \bowtie_S d_b)()$;
return $\mathcal{A}_3(\mathsf{acc}_0,\overrightarrow{\mathsf{acc}},\mathcal{D}.\mathsf{em})$;

Figure 1: Experiments for security definitions

setting where cloud storage is large and persistent due to lack of user control, and local storage is more expensive but also controlled directly.

We define a data structure $\mathcal{D}$ as a collection of data that supports initialization, insertion, deletion, and lookup, using both the erasable memory and the persistent storage. Each operation may be parameterized by some operands (e.g., lookup by a label). For a data structure $\mathcal{D}$ stored in this model, let $\mathcal{D}.\mathsf{em}$ and $\mathcal{D}.\mathsf{ps}$ denote the contents of the erasable memory and persistent storage, respectively. For example, an encrypted graph structure may be stored in $\mathcal{D}.\mathsf{ps}$ while the decryption key resides in $\mathcal{D}.\mathsf{em}$. For an operation op on $\mathcal{D}$, let $\mathsf{acc} \leftarrow \mathcal{D}.\mathsf{op}()$ denote executing the operation op on the data structure $\mathcal{D}$ where acc is the access pattern over the persistent storage during the operation. The access pattern to erasable memory is assumed to be hidden. For a sequence of operations $\overrightarrow{\mathsf{op}} = (\mathsf{op}_1,\ldots,\mathsf{op}_m)$, let $\overrightarrow{\mathsf{acc}} \leftarrow \mathcal{D}.\overrightarrow{\mathsf{op}}()$ denote applying the operations on $\mathcal{D}$, that is, $\mathsf{acc}_1 \leftarrow \mathcal{D}.\mathsf{op}_1()$, $\ldots$, $\mathsf{acc}_m \leftarrow \mathcal{D}.\mathsf{op}_m()$, with $\overrightarrow{\mathsf{acc}} = (\mathsf{acc}_1,\ldots,\mathsf{acc}_m)$. We note that the access pattern $\overrightarrow{\mathsf{acc}}$ completely determines the state of persistent storage $\mathcal{D}.\mathsf{ps}$.

Obliviousness and history independence. Obliviousness requires that the adversary without access to erasable memory cannot obtain any information about actual operations performed on data structure $\mathcal{D}$ other than the number of operations. This security notion is defined through an experiment obl-hi, given in Figure 1, where $\mathcal{D}$, λ, n, h, b denote a data structure, the security parameter, the maximum number of items $\mathcal{D}$ can contain, history independence, and the challenge choice.

In the experiment, the adversary chooses two sequences of operations on the data structure and tries to guess which sequence was chosen by the experiment with the help of access patterns. The data structure provides obliviousness if every polynomial-time adversary has only a negligible advantage.

Definition 1. *For a data structure $\mathcal{D}$, consider the experiment $\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D},\lambda,n,0,b)$ with adversary $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2)$. We call the adversary $\mathcal{A}$* admissible *if $\mathcal{A}_1$ always outputs two sequences with the same number of operations storing at most n items. We define the advantage of the adversary $\mathcal{A}$ in this experiment as:*

$$\mathbf{Adv}^{\mathsf{obl}}_{\mathcal{A}}(\mathcal{D},\lambda,n) = \left| \begin{array}{l} \Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D},\lambda,n,0,0)=1] \\ \quad -\Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D},\lambda,n,0,1)=1] \end{array} \right|.$$

We say that $\mathcal{D}$ provides obliviousness *if for any sufficiently large λ, any $n \in poly(\lambda)$, and any PPT admissible adversary $\mathcal{A}$, we have $\mathbf{Adv}^{\mathsf{obl}}_{\mathcal{A}}(\mathcal{D},\lambda,n) \le \mathsf{negl}(\lambda)$.*

Now we define history independence. As we will see, perfect history independence is inherently at odds with obliviousness and sub-linear communication cost. Therefore, we define *parameterized* history independence instead that allows for a relaxation of the security requirement. The parameter determines the allowable leakage of recent history of operations. One can interpret a history-independent data structure with leakage of ℓ operations as follows: Although the data structure may reveal some recent ℓ operations applied to itself, it does not reveal any information about older operations, except that the total sequence resulted in the current state of data storage.

The experiment in this case is equivalent to that for obliviousness, except that (1) the two sequences must result in the same state of the data structure at the end, (2) the last ℓ operations in both sequences must be identical, and (3) the adversary gets to view the local, erasable memory as well as the access pattern to persistent storage.

Definition 2. *For a data structure $\mathcal{D}$, consider the experiment $\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, b)$ with adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. We call the adversary $\mathcal{A}$ ℓ-*admissible *if $\mathcal{A}_1$ always outputs sequences $\overrightarrow{\mathsf{op}}^{(0)}$ and $\overrightarrow{\mathsf{op}}^{(1)}$ which have the same number of operations and result in the same set storing at most n data items, and the last ℓ operations of both are identical. We define the advantage of an adversary $\mathcal{A}$ in this experiment above as:*

$$\mathbf{Adv}^{\mathsf{hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n) = \left| \begin{array}{l} \Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 0) = 1] \\ \quad - \Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 1) = 1] \end{array} \right|.$$

We say that the data structure $\mathcal{D}$ provides history independence *with leakage of ℓ operations if for any sufficiently large λ, any $n \in poly(\lambda)$, and any PPT ℓ-admissible adversary $\mathcal{A}$, we have $\mathbf{Adv}^{\mathsf{hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n) \leq \mathsf{negl}(\lambda)$.*

Lower bound on history independence. Unfortunately, the history independence property is inherently at odds with the nature of oblivious RAM. The following lower bound demonstrates that there is a linear tradeoff between the amount of history independence and the communication bandwidth of any ORAM mechanism.

Theorem 1. *Any oblivious RAM storage system with a bandwidth of k bytes per access achieves at best history independence with leakage of $\Omega(n/k)$ operations in storing n blocks.*

The intuition behind the proof[2] is that, in a catastrophic attack, an adversary can observe which persistent storage locations were recently accessed, and furthermore can decrypt the contents of those locations because they have the keys from erasable memory. This will inevitably reveal information to the attacker about the order and contents of recent accesses, up to the point at which all n elements have been touched by the ORAM and no further past information is recoverable.

Admittedly this lower bound limits what may be achievable in terms of history independence. But still, leaking only a known maximum number of prior operations is better than (potentially) leaking all of them!

Consider, by contrast, an AVL tree implemented within a standard ORAM as in prior work. Using the fact that AVL tree shapes reveal information about past operations, the adversary can come up with two sequences of operations such that (i) the first operations of each sequence result in a distinct AVL tree shape but the same data items, and (ii) the same read operations, as many as necessary, follow at the end. With the catastrophic attack, the adversary will simply observe the tree shape and make a correct guess. This argument holds for any data structure whose shape reveals information about past operations, which therefore have *no upper bound* on the amount of history leakage.

Secure deletion. Perfect history independence implies secure deletion. However, the above lower bound shows that complete history independence will not be possible in our setting. So, we consider a complementary security notion that requires *strong security for the deleted data*. Secure deletion is defined through an experiment sdel, given in Figure 1. In the experiment, $\mathcal{A}_1$ chooses two data items d_0 and d_1 at random, based on which $\mathcal{A}_2$ outputs $(\overrightarrow{\mathsf{op}}_{d_0,d_1}, S)$. Here, $\overrightarrow{\mathsf{op}}_{d_0,d_1}$ denotes a vector of operations containing neither d_0 nor d_1, and $S = (s_1, s_2, \ldots, s_m)$ is a monotonically increasing sequence. $\overrightarrow{\mathsf{op}}_{d_0,d_1} \bowtie_S d_b$ denotes injecting d_b into $\overrightarrow{\mathsf{op}}_{d_0,d_1}$ according to S. In particular, "insert d_b" is placed at position s_1; for example, if s_1 is 5, this insert operation is placed right before the 6th operation of $\overrightarrow{\mathsf{op}}_{d_0,d_1}$. Then, "look-up d_b" is placed at positions $s_2, \ldots, s_{m-1}$, and finally "delete d_b" at s_m.

[2]Full proofs for the main theorems may be found in Appendix C.

Definition 3. *For a data structure $\mathcal{D}$, consider the experiment $\mathsf{EXP}^{\mathsf{sdel}}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3}(\mathcal{D},\lambda,n,b)$ with adversary $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3)$. We call the adversary $\mathcal{A}$* admissible *if for any data item d that $\mathcal{A}_1(1^\lambda,0)$ (resp., $\mathcal{A}_1(1^\lambda,1)$) outputs, the probability that $\mathcal{A}_1$ outputs d is negligible in λ, i.e., the output $\mathcal{A}_1$ forms a high-entropy distribution; moreover, the sequence of operations from $\mathcal{A}_2$ must store at most n items. We define the advantage of $\mathcal{A}$ as:*

$$\mathbf{Adv}^{\mathsf{sdel}}_{\mathcal{A}}(\mathcal{D},\lambda,n) = \left| \begin{array}{l} \Pr[\mathsf{EXP}^{\mathsf{sdel}}_{\mathcal{A}}(\mathcal{D},\lambda,n,0)=1] \\ \; - \Pr[\mathsf{EXP}^{\mathsf{sdel}}_{\mathcal{A}}(\mathcal{D},\lambda,n,1)=1] \end{array} \right| .$$

We say that the data structure $\mathcal{D}$ provides secure deletion *if for any sufficiently large λ, any $n \in poly(\lambda)$, and any PPT admissible adversary $\mathcal{A}$, we have $\mathbf{Adv}^{\mathsf{sdel}}_{\mathcal{A}}(\mathcal{D},\lambda,n) \le \mathsf{negl}(\lambda)$.*

Note that our definition is stronger than just requiring that the adversary cannot recover the deleted item; for any two high entropy distributions chosen by the adversary, the adversary cannot tell from which distribution the deleted item was drawn.

4 ORAM with variable-size blocks (vORAM)

The design of vORAM is based on the non-recursive version of Path ORAM [37], but we are able to add more flexibility by allowing each ORAM bucket to contain as many variable-size blocks (or parts of blocks) as the bucket space allows. We will show that vORAM preserves obliviousness and maintains a small stash as long as the size of variable blocks can be bounded by a geometric probability distribution, which is the case for the HIRB that we intend to store within the vORAM. To support secure deletion, we also store encryption keys within each bucket for its two children, and these keys are re-generated on every access, similarly to other work on secure deletion [10, 34].

Parameters. The vORAM construction is governed by the following parameters:

- The height T of the vORAM tree: The vORAM is represented as a complete binary tree of buckets with height T (the levels of the tree are numbered 0 to T), so the total number of buckets is $2^{T+1} - 1$. T also controls the total number of allowable data blocks, which is 2^T.
- The bucket size Z: Each bucket has Z bits, and this Z must be at least some constant times the *expected block size B* for what will be stored in the vORAM.
- The stash size parameter R: Blocks (or partial blocks) that overflow from the root bucket are stored temporarily in an additional memory bank in local storage called the stash, which can contain up to $R \cdot B$ bits.
- Block collision parameter γ: Each block will be assigned a random identifier id; these identifiers will all be distinct at every step with probability $1 - \mathsf{negl}(\gamma)$.

Bucket structure. Each bucket is split into two areas: header and data. See Figure 2 for a pictorial description. The header area contains two encryption keys for the two child buckets. The data area contains a sequence of (possibly partial) blocks, each preceded by a unique identifier string and the block data length. The end of the data area is filled with 0 bits, if necessary, to pad up to the bucket size Z.

Each id_i uniquely identifies a block and also encodes the path of buckets along which the block should reside. Partial blocks share the same identifier with each length l indicating how many bytes of the block are stored in that bucket. Recovering the full block is accomplished by scanning from the stash along the path associated with id (see Figure 3). We further require the first bit of each identifier to be always 1 in

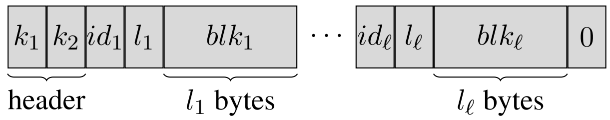

Figure 2: A single vORAM bucket with ℓ partial blocks.

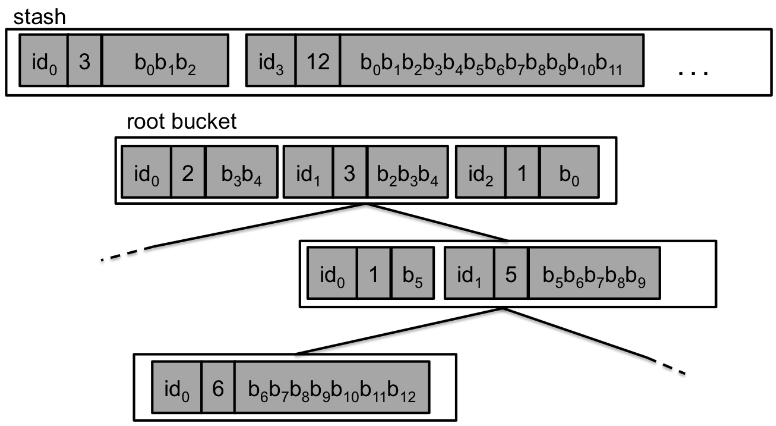

Figure 3: A sample vORAM state with partial blocks with id_0, id_1, id_2, id_3: Note that the partial blocks for id_0 are opportunistically filled up the vORAM from leaf to root and then remaining partial blocks are placed in the stash.

order to differentiate between zero padding and the start of next identifier. Moreover, to avoid collisions in identifiers, the length of each identifier is extended to $2T + \gamma + 1$ bits, where γ is the collision parameter mentioned above. The most significant $T + 1$ bits of the identifier (including the fixed leading 1-bit) are used to match a block to a leaf, or equivalently, a path from root to leaf in the vORAM tree.

vORAM operations. Our vORAM construction supports the following operations.

- $\mathsf{insert}(blk) \mapsto id$. Inserts the given block blk of data into the ORAM and returns a new, randomly-generated id to be used only once at a later time to retrieve the original contents.
- $\mathsf{remove}(id) \mapsto blk$. Removes the block corresponding to id and returns the original data blk as a sequence of bytes.
- $\mathsf{update}(id, callback) \mapsto id^+$. Given id and a user-defined function $callback$, perform $\mathsf{insert}(callback(\mathsf{remove}(id)))$ in a single step.

Each vORAM operation involves two phases:

1. $\mathsf{evict}(id)$. Decrypt and read the buckets along the path from the root to the leaf encoded in the identifier id, and remove all the partial blocks along the path, merging partial blocks that share an identifier, and storing them in the stash.
2. $\mathsf{writeback}(id)$. Encrypt all blocks along the path encoded by id with new encryption keys and opportunistically store any partial blocks from stash, dividing blocks as necessary, filling from the leaf to the root.

An insert operation first evicts a randomly-chosen path, then inserts the new data item into the stash with a second randomly-chosen identifier, and finally writes back the originally-evicted path. A remove operation evicts the path specified by the identifier, then removes that item from the stash (which must have had all its partial blocks recombined along the evicted path), and finally writes back the evicted path without the deleted item. The update operation evicts the path from the initial id, retrieves the block from stash, passes it to the $callback$ function, re-inserts the result to the stash with a new random id^+, and finally calls writeback on the original id. A full pseudocode description of all these operations is provided in Appendix A.

Security properties. For obliviousness, any insert, remove, update operation is computationally indistinguishable based on its access pattern because the identifier of each block is used only once to retrieve that item and then immediately discarded. Each remove or update trivially discards the identifier after reading the path, and each insert evicts buckets along a bogus, randomly chosen path before returning a fresh id^+ to be used as the new identifier for that block.

Theorem 2. *The vORAM provides obliviousness.*

Secure deletion is achieved via key management of buckets. Every evict and writeback will result in a path's worth of buckets to be re-encrypted and re-keyed, including the root bucket. Buckets containing any removed data may persist, but the decryption keys are erased since the root bucket is re-encrypted, rendering the data unrecoverable. Similarly, recovering any previously deleted data reduces to acquiring the old-root key, which was securely deleted from local, erasable memory.

However, each evict and writeback will disclose the vORAM path being accessed, which must be handled carefully to ensure no leakage occurs. Fortunately, identifiers (and therefore vORAM paths as well) are uniformly random, independent of the deleted data and revealing no information about them.

Theorem 3. *The vORAM provides secure deletion.*

Regarding history independence, although any removed items are unrecoverable, the height of each item in the vORAM tree, as well as the history of accesses to each vORAM tree bucket, may reveal some information about the *order*, or timing, of when each item was inserted. Intuitively, items appearing closer to the root level of the vORAM are more likely to have been inserted recently, and vice versa. However, if an item is inserted and then later has its path entirely evicted due to some other item's insertion or removal, then any history information of the older item is essentially wiped out; it is as if that item had been removed and re-inserted. Because the identifiers used in each operation are chosen at random, after some $O(n \log n)$ operations it is likely that *every* path in the vORAM has been evicted at least once.

Theorem 4. *The vORAM provides history independence with leakage of $O(n \log n + \lambda n)$ operations.*

In fact, we can achieve asymptotically optimal leakage with only a constant-factor blowup in the bandwidth. Every vORAM operation involves reading and writing a single path. Additionally, after each operation, we can evict and then re-write a complete *subtree* of size $\lg n$ which contains $(lgn)/2 - 1$ leaf buckets in a deterministicly chosen dummy operation that simply reads the buckets into stash, then rewrites the buckets with no change in contents but allowing the blocks evicted from the dummy operation and those evicted from the access to all move between levels of the vORAM as usual. The number of nodes evicted will be less than $2 \lg n$, to encompass the subtree itself as well as the path of buckets to the root of the subtree, and hence the total bandwidth for the operation remains $O(\log n)$.

The benefit of this approach is that if these dummy subtree evictions are performed *sequentially* across the vORAM tree on each operation, any sequence of $n/\lg n$ operations is guaranteed to have evicted every bucket in the vORAM at least once. Hence this would achieve history independence with only $O(n/\log n)$ leakage, which matches the lower bound of Theorem 1 and is therefore optimal up to constant factors.

Stash size. Our vORAM construction maintains a small stash as long as the size of variable blocks can be bounded by a geometric probability distribution, which is the case for the HIRB that we intend to store within the vORAM.

Theorem 5. *Consider a vORAM with T levels, collision parameter γ, storing at most $n = 2^T$ blocks, where the length l of each block is chosen independently from a distribution such that $\mathbb{E}[l] = B$ and $\Pr[l > mB] < 0.5^m$. Then, if the bucket size Z satisfies $Z \geq 20B$, for any $R \geq 1$, and after any single access to the vORAM, we have*

$$\Pr[|stash| > RB] < 28 \cdot (0.883)^R.$$

Note that the constants 28 and 0.883 are technical artifacts of the analysis, and do not matter except to say that $0.883 < 1$ and thus the failure probability decreases exponentially with the size of stash.

As a corollary, for a vORAM storing at most n blocks, the cloud storage requirement is $40Bn$ bits, and the bandwidth for each operation amounts to $40B \lg n$ bits. However, this is a theoretical upper bound, and our experiments in Section 6 show a smaller constants suffice. namely, setting $Z = 6B$ and $T = \lceil \lg n - 1 \rceil$ stabilizes the stash, so that the actual storage requirement and bandwidth per operation are $6Bn$ and $12B \lg n$ bits, respectively.

Furthermore, to avoid failure due to stash overflow or collisions, the client storage R and collision parameter γ should both grow slightly faster than $\log n$, i.e., $R, \gamma \in \omega(\log n)$.

5 HIRB Tree Data Structure

We now use the vORAM construction described in the previous section to implement a data structure supporting the operations of a *dictionary* that maps labels to values. In this paper, we intentionally use the word "labels" rather than the word "keys" to distinguish from the encryption *keys* that are stored in the vORAM.

Motivating the HIRB. Before describing the construction and properties of the history independent, randomized B-Tree (HIRB), we first wish to motivate the need for the HIRB as it relates to the security and efficiently requirements of storing it within the vORAM:

- The data structure must be easily partitioned into blocks that have expected size bounded by a geometric distribution for vORAM storage.
- The data structure must be *pointer-based*, and the structure of blocks and pointers must form a directed graph that is an *arborescence*, such that there exists *at most one* pointer to each block. This is because a non-recursive ORAM uses random identifiers for storage blocks, which must change on every read or write to that block.
- The memory access pattern for an operation (e.g., get, set, or delete) must be bounded by a fix parameter to ensure obliviousness; otherwise the number of vORAM accesses could leak information about the data access.
- Finally, the data structure must be *uniquely represented* such that the pointer structures and contents are determined only by the set of $(\mathsf{label}, \mathsf{value})$ pairs stored within, up to some randomization performed during initialization. Recall that *strong history independence* is provided via a unique representation, a sufficient and necessary condition [17] for the desired security property.

In summary, we require a uniquely-represented, tree-based data structure with bounded height. While a variety of uniquely represented (or strongly history independent) data structures have been proposed in the literature [14, 30], we are not aware of any that satisfy all of the requisite properties.

While some form of hash table might seem like an obvious choice, we note that such a structure would violate the second condition above; namely, it would be impossible to store a hash table within an ORAM without having a separate *position map*, incurring an extra logarithmic factor in the cost. As it turns out, our

HIRB tree does use hashing in order to support secure deletion, but this is only to sort the labels within the tree data structure.

Overview of HIRB tree. The closest data structure to the HIRB is the B-Skip List [15]; unfortunately, a skip list does not form a tree. The HIRB is essentially equivalent to a B-Skip List after sorting labels according to a hash function and removing pointers between skip-nodes to impose a top-down tree structure.

Recall that a typical B-tree consists of large nodes, each with an array of $(\mathsf{label}, \mathsf{value})$ pairs and child nodes. A B-tree node has branching factor of k, and we call it a k-node, if the node contains $k-1$ labels, $k-1$ values, and k children (as in Figure 4). In a typical B-tree, the branching factor of each node is allowed to vary in some range $[B+1, 2B]$, where B is a fixed parameter of the construction that controls the maximum size of any single node.

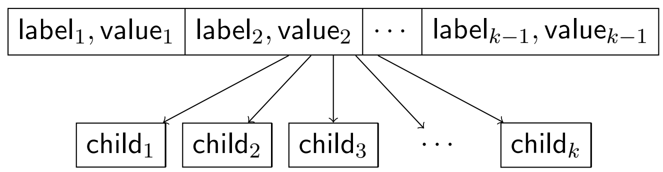

Figure 4: B-tree node with branching factor k

HIRB tree nodes differ from typical B-tree nodes in two ways. First, instead of storing the label in the node a cryptographic hash[3] of the label is stored. This is necessary to support secure deletion of vORAM+HIRB even when the nature of vORAM leaks some history of operations; namely, revealing which HIRB node an item was deleted from should not reveal the label that was deleted.

The second difference from a normal B-tree node is that the branching factor of each node, rather than being limited to a fixed range, can take any value $k \in [1, \infty)$. This branching factor will observe a geometric distribution for storage within the vORAM. In particular, it will be a random variable X drawn independently from a geometric distribution with *expected* value β, where β is a parameter of the HIRB tree construction.

The *height* of a node in the HIRB tree is defined as the length of the path from that node to a leaf node; all leaf nodes are the same distance to the root node for B-trees. The height of a new insertion of $(\mathsf{label}, \mathsf{value})$ in the HIRB is determined by a series of pseudorandom biased coin flips based on the hash of the label[4]. The distribution of selected heights for insertions uniquely determines the structure of the HIRB tree because the process is deterministic, and thus the HIRB is uniquely-represented.

Parameters and preliminaries. Two parameters are fixed at initialization: the *expected branching factor* β, and the *height* H. In addition, throughout this section we will write n as the maximum number of distinct labels that may be stored in the HIRB tree, and γ as a parameter that affects the length of hash digests[5].

A HIRB tree node with branching factor k consists of $k-1$ label hashes, $k-1$ values, and k vORAM identifiers which represent pointers to the child nodes. This is described in Figure 5 where h_i indicates $\mathsf{Hash}(\mathsf{label}_i)$.

Similar to the vORAM itself, the length of the hash function should be long enough to reduce the probability of collision below $2^{-\gamma}$, so define $|\mathsf{Hash}(\mathsf{label})| = \max(2H \lg \beta + \gamma, \lambda)$, and define $\mathsf{nodesize}_k$ to be the size of a HIRB tree node with branching factor k, given as

$$\mathsf{nodesize}_k = (k+1)(2T + \gamma + 1) + k(|\mathsf{Hash}(\mathsf{label})| + |\mathsf{value}|),$$

[3] We need a random oracle for formal security. In practice, we used a SHA1 initialized with a random string chosen when the HIRB tree is instantiated.

[4] Note that this choice of heights is more or less the same as the randomly-chosen node heights in a skip list.

[5] The parameter γ for HIRB and vORAM serves the same purpose in avoiding collisions in identifiers so for simplicity we assume they are the same

id_0	h_1	value_1	id_1	$\cdots$	h_k	value_k	id_k

Figure 5: HIRB node with branching factor k.

where we write $|\mathsf{value}|$ as an upper bound on the size of the largest value stored in the HIRB. (Recall that the size of each vORAM identifier is $2T + \gamma + 1$.) Each HIRB tree *node* will be stored as a single *block* in the vORAM, so that a HIRB node with branching factor k will ultimately be a vORAM block with length $\mathsf{nodesize}_k$.

As β reflects the expected branching factor of a node, it must be an integer greater than or equal to 1. This parameter controls the efficiency of the tree and should be chosen according to the size of vORAM buckets. In particular, using the results of Theorem 5 in the previous section, and the HIRB node size defined above, one would choose β according to the inequality $20\mathsf{nodesize}_\beta \le Z$, where Z is the size of each vORAM bucket. According to our experimental results in Section 6, the constant 20 may be reduced to 6.

The height H must be set so that $H \ge \log_\beta n$; otherwise we risk the root node growing too large. We assume that H is fixed at all times, which is easily handled when an upper bound n is known *a priori*.

HIRB tree operations. As previously described, the entries in a HIRB node are sorted by the hash of the labels, and the *search path* for a label is also according to the label hashes. A lookup operation for a label requires fetching each HIRB node along the search path from the vORAM and returning the matching value.

Initially, an empty HIRB tree of height H is created, as shown in Figure 6. Each node has a branching factor of 1 and contains only the single vORAM identifier of its child.

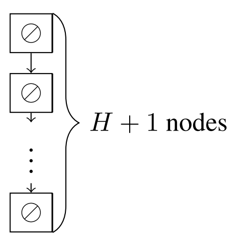

Figure 6: Empty HIRB with height H.

Modifying the HIRB with a set or delete operation on some label involves first computing the height of the label. The height is determined by sampling from a geometric distribution with probability $(\beta - 1)/\beta$, which we derandomize by using a pseudorandom sequence based on $\mathsf{Hash}(\mathsf{label})$. The distribution guarantees that, in expectation, the number of items at height 0 (i.e., in the leaves) is $\frac{\beta-1}{\beta}n$, the number of items at height 1 is $\frac{\beta-1}{\beta^2}n$, and so on.

Inserting or removing an element from the HIRB involves (respectively) splitting or merging nodes along the search path from the height of the item down to the leaf. This differs from a typical B-tree in that rather than inserting items at the leaf level and propagating up or down with splitting or merging, the HIRB tree requires that the heights of all items are fixed. As a result, insertions and deletions occur at the selected height within the tree according to the label hash. A demonstration of this process is provided in Figure 7.

In a HIRB tree with height H, each get operation requires reading *exactly* $H + 1$ nodes from the vORAM, and each set or delete operation involves reading and writing at most $2H + 1$ nodes. To support obliviousness, each operation will require exactly $2H+1$, accomplished by padding with "dummy" accesses so that every operation has an indistinguishable access pattern.

One way of reading and updating the nodes along the search path would be to read all $2H + 1$ HIRB nodes from the vORAM and store them in temporary memory and then write back the entire path after any

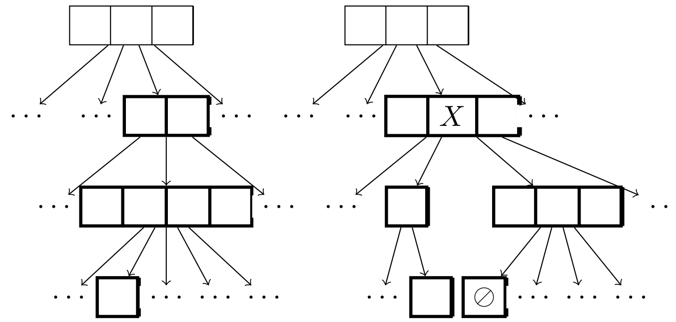

Figure 7: HIRB insertion/deletion of $X = (\mathsf{Hash}(\mathsf{label}), \mathsf{value})$: On the left is the HIRB without item X, displaying only the nodes along the search path for X, and on the right is the state of the HIRB with X inserted. Observe that the insertion operation (left to right) involves splitting the nodes below X in the HIRB, and the deletion operation (right to left) involves merging the nodes below X.

update. However, properties of the HIRB tree enable better performance because the height of each HIRB tree element is uniquely determined, which means we can perform the updates *on the way down* in the search path. This only requires 2 HIRB tree nodes to be stored in local memory at any given time.

Facilitating this extra efficiency requires considerable care in the implementation due to the nature of vORAM identifiers; namely, each internal node must be written back to vORAM before its children nodes are fetched. Fetching children nodes will *change* their vORAM identifiers and invalidate the pointers in the parent node. The solution is to *pre-generate* new random identifiers of the child nodes before they are even accessed from the vORAM.

The full details of the HIRB operations can be found in Appendix B.

HIRB tree properties. For our analysis of the HIRB tree, we first need to understand the distribution of items among each level in the HIRB tree. We assume a subroutine $\mathsf{chooseheight}(\mathsf{label})$ evaluates a random function on label to generate random coins, using which it samples from a truncated geometric distribution with maximum value H and probability $(\beta - 1)/\beta$.

Assumption 6. *If* $\mathsf{label}_1, \ldots, \mathsf{label}_n$ *are any* n *distinct labels stored in a HIRB, then the heights*

$$\mathsf{chooseheight}(\mathsf{label}_1), \ldots, \mathsf{chooseheight}(\mathsf{label}_n)$$

are independent random samples from a truncated geometric distribution over $\{0, 1, \ldots, H\}$ *with probability* $(\beta - 1)/\beta$*, where the randomness is determined entirely by the the random oracle and the random function upon creation of the HIRB.*

In practice, the random coins for $\mathsf{chooseheight}(\mathsf{label})$ are prepared by computing $\mathsf{coins} = \mathrm{PRG}(\mathrm{SHA1}(seed \| \mathsf{label}))$, where $seed$ is a global random seed, and PRG is a pseudorandom generator. With SHA1 modeled as a random oracle, the coins will be pseudorandom.

Theorem 7. *The HIRB tree is a dictionary data structure that associates arbitrary* label*s to* value*s. If it contains* n *items, and has height* $H \geq \log_\beta n$*, and the nodes are stored in a vORAM, then the following properties hold:*

- *The probability of failure in any operation is at most* $2^{-\gamma}$*.*
- *Each operation requires exactly* $2H + 1$ *node accesses, only 2 of which need to be stored in temporary memory at any given time.*
- *The data structure itself,* not *counting the pointers, is strongly history independent.*

The first property follows from the fact that the only way the HIRB tree can fail to work properly is if there is a hash collision. Based on the hash length defined above, the probability that any 2 keys collide amongst the n labels in the HIRB is at most $2^{-\gamma}$. The second property follows from the description of the operations get, set, and delete, and is crucial not only for the performance of the HIRB but also for the obliviousness property. The third property is a consequence of the fact that the HIRB is uniquely represented up to the pointer values, after the hash function is chosen at initialization.

vORAM+HIRB properties. We are now ready for the main theoretical results of the paper, which have to do with the performance and security guaranteed by the vORAM+HIRB construction. These proofs follow in a straightforward way from the results we have already stated on vORAM and on the HIRB, so we leave their proofs to Appendix B.

Theorem 8. *Suppose a HIRB tree with n items and height H is stored within a vORAM with L levels, bucket size Z, and stash size R. Given choices for Z and $\gamma > 0$, set the parameters as follows:*

$$\begin{aligned} T &\geq \lg(4n + \lg n + \gamma) \\ \beta &= \max\{\beta | Z \geq 20 \cdot \mathsf{nodesize}_\beta\} \\ R &\geq \gamma \cdot \mathsf{nodesize}_\beta \\ H &\geq \log_\beta n \end{aligned}$$

Then the probability of failure due to stash overflow or collisions after each operation is at most

$$\Pr[\textit{vORAM+HIRB failure}] \leq 30 \cdot (0.883)^\gamma.$$

The parameters follow from the discussion above. Again note that the constants 30 and 0.883 are technical artifacts of the analysis.

Theorem 9. *Suppose a vORAM+HIRB is constructed with parameters as above. The vORAM+HIRB provides obliviousness, secure deletion, and history independence with leakage of $O(n + n\lambda/(\log n))$ operations.*

The security properties follow from the previous results on the vORAM and the HIRB. Note that the HIRB structure itself provides history independence with no leakage, but when combined with the vORAM, the pointers may leak information about recent operations. The factor $O(\log n)$ difference from the amount of leakage from vORAM in Theorem 4 arises because each HIRB operation entails $O(\log n)$ vORAM operations. Following the discussion after Theorem 4, we could also reduce the leakage in vORAM+HIRB to $O(n/\log^2 n)$, with constant-factor increase in bandwidth, which again is optimal according to Theorem 1.

6 Evaluation

We completed two empirical analyses of the vORAM+HIRB system. First, we sought to determine the most effective size for vORAM buckets with respect to the expected block size, i.e., the ratio Z/B. Second, we made a complete implementation of the vORAM+HIRB and measured its performance in storing a realistic dataset of key/value pairs of 22MB in size. The complete source code of our implementation is available upon request.

6.1 Optimizing vORAM parameters

A crucial performance parameter in our vORAM construction is the ratio Z/B between the size Z of each bucket and the expected size B of each block. (Note that $B = \mathsf{nodesize}_\beta$ when storing HIRB nodes within

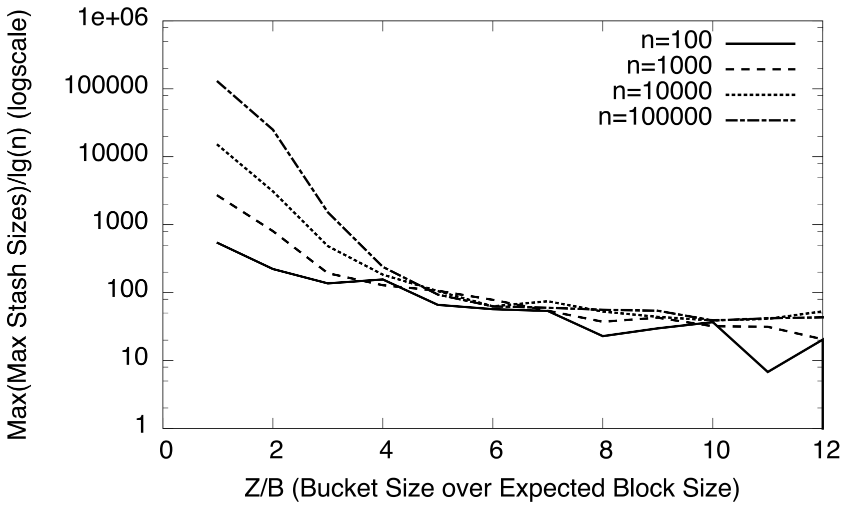

Figure 8: Maximum stash size, scaled by $\log n$, observed across 50 simulations of a vORAM for various Z/B values.

the vORAM.) This ratio is a constant factor in the bandwidth of every vORAM operation and has a considerable effect on performance. In the Path ORAM, the best corresponding theoretical ratio is 5, whereas it has been shown experimentally that a ratio of 4 will also work, even in the worst case [37].

We performed a similar experimental analysis of the ratio Z/B for the vORAM. Our best theoretical ratio from Theorem 5 is 20, but as in related work, the experimental performance is better. The goal is then to find the optimal, empirical choice for the ratio Z/B: If Z/B is too large, this will increase the overall communication cost of the vORAM, and if it is too small, there is a risk of stash overflow and loss of data or obliviousness.

For the experiments described below, we implemented a vORAM structure without encryption and inserted a chosen number of variable size blocks whose sizes were randomly sampled from a geometric distribution with expected size 68 bytes. To avoid collisions, we ensured the identifier lengths satisfied $\gamma \geq 40$.

Stash size. To analyze the stash size for different Z/B ratios, we ran a number of experiments and monitored the maximum stash size observed at any point throughout the experiment. Recall, while the stash will typically be empty after every operation, the max stash size should grow logarithmically with respect to the number of items inserted in the vORAM. The primary results are presented in Figure 8.

This experiment was conducted by running 50 simulations of a vORAM with n insertions and a height of $T = \lg n$. The Z/B value ranged from 1 to 50, and results in the range 1 through 12 are presented in the graph for values of n ranging from 10^2 through 10^5. The graph plots the ratio $R/\lg n$, where R is the largest max stash size at any point in any of the 50 simulations. Observe that between $Z/B = 4$ and $Z/B = 6$ the ratio stabilizes for all values of n, indicating a maximum stash of approximately $100 \lg n$.

In order to measure how much stash would be needed in practice for much larger experimental runs, we fixed $Z/B = 6$ and for three large database sizes, $n = 2^{18}, 2^{19}, 2^{20}$, For each size, we executed $2n$ operations, measuring the size of stash after each. In practice, as we would assume from the theoretical results, the stash size is almost always zero. However, the stash does occasionally become non-empty, and it is precisely the frequency and size of these rare events that we wish to measure.

Figure 9 shows the result of our stash overflow experiment. We divided each test run of roughly $2n$ operations into roughly n overlapping windows of n operations each, and then for each window, and each possible stash size, calculated the number of operations before the first time that stash size was exceeded. The average number of operations until this occurred, over all n windows, is plotted in the graph. The data

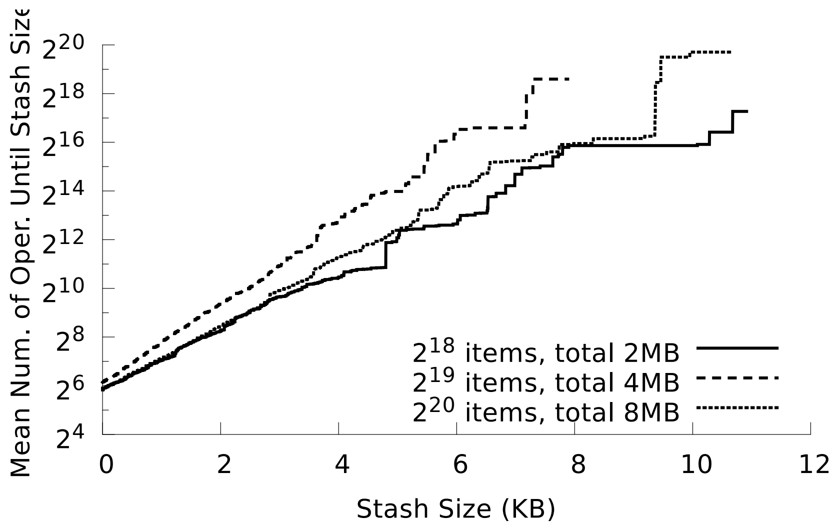

Figure 9: Average time until stash overflow, for varying vORAM and stash sizes. Stash size is linear-scale, number of operations in log-scale. Higher is better. For each vORAM size n, we performed $2n$ operations to gather sufficient experimental data.

—

shows a linear trend in log-scale, meaning that the stash size necessary to ensure low overflow probability after N operations is $O(\log N)$, as expected. Furthermore, in all experiments we never witnessed a stash size larger than roughly 10KB, whereas the theoretical bound of $100 \lg n$ items would be 16KB for the largest test with 2^{20} 8-byte items.

Bucket utilization. Stash size is the most important parameter of vORAM, but it provides a limited view into the optimal bucket size ratio, in particular as the stash overflow is typically zero after every operation, for sufficiently large buckets. We measured the utilization of buckets at different levels of the vORAM with varied heights and Z/B values. The results are presented in Figure 10 and were collected by averaging the final bucket utilization from 10 simulations. The utilization at each level is measured by dividing the total storage capacity of the level by the number of bytes at the level. In all cases, $n = 2^{15}$ elements were inserted, and the vORAM height varied between 14, 15, and 16. The graph shows that with height $\lg n = 15$ or higher and Z/B is 6 or higher, utilization stabilizes throughout all the levels (with only a small spike at the leaf level).

The results indicate, again, that when $Z/B = 6$, the utilization at the interior buckets stabilizes. With smaller ratios, e.g., $Z/B = 4$, the utilization of buckets higher in the tree dominates those lower in the tree; essentially, blocks are not able to reach lower levels resulting in higher stash sizes (see previous experiment). With larger ratios, which we measured all the way to $Z/B = 13$, we observed consistent stabilization.

In addition, our data shows that decreasing the number of levels from $\lg n$ to $\lg n - 1$ (e.g., from 15 to 14 in the figure) increases utilization at the leaf nodes as expected (as depicted in the spike in the tail of the graphs), but when $Z/B \geq 6$ the extra blocks in leaf nodes do *not* propagate up the tree and affect the stash. It therefore appears that in practice, the number of levels T could be set to $\lg n - 1$, which will result in a factor of 2 savings in the size of persistent (cloud) storage due to high utilization at the leaf nodes. This follows a similar observation about the height of the Path ORAM made by [37].

6.2 Measuring vORAM+HIRB Performance

We measured the performance of our vORAM+HIRB implementation on a real data set of reasonable size, and compared to some alternative methods for storing a remote map data structure that provide varying

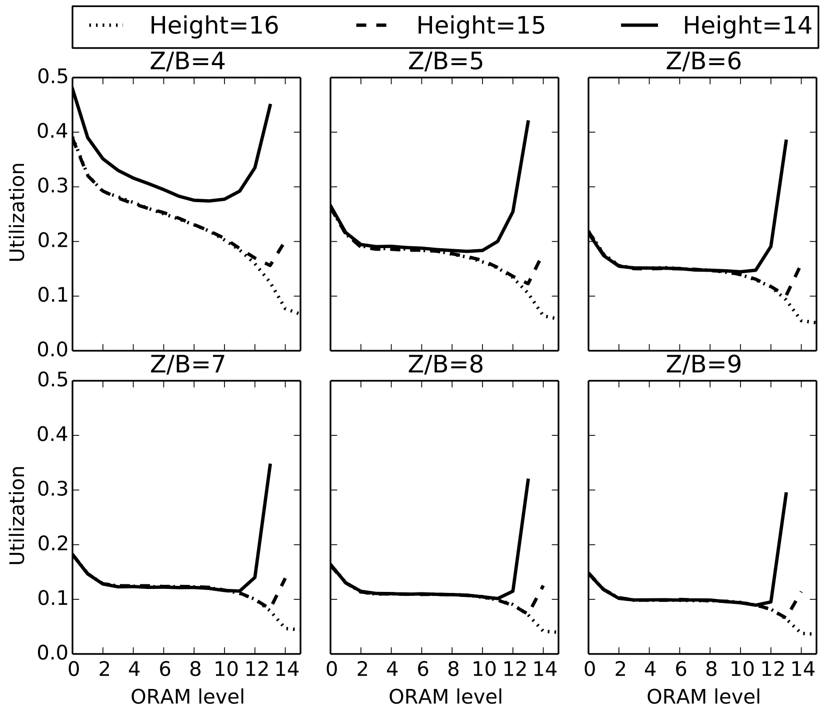

Figure 10: Utilization at different levels of ORAM

levels of security and efficiency. All of our implementations used the same client/server setup, with a Python3 implementation and AWS as the cloud provider, in order to give a fair comparison.

Sample dataset. We tested the performance of our implementation on a dataset of 300,000 synthetic key/value pairs where where keys were variable sizes (in the order of bytes 10-20 bytes) and values were fixed at 16 bytes. The total unencrypted data set is 22MB in size. In our experiments, we used some subset of this data dependent on the size of the ORAM, and for each size, we also assumed that the ORAM user would want to allow the database to grow. As such, we built the ORAM to double the size of the initialization.

Optimized vORAM+HIRB implementation. We fully implemented our vORAM+HIRB map data structure using Python3 and Amazon Web Services as the cloud service provider. We used AES256 for encryption in vORAM, and used SHA1 to generate labels for the HIRB. In our setting, we considered a client running on the local machine that maintained the erasable memory, and the server (the cloud) provided the persistent storage with a simple get/set interface to store or retrieve a given (encrypted) vORAM bucket.

For the vORAM buckets, we choose $Z/B = 6$ based on the prior experiments, and a bucket size of 4K, which is the preferred back-end transfer size for AWS, and was also the bucket size used by [4]. One of the advantages of the vORAM over other ORAMs is that the bucket size can be set to match the storage requirements with high bucket utilization. The settings for the HIRB were then selected based on Theorem 8 and based on that, we calculated a $\beta = 12$ for the sample data (labels and values) stored within the HIRB. The label, value, and associated vORAM identifiers total 56 bytes per item.

In our experiments, we found that the round complexity of protocols dominate performance and so we made a number of improvements and optimizations to the vORAM access routines to compensate. The result is an optimized version of the vORAM. In particular:

- *Parallelization*: The optimized vORAM transfers buckets along a single path in parallel over simultaneous connections for both the evict and writeback methods. Our experiments used up to T threads in parallel to fetch and send ORAM block files, and each maintained a persistent `sftp` connection.

- *Buffering*: A local buffer storing $2T$ top-most ORAM buckets was used to facilitate asynchronous path reading and writing by our threads. Note the size of the client storage still remains with $O(\log n)$ since $T = O(\log n)$. This had an added performance benefit beyond the parallelization because the top few levels of the ORAM generally resided in the buffer and did not need to be transferred back and forth to the cloud after every operation.

These optimizations had a considerable effect on the performance. We did not include the cost of the ≈ 2 second setup/teardown time for these SSL connections in our results as these were a one-time cost incurred at initialization. Many similar techniques to these have been used in previous work to achieve similar performance gains (e.g., [25, 41]), although they have not been previously applied to oblivious data structures.

Comparison baselines. We compared our optimized vORAM+HIRB construction with four other alternative implementations of a remote map data structure, with a wide range of performance and security properties:

- *Un-optimized vORAM+HIRB*. This is the same as our normal vORAM+HIRB construction, but without any buffering of vORAM buckets and with only a single concurrent sftp connection. This comparison allows us to see what gains are due to the algorithmic improvements in vORAM and HIRB, and which are due to the network optimizations.

- *Naive Baseline*: We implemented a naive approach that provides all three security properties, obliviousness, secure deletion, and history independence. The method involves maintaining a single, fixed-size encrypted file transferred back and forth between the server and client and re-encrypted on each access. While this solution is cumbersome for large sizes, it is the obvious solution for small databases and thus provides a useful baseline. Furthermore, we are not aware of any other method (other than vORAM+HIRB) to provide obliviousness, secure-deletion, and history independence.

- *ORAM+AVL*: We implemented the ODS proposed by [40] of an AVL embedded within an non-recursive Path ORAM. Note that ORAM+AVL does not provide secure deletion nor history independence. We used the same cryptographic settings as our vORAM+HIRB implementation, and used 256 byte blocks for each AVL node, which was the smallest size we could achieve without additional optimizations. As recommended by [37], we stored $Z = 4$ fixed-size blocks in each bucket, for a total of 1K bucket size. Note that this bucket size is less than the 4K transfer size recommended by the cloud storage, which reflects the limitation of ORAM+AVL in that it cannot effectively utilize larger buckets. We add the observation that, when the same experiments were run with 4K size buckets (more wasted bandwidth, but matching the other experiments), the timings did not change by more than 1 second, indicating that the 4K bucket size is a good choice for the AWS back-end.

- *SD-B-Tree*: As another comparison point, we implemented a remotely stored B-Tree with secure deletion where each node is encrypted with a key stored in the parent with re-keying for each access, much as described by Reardon et al. [34]. While this solution provides secure deletion, and stores all data encrypted, it does not provide obliviousness nor history independence. Again, we used AES256 encryption, with $\beta = 110$ for the B-tree max internal node size in order to optimize 4K-size blocks.

In terms of security, only our vORAM+HIRB as well as the naive baseline provide obliviousness, secure deletion, and history independence. The ORAM+AVL provides obliviousness only, and the SD-B-Tree is most vulnerable to leaking information in the cloud, as it provides secure deletion only.

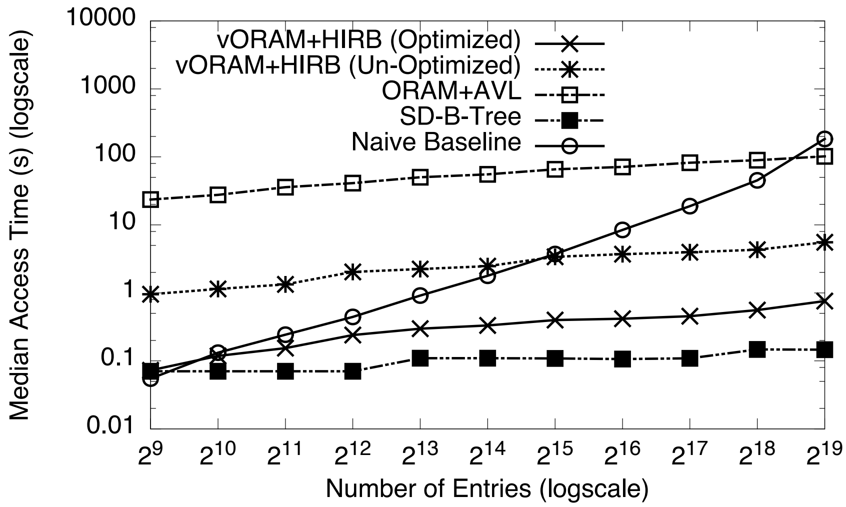

Figure 11: Median of 100 access times for different number of entries

In terms of asymptotic performance, the SD-B-Tree is fastest, requiring only $O(\log n)$ data transfer per operation. The vORAM+HIRB and ORAM+AVL both require $O(\log^2 n)$ data transfer per operation, although as discussed previously the vORAM+HIRB saves a considerable constant factor. The naive baseline requires $O(n)$ transfer per operation, albeit with the smallest possible constant factor.

Experimental results. The primary result of the experiment is presented in Figure 11 where we compared the cost of a single access time (in seconds) across the back end storage (note, graph is log-log). Unsurprisingly, the SD-B-Tree implementation is fastest for sufficiently large database sizes. However, our optimized vORAM+HIRB implementation was competitive to the SD-B-Tree performance, both being less than 1 second across our range of experiments.

Most striking is the access time of ORAM+AVL compared to the vORAM+HIRB implementations. In both the optimized and un-optimized setting, the vORAM+HIRB is orders of magnitude faster than ORAM+AVL, 20X faster un-optimized and 100X faster when optimized. Even for a relatively small number of entries such as 2^{11}, a single access of ORAM+AVL takes 35 seconds, while it only requires 1.3 seconds of un-optimized vORAM+HIRB and 0.2 second of an optimized implementation. It is not until 2^{19} entries that ORAM+AVL even outperforms the naive $O(n)$ baseline solution.

As described previously, we attribute much of the speed to decreasing the round complexity. The HIRB tree requires much smaller height as compared to an AVL tree because each HIRB node contains β items on average as compared to just a single item for an AVL tree. Additionally, the HIRB's height is fixed and does not require padding to achieve obliviousness. Each AVL operation entails $3 \cdot 1.44 \lg N$ ORAM operations as compared to just $2 \log_\beta N$ vORAM operations for the HIRB. This difference in communication cost is easily observed in Table 1. Overall, we see that the storage and communication costs for vORAM+HIRB are not too much larger than that for a secure deletion B-tree, which does not provide any access pattern hiding as the oblivious alternatives do.

(The values in this table were generated by considering the worst-case costs in all cases, for our actual implementations, but considering only a single operation. Note that, for constructions providing obliviousness, every operation must *actually* follow this worst case cost, and so the comparison is fair.)

Put simply, the vORAM+HIRB and SD-B-Tree are the only implementations which can be considered practical for real data sizes, and the benefit of vORAM+HIRB is its considerable additional security guarantees of oblivious and bounded history independence.

Size 2^{10}			
	Total storage	Bandwidth	Rounds
Naive baseline	8.2 KB	8.2 KB	1
Secure deletion B-tree	36.9 KB	12.3 KB	2
ORAM+AVL	8.4 MB	4.0 MB	968
vORAM+HIRB	**127.0 KB**	**102.4 KB**	**3**
Size 2^{15}			
	Total storage	Bandwidth	Rounds
Naive baseline	262.1 KB	262.1 KB	1
Secure deletion B-tree	1.1 MB	20.5 KB	3
ORAM+AVL	268.4 MB	8.6 MB	2096
vORAM+HIRB	**4.2 MB**	**286.7 KB**	**4**
Size 2^{20}			
	Total storage	Bandwidth	Rounds
Naive baseline	8.4 MB	8.4 MB	1
Secure deletion B-tree	33.8 MB	20.5 KB	3
ORAM+AVL	8.6 GB	15.1 MB	3675
vORAM+HIRB	**134.2 MB**	**553.0 KB**	**5**
Size 2^{25}			
	Total storage	Bandwidth	Rounds
Naive baseline	268.4 MB	268.4 MB	1
Secure deletion B-tree	1.1 GB	28.7 KB	4
ORAM+AVL	274.9 GB	23.2 MB	5668
vORAM+HIRB	**4.3 GB**	**901.1 KB**	**6**
Size 2^{30}			
	Total storage	Bandwidth	Rounds
Naive baseline	8.6 GB	8.6 GB	1
Secure deletion B-tree	34.6 GB	36.9 KB	5
ORAM+AVL	8.8 TB	33.3 MB	8122
vORAM+HIRB	**137.4 GB**	**1.5 MB**	**8**

Table 1: Storage and communication cost comparisons. Total storage is the amount of space required for the server, and the bandwidth and rounds are counted *per operation*. Each stored item consists of a 4-byte label and 4-byte value.

7 Conclusion

In this paper, we have shown a new secure cloud storage system combining the previously disjoint security properties of obliviousness, secure deletion, and history independence. This was accomplished by developing a new variable block size ORAM, or vORAM, and a new history independent, randomized data structure (HIRB) to be stored within the vORAM.

The theoretical performance of our vORAM+HIRB construction is competitive to existing systems which provide fewer security properties. Our implemented system is up to 100X faster (w.r.t. access time) than current best oblivious map data structure (which provides no secure deletion or history independence) by Wang et al. (CCS 14), bringing our single-operation time for a reasonable-sized database ($> 2^{19}$) to less than 1 second per access.

There much potential for future work in this area. For example, one could consider data structures that support a richer set of operations, such as range queries, while preserving obliviousness, secure deletion, and history independence. Additionally, the vORAM construction in itself may provide novel and exciting new analytic results for ORAMs generally by not requiring fixed bucket sizes. There is a potential to improve the overall utilization and communication cost compared to existing ORAM models that used fixed size blocks.

Finally, while we have demonstrated the practicality in terms of overall per-operation speed, we did not consider some additional practical performance measures as investigated by [4], such as performing asynchronous operations and optimizing upload vs download rates. Developing an ODS map considering these concerns as well would be a useful direction for future work.

Acknowledgements

The authors are supported by the Office of Naval Research (ONR), as well as National Science Foundation awards #1406192 and #1319994.

References

[1] Lastpass security notice. https://blog.lastpass.com/2015/06/lastpass-security-notice.html/.

[2] Sumeet Bajaj and Radu Sion. Ficklebase: Looking into the future to erase the past. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 86–97, 2013.

[3] Sumeet Bajaj, Anrin Chakraborti, and Radu Sion. Practical foundations of history independence. *CoRR*, abs/1501.06508, 2015.

[4] 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 *ACM CCS 15*, 2015. To appear.

[5] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS 13*, pages 207–218. ACM Press, May 2013.

[6] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *48th FOCS*, pages 272–282. IEEE Computer Society Press, October 2007.

[7] Dan Boneh and Richard J. Lipton. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*, 1996. URL `https://www.usenix.org/conference/6th-usenix-security-symposium/revocable-backup-system`.

[8] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. *Inf. Comput.*, 204(2):291–337, 2006. doi: 10.1016/j.ic.2005.11.001. URL `http://dx.doi.org/10.1016/j.ic.2005.11.001`.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, September 2001. ISBN 0262032937.

[10] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In *STACS*, pages 500–509, 1999.

[11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 144–163. Springer, March 2011.

[12] Jonathan L Dautrich Jr and Chinya V Ravishankar. Compromising privacy in precise query protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 155–166. ACM, 2013.

[13] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. doi: 10.1145/233551.233553. URL `http://doi.acm.org/10.1145/233551.233553`.

[14] Daniel Golovin. B-treaps: A uniquely represented alternative to B-trees. In *ICALP 2009, Part I*, volume 5555 of *LNCS*, pages 487–499. Springer, July 2009.

[15] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *CoRR*, abs/1005.0662, 2010.

[16] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *23rd SODA*, pages 157–167. ACM-SIAM, January 2012.

[17] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily C. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005. doi: 10.1007/s00453-004-1140-z.

[18] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963. ISSN 0162-1459. URL `http://www.jstor.org/stable/2282952`.

[19] Mike Isaac. Nude photos of jennifer lawrence are latest front in online privacy debate. *New York Times*, Sept. 2, 2014. http://www.nytimes.com/2014/09/03/technology/trove-of-nude-photos-sparks-debate-over-online-behavior.html.

[20] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, February 2012.

[21] Jonathan L. Dautrich Jr., Emil Stefanov, and Elaine Shi. Burst ORAM: minimizing ORAM response times for bursty access patterns. In *Proceedings of the 23rd USENIX Security Symposium*, pages 749–764, 2014. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dautrich`.

[22] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007. ISBN 978-1-58488-551-1.

[23] Frank Konkel. The details about the CIA's deal with Amazon. *The Atlantic*, Jul. 17, 2014. http://www.theatlantic.com/technology/archive/2014/07/the-details-about-the-cias-deal-with-amazon/374632/.

[24] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd SODA*, pages 143–156. ACM-SIAM, January 2012.

[25] 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*, volume 2013, pages 199–213, 2013.

[26] Daniele Micciancio. Oblivious data structures: Applications to cryptography. In *29th ACM STOC*, pages 456–464. ACM Press, May 1997.

[27] John C. Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, pages 554–565, 2014. doi: 10.4230/LIPIcs.STACS.2014.554. URL `http://dx.doi.org/10.4230/LIPIcs.STACS.2014.554`.

[28] Soumyadeb Mitra, Marianne Winslett, and Nikita Borisov. Deleting index entries from compliance storage. In *EDBT 2008, 11th International Conference on Extending Database Technology*, pages 109–120, 2008. doi: 10.1145/1353343.1353361. URL `http://doi.acm.org/10.1145/1353343.1353361`.

[29] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 862–873, 2015.

[30] Moni Naor and Vanessa Teague. Anti-presistence: History independent data structures. In *33rd ACM STOC*, pages 492–501. ACM Press, July 2001.

[31] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 631–642. Springer, July 2008.

[32] Zachary N. J. Peterson, Randal C. Burns, Joseph Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, 2005. URL `http://www.usenix.org/events/fast05/tech/peterson.html`.

[33] Joel Reardon, David A. Basin, and Srdjan Capkun. Sok: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, pages 301–315, 2013. doi: 10.1109/SP.2013.28. URL `http://dx.doi.org/10.1109/SP.2013.28`.

[34] Joel Reardon, Hubert Ritzdorf, David A. Basin, and Srdjan Capkun. Secure data deletion from persistent media. In *ACM CCS 13*, pages 271–284. ACM Press, November 2013.

[35] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430, 2015.

[36] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *NDSS*, 2012.

[37] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 13*, pages 299–310. ACM Press, November 2013.

[38] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcherand Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *CoRR*, abs/1202.5150v3, 2014. URL `http://arxiv.org/abs/1202.5150v3`.

[39] Tomas Toft. Brief announcement: Secure data structures based on multi-party computation. In Cyril Gavoille and Pierre Fraigniaud, editors, *30th ACM PODC*, pages 291–292. ACM, June 2011.

[40] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 215–226. ACM Press, November 2014.

[41] Xiangyao Yu, Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Enhancing oblivious ram performance using dynamic prefetching. *IACR Cryptology ePrint Archive*, 2014:234, 2014.

[42] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS 2004*, pages 72–84, 2004.

A vORAM Operation Details

The full detail of the vORAM helper functions is provided in Figure 12, and the three main operations are shown in Figure 13.

B HIRB Operation Details

We described the HIRB data structure in Section 5. The full details of the different subroutines are provided in Figures 14 and 15.

All the HIRB tree operations depend on a subroutine $\mathsf{HIRBpath}$, which given a label hash, HIRB root node identifier, and vORAM, generates tuples $(\ell, v_0, v_1, cid_1^+)$ corresponding to the search path for that label in the HIRB. In each tuple, ℓ is the level of node v_0, which is along the search path for the label. In the $initial$ part of the search path, that is, before the given label hash is found, node v_1 is always nil, a dummy access used to preserve obliviousness. The value cid_1^+ is the *pre-generated* identifier of the new node that will be inserted on the *next level*, for possible inclusion in one of the parent nodes as a child pointer. This pre-generation is important, as discussed in Section 5, so that only 2 nodes need to be stored in local memory at any given time.

When the given label hash is found, the search path splits into two below that node, and nodes v_0 and v_1 will be the nodes on either side of that hash label. Note that in the actual implementation of $\mathsf{HIRBpath}$, v_0

$\mathsf{idgen}()$

Choose $r \leftarrow \{0,1\}^{2T+\gamma}$.
return $1\|r$.

$\mathsf{loc}(id, t)$

return the location of the node at level t along the path from the root to the leaf node identified by id. This is is simply the index indicated by the $(t+1)$ most significant bits of id.

$\mathsf{evict}(id)$

$key \leftarrow rootkey$ ▷ $rootkey$: enc key for root bucket
$B \leftarrow$ empty list
for $t = 0, 1, \ldots T$ **do**
  remove bucket at $\mathsf{loc}(id, t)$ from persistent storage
   decrypt it with key
  Append all partial blocks in the bucket to the end of B
  $key \leftarrow$ child key from bucket according to $\mathsf{loc}(id, t+1)$
end for
for each partial block (id^*, ℓ, blk) in B **do**
  if (id^*, ℓ_0, blk_0) is in stash already
  then replace with $(id^*, \ell_0 + \ell, blk_0\, blk)$ ▷ merge
  else Add (id^*, ℓ, blk) to stash
end for

$\mathsf{writeback}(id)$

$key \leftarrow \mathsf{nil}$
for $t = T, T-1, \ldots, 0$ **do**
  $W \leftarrow \{(id^*, \ell, blk) \in stash : \mathsf{loc}(id^*, t) = \mathsf{loc}(id, t)\}$
  ▷ W is the partial blocks storable in the bucket
  create empty bucket with new child key key
   (other child key remains the same)
  while W is not empty and bucket is not full **do**
    $(id^*, \ell, blk) \leftarrow$ arbitrary element from W
    $(id^*, \ell_1, blk_1) \leftarrow$ largest partial block of the above, fitting
     in the bucket with $blk = blk_0\, blk_1$ and $|blk_1| = \ell_1$.
    Add (id^*, ℓ_1, blk_1) to the bucket
    if $\ell_1 = \ell$
    then remove (id^*, ℓ, blk) from W and from stash
    else replace (id^*, ℓ, blk) in stash with $(id^*, \ell - \ell_1, blk_0)$.
    ▷ split a partial block
  end while
  $key \leftarrow \{0,1\}^\lambda$ chosen uniformly at random
  insert $\mathsf{Enc}_{key}(\text{bucket})$ at $\mathsf{loc}(id, t)$ in persistent storage.
end for
$rootkey \leftarrow key$

Figure 12: vORAM helper functions

$\underline{\mathsf{insert}(blk)}$

$id_0 \leftarrow \mathsf{idgen}()$
$\mathsf{evict}(id_0)$
$id^+ \leftarrow \mathsf{idgen}()$
insert $(id^+, |blk|, blk)$ into stash
$\mathsf{writeback}(id_0)$
return id^+

$\underline{\mathsf{remove}(id)}$

$\mathsf{evict}(id)$
remove (id, ℓ, blk) from stash
$\mathsf{writeback}(id)$
return blk

$\underline{\mathsf{update}(id, callback)}$

$\mathsf{evict}(id)$
remove (id, ℓ, blk) from stash
$id^+ \leftarrow \mathsf{idgen}()$
$blk^+ \leftarrow callback(blk)$
insert $(id^+, |blk^+|, blk^+)$ into stash
$\mathsf{writeback}(id)$
return id^+

Figure 13: vORAM operations

(resp. v_1, if defined) corresponds to a vORAM block, evicted with identifier id_0 (resp. id_1) and taken out from vORAM stash. When each tuple $(\ell, v_0, v_1, cid_1^+)$ is returned from the generator, the two nodes can be modified by the calling function, and the modified nodes will be written back to the HIRB. If v_1 is returned from HIRBpath as nil, but is then modified to be a normal HIRB node, that new node is subsequently inserted into the HIRB.

The update operation simply looks in each returned v_0 along the search path for the existence of the indicated label hash, and if found, the corresponding data value is passed to the *callback* function, possibly modifying it.

As with update, the insert operation uses subroutine HIRBpath as a generator to traverse the HIRB tree. Inserting an element from the HIRB involves splitting nodes along the search path from the height of the item down to the leaf. That is, for each tuple (ℓ, v_0, v_1) with $\ell > \ell_h$, where ℓ_h is the height of the label hash h, if v_1 is nil, then a new node v_1 is created, and the items in v_0 with a label greater than h are moved to a new node v_1.

The remove operation works similarly, but instead of splitting each v_0 below the level of the found item, the values in v_0 and v_1 are merged into v_0, and v_1 is removed by setting it to nil.

C Proofs of Important Theorems

Complete proofs of our main theorems are given here.

$\underline{\mathsf{HIRBpath}(h, rootid, \mathsf{M})}$ ▷ M is vORAM

$(id_0,\ id_0^+) \leftarrow (rootid,\ \mathsf{M.idgen}())$
$rootid \leftarrow id_0^+$
$(id_1,\ id_1^+) \leftarrow (\mathsf{M.idgen}(),\ \mathsf{M.idgen}())$ ▷ dummy access
$found \leftarrow \mathsf{false}$
for $\ell = 0, 1, 2, \ldots, H$ **do**
    $\mathsf{M.evict}(id_0)$
    $\mathsf{M.evict}(id_1)$
    if $\ell = H$ **then** $(cid_0^+,\ cid_1^+) \leftarrow (\mathsf{nil},\ \mathsf{nil})$
    else $(cid_0^+,\ cid_1^+) \leftarrow (\mathsf{M.idgen}(),\ \mathsf{M.idgen}())$
    remove $(id_0, |v_0|, v_0)$ from $\mathsf{M.stash}$
    if $found = \mathsf{true}$ **then**
        remove $(id_1, |v_1|, v_1)$ from $\mathsf{M.stash}$
        $(cid_0,\ v_0.\mathsf{child}_{last}) \leftarrow (v_0.\mathsf{child}_{last},\ cid_0^+)$
        $(cid_1,\ v_1.\mathsf{child}_0) \leftarrow (v_1.\mathsf{child}_0,\ cid_1^+)$
        ▷ v_1 is right next to v_0 at level ℓ
    else
        $v_1 \leftarrow \mathsf{nil}$ ▷ only fetched after the target is found.
        $i \leftarrow$ index of h in v_0 ▷ $v_0.h_{i-1} < h \le v_0.h_i$
        $(cid_0,\ v_0.\mathsf{child}_i) \leftarrow (v_0.\mathsf{child}_i,\ cid_0^+)$
        if $v_0.h_i = h$ **then**
            $found \leftarrow \mathsf{true}$
            $(cid_1,\ v_0.\mathsf{child}_{i+1}) \leftarrow (v_0.\mathsf{child}_{i+1},\ cid_1^+)$
            ▷ split path: $cid_0 = v_0.\mathsf{child}_i$, $cid_1 = v_0.\mathsf{child}_{i+1}$
        else
            $cid_1 \leftarrow \mathsf{M.idgen}()$ ▷ dummy access until found
        end if
    end if
    yield $(\ell,\ v_0,\ v_1, cid_1^+)$
    ▷ Return to the caller, who may modify nodes.
    insert $(id_0^+, |v_0|, v_0)$ into $\mathsf{M.stash}$
    if $v_1 \neq \mathsf{nil}$ **then** insert $(id_1^+, |v_1|, v_1)$ into $\mathsf{M.stash}$
    $\mathsf{M.writeback}(id_0)$
    $\mathsf{M.writeback}(id_1)$
    $(id_0,\ id_0^+) \leftarrow (cid_0,\ cid_0^+)$
    $(id_1,\ id_1^+) \leftarrow (cid_1,\ cid_1^+)$
end for

Figure 14: Fetching the nodes along a search path in the HIRB

$\underline{\mathsf{hirbinit}(H, \mathsf{M})}$

$rootid \leftarrow \mathsf{nil}$
$salt \leftarrow \{0,1\}^\lambda$. Initialize Hash with $salt$.
for $\ell = H, H-1, \ldots, 0$ **do**
  $node \leftarrow$ new 1-ary HIRB node with child id $rootid$
  $rootid \leftarrow \mathsf{M.insert}(node)$
end for
return $rootid$

$\underline{\mathsf{chooseheight}(\mathsf{label})}$

$h \leftarrow \mathsf{Hash}(\mathsf{label})$
Choose coins $(c_0, c_1, \ldots, c_{H-1}) \in \{0, 1, \ldots, \beta - 1\}^H$ by evaluating PRG(h).
return The largest integer $\ell \in \{0, 1, \ldots, H\}$ such that $c_1 = c_2 = \cdots = c_\ell = 0$.

$\underline{\mathsf{insert}(\mathsf{label}, \mathsf{value}, rootid, \mathsf{M})}$

$(h,\ \ell_h) \leftarrow (\mathsf{Hash}(\mathsf{label}),\ \mathsf{chooseheight}(\mathsf{label}))$
for $(\ell, v_0, v_1, cid_1^+) \in \mathsf{HIRBpath}(h, rootid, \mathsf{M})$ **do**
  $i \leftarrow$ index of h in v_0 ▷ $v_0.h_{i-1} < h \le v_0.h_i$
  if $v_0.h_i = h$ **then**
    $v_0.\mathsf{value}_i \leftarrow \mathsf{value}$
  else if $\ell = \ell_h$ **then**
    Insert $(h, \mathsf{value}, cid_1^+)$ before $(v_0.h_i, v_0.\mathsf{value}_i, v_0.\mathsf{child}_i)$ ▷ Other items in v_0 are shifted over
  else if $\ell > \ell_h$ and $v_1 = \mathsf{nil}$ **then**
    $v_1 \leftarrow$ new node with $v_1.\mathsf{child}_0 \leftarrow cid_1^+$
    Move items in v_0 past index i into v_1
  end if
end for

$\underline{\mathsf{remove}(\mathsf{label}, rootid, \mathsf{M})}$

$(h,\ \ell_h) \leftarrow (\mathsf{Hash}(\mathsf{label}),\ \mathsf{chooseheight}(\mathsf{label}))$
for $(\ell, v_0, v_1, cid_1^+) \in \mathsf{HIRBpath}(\mathsf{label}, rootid, \mathsf{M})$ **do**
  if $h \in v_0$ **then**
    Remove h and its associated value and subtree from v_0
  else if $\ell > \ell_h$ and $v_1 \neq \mathsf{nil}$ **then**
    Add all items in v_1 except $v_1.\mathsf{child}_0$ to v_0
    $v_1 \leftarrow \mathsf{nil}$
  end if
end for

$\underline{\mathsf{update}(\mathsf{label}, callback, rootid, \mathsf{M})}$

$h \leftarrow \mathsf{Hash}(\mathsf{label})$
for $(\ell, v_0, v_1) \in \mathsf{HIRBpath}(h, rootid, \mathsf{M})$ **do**
  $i \leftarrow$ index of h in v_0
  if $v_0.h_i = h$ **then** $v_0.\mathsf{value}_i \leftarrow callback(v_0.\mathsf{value}_i)$
end for

Figure 15: Description of HIRB tree operations.

C.1 Proof of Theorem 1

Let $\mathcal{D}$ be any system that stores blocks of data in persistent storage and erasable memory and supports insert and remove operations, accessing at most k bytes in persistent or local storage in each insert or remove operation.

Let $n \geq 36$ and $k \leq \sqrt{n}/2$. For any $\ell \leq n/(4k)$, we describe a PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that breaks history independence with leakage of ℓ operations.

Supposing all operations are insertions, $\mathcal{D}$ must access the location where that item's data is actually to be stored during execution of the insert operation, which is required to correctly store the data somehow. However, it may access some other locations as well to "hide" the access pattern from a potential attacker. This hiding is limited of course by k, which we will now exploit.

The "chooser", $\mathcal{A}_1$, randomly chooses n items which will be inserted; these could simply be random bit strings of equal length. Call these items (and their arbitrary order) $a_1, a_2, \ldots, a_n$. The chooser also randomly picks an index $j \in \{1, 2, \ldots, n - \ell - 1\}$ from the beginning of the sequence. The operation sequence $\overrightarrow{\mathsf{op}}^{(0)}$ returned by $\mathcal{A}_1$ consists of n insertion operations for $a_1, \ldots, a_n$ in order:

$$a_1, \ldots, a_{j-1}, a_j, a_{j+1}, \ldots, a_{n-\ell-1}, a_{n-\ell}, a_{n-\ell+1}, \ldots, a_n,$$

whereas the second operation sequence $\overrightarrow{\mathsf{op}}^{(1)}$ returned by $\mathcal{A}_1$ contains the same n insertions, with only the order of the j'th and $(n-\ell)$'th insertions swapped:

$$a_1, \ldots, a_{j-1}, a_{n-\ell}, a_{j+1}, \ldots, a_{n-\ell-1}, a_j, a_{n-\ell+1}, \ldots, a_n.$$

The adversary $\mathcal{A}_1$ includes the complete list of a_1 up to a_n, along with the distinguished index j, in the $\textsc{st}$ which is passed to $\mathcal{A}_2$. As the last ℓ operations are identical (insertion of items $a_{n-\ell+1}$ up to a_n), $\mathcal{A}_1$ is ℓ-admissible.

The "guesser", $\mathcal{A}_2$, looks back in the last $(\ell+1)k$ entries in the access pattern history of persistent storage $\overrightarrow{\mathsf{acc}}$, and tries to opportunistically decrypt the data in each access entry using the keys from $\mathcal{D}.\mathsf{em}$ (and, recursively, any other decryption keys which are found from decrypting data in the access pattern history). Some of the data may be unrecoverable, but at least the $\ell+1$ items which were inserted in the last $\ell+1$ operations *must* be present in the decryptions, since their data must be recoverable using the erasable memory. Then the guesser simply looks to see whether a_j is present in the decryptions; if a_j is present then $\mathcal{A}_2$ returns 1, otherwise if a_j is not present then $\mathcal{A}_2$ returns 0.

In the experiment $\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 1)$, a_j must be among the decrypted values in the last $(\ell+1)k$ access entries, since a_j was inserted within the last $\ell+1$ operations and each operation is allowed to trigger at most k operations on the persistent storage. Therefore $\Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 1) = 1] = 1$.

In the experiment $\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 0)$, we know that each item $a_{n-\ell}, \ldots, a_n$ must be present in the decryptions, and there can be at most $(\ell+1)(k-1)$ other items in the decryptions. Since the index j was chosen randomly from among the first $n-\ell-1$ items in the list, the probability that a_j is among the decrypted items in this case is at most

$$\frac{(\ell+1)(k-1)}{n-\ell-1}.$$

From the restriction that $\ell \leq n/(4k)$, and $k \leq \sqrt{n}/2 \leq n/12$, we have

$$(\ell+1)(k-1) < (\ell+1)k = \ell k + k \leq \tfrac{n}{4} + \tfrac{n}{12} = \tfrac{n}{3}.$$

In addition, we have $n - \ell - 1 > n/2$, so the probability that a_j is among the decrypted items is at most $\frac{2}{3}$, and we have $\Pr[\mathsf{EXP}^{\mathsf{obl\text{-}hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1, 0) = 1] \leq 2/3$, and therefore $\mathbf{Adv}^{\mathsf{hi}}_{\mathcal{A}}(\mathcal{D}, \lambda, n) \geq 1/3$. According to the definition, this means that $\mathcal{D}$ does not provide history independence with leakage of ℓ operations.

C.2 Proof of Theorem 5

Our proofs on the distribution of block sizes in the ORAM and on the number of HIRB nodes depend on the following bound on the sum of geometric random variables. This is a standard type of result along the lines of Lemma 6 in [40].

Lemma 10. *Let $X = \sum_{1 \le i \le n} X_i$ be the sum of $n \ge 1$ independent random variables X_i, each stochastically dominated by a geometric distribution over $\{0, 1, 2, \ldots\}$ with expected value $\mathbb{E}[X_i] \le \mu$. Then there exists a constant $c_0 > 0$ whose value depends only on μ such that, for any $a \ge 2$ and $b \ge 0$, we have*

$$\Pr[X \ge (\mu + 1)(an + b)] < \exp(-c_0(an + b)).$$

Proof. By linearity of expectation, $\mathbb{E}[X] = \sum_{i \in [n]} \mathbb{E}[X_i] \le n\mu$.

Recall that a geometric random variable with expected value μ is equivalent to the number of independent Bernoulli trials, each with probability $p = 1/(\mu+1)$, before the first success. If $X \ge (\mu+1)(an+b)$, this is equivalent to having fewer than n successes over $k = (\mu + 1)(an + b)$ independent Bernoulli trials with probability p.

Using this formulation, we can apply the Hoeffding inequality to obtain

$$\Pr[X \ge k] = \Pr[\mathsf{Binomial}(k, p) \le n - 1] < \exp(-2\epsilon^2 k),$$

where ϵ is defined such that $n - 1 = (p - \epsilon)k$; namely

$$\epsilon = p - \tfrac{n-1}{k} = \tfrac{1}{\mu+1} - \tfrac{n-1}{k}.$$

We do some manipulation:

$$\begin{aligned} 2\epsilon^2 k &= \tfrac{2k}{(\mu+1)^2} \cdot \left(1 - \tfrac{(n-1)(\mu+1)}{k}\right)^2 \\ &= \tfrac{2(an+b)}{\mu+1} \cdot \left(1 - \tfrac{n-1}{an+b}\right)^2. \end{aligned}$$

Because $a \ge 2$ and $b \ge 0$, we have

$$\tfrac{n-1}{an+b} < \tfrac{n}{an} \le \tfrac{1}{2},$$

and so

$$\exp(-2\epsilon^2 k) < \exp\left(-\tfrac{1}{2(\mu+1)}(an + b)\right).$$

The stated result follows with the constant defined by

$$c_0 = \tfrac{1}{2(\mu+1)}. \tag{1}$$

■

Outline of proof of Theorem 5. We will mostly follow the proof of the small-stash-size theorem in Path ORAM [37]. The proof of the theorem consists of several steps.

1. We recall the definition of ∞-ORAM (ORAM with infinitely large buckets) and show that stash usage in an ∞-ORAM with post-processing is the same as that in the actual vORAM.

2. We rely on results from the most recent version of [38] to show that the stash usage after post-processing is greater than R if and only if there exists a subtree for which its usage in ∞-ORAM is more than its capacity.

3. We bound the total size of all blocks in any such subtree by combining two separate measure concentrations on the number of blocks in any such subtree, and the total size of any fixed number of variable-length blocks.

4. We complete the proof by connecting the measure concentrations to the actual stash size, in a similar way to [38].

Note that the first and third steps are those that differ most substantially from prior work, and where we must incorporate the unique properties of the vORAM.

Proof of Theorem 5. We now give the proof.

Step 1: ∞-ORAM. The ∞-ORAM is the same as our vORAM tree, except that each bucket has infinite size. In any writeback operation all blocks will go as far down along the path as their identifier allows.

After simulating a series of vORAM operations on the ∞-ORAM, we perform a greedy post-processing to restore the block size condition:

- Repeatedly select a bucket storing more than Z bytes. Remove a partial block from the bucket, and let b be the number of remaining bytes stored in the bucket.

- If $Z - b$ is greater than the size of metadata per partial block (identifier and length), then there is some room left in the bucket. In this case, split the removed block into two parts. Place the last $Z - b$ bytes into the current bucket and the remainder into the parent bucket. Otherwise, if there is insufficient room in the bucket, place the entire block into the parent bucket, or into the stash if the current bucket is the root.

By continuing this process until there are no remaining buckets with greater than Z bytes, we have returned to a normal vORAM with bucket size Z. Furthermore, there is an ordering of the accesses, with the same identifiers and block lengths, that would result in the same vORAM. Since the access order of the ∞-ORAM does not matter, this shows that the two models are equivalent after post-processing.

Observe that we are ignoring the metadata (block identifiers and length strings). This is acceptable, as the removal process in the actual vORAM always ensures that each partial block of a given block, except possibly for the first (highest in the vORAM tree), has size at least equal to the size of its metadata. In that way, at most half the vORAM is used for metadata storage, and so the metadata has only a constant factor effect on the overall performance.

Step 2: Overflowing subtrees. Consider the size of vORAM stash after any series of vORAM operations that result in a total of at most n blocks being stored. Similarly to [38, Lemma 2], the stash size at this point is equal to the total overflow from some subtree of the ∞-ORAM buckets that contains the root. If we write τ for that subtree, then we have

$$\begin{aligned}&|stash| > BR \quad \text{iff} \\ &\textstyle\sum_{\text{node } v\in\tau}(\text{size of } v \text{ in } \infty\text{-ORAM}) \geq Z|\tau| + BR.\end{aligned}$$

Step 3: Size of subtrees. We prove a bound on the total size of all blocks in any subtree τ in the ∞-ORAM in two steps. First we bound the *number* of blocks in the subtree, which can use the same analysis as the Path ORAM; then we bound the total *size* of a given number of variable-length blocks; and, finally, we combine these with a union bound argument.

To bound the total number of blocks that occur in τ, because the block sizes do not matter in the ∞-ORAM, we can simply recall from [38, Lemma 5] that, for any subtree τ, the probability that τ contains more than $5|\tau| + R/4$ blocks is at most

$$\tfrac{1}{4^{|\tau|}} \cdot (0.9332)^{|\tau|} \cdot (0.881)^{R}. \tag{2}$$

Next we consider the total size of $5|\tau| + R/4$ variable-length blocks. From the statement of the theorem, each block size is stochastically dominated by BX, where B is the expected block size and X is a geometric random variable with expected value $\mu = 1$. From Lemma 10, the total size of all $5|\tau| + R/4$ blocks exceeds $2(a(5|\tau| + R/4))B$ with probability at most

$$\exp\left(-c_0 a\left(5|\tau| + R/4\right)\right).$$

From (1), we can take $c_0 = 1/4$, and by setting $a = 2 > (4/5)\ln 4$, the probability that the total size of $5|\tau| + R/4$ blocks exceeds $(20|\tau| + R)B$ is at most $\exp\left(-\frac{5}{2}|\tau| - \frac{1}{8}R\right)$, which in turn is less than

$$\tfrac{1}{4^{|\tau|}} \cdot (0.329)^{|\tau|} \cdot (0.883)^R. \tag{3}$$

Finally, by the union bound, the probability that the total size of all blocks in τ exceeds $(20|\tau| + R)B$ is at most the sum of the probabilities in (2) and (3), which is less than

$$\tfrac{2}{4^{|\tau|}} \cdot (0.9332)^{|\tau|} \cdot (0.883)^R. \tag{4}$$

Step 4: Stash overflow probability. As in [38, Section 5.2], the number of subtrees of size i is less than 4^i, and therefore by another application of the union bound along with (4), the probability of *any* subtree τ having total block size greater than $(20|\tau| + R)B$ is at most

$$\begin{aligned}&\sum_{i\geq 1} 4^i \tfrac{2}{4^i} \cdot (0.9332)^i \cdot (0.883)^R \\ &< 28 \cdot (0.883)^R.\end{aligned}$$

C.3 Proof of Theorem 8

We now utilize Lemma 10 to prove the two lemmata on the distributions of the number and size of HIRB tree nodes.

Lemma 11. *Suppose a HIRB tree with n items has height $H \geq \log_\beta n$, and let X be the total number of nodes in the HIRB, which is a random variable over the choice of hash function in initializing the HIRB. Then for any $m \geq 1$, we have*

$$\Pr\left[X \geq H + 4n + m\right] < 0.883^m.$$

In other words, the number of HIRB nodes in storage at any given time is $O(n)$ with high probability. The proof is a fairly standard application of the Hoeffding inequality [18].

Proof. The HIRB has H nodes initially. Consider the n items $\mathsf{label}_1, \ldots, \mathsf{label}_n$ in the HIRB. Because the tree is uniquely represented, we can consider the number of nodes after inserting the items in any particular order.

When inserting an item with label_i into the HIRB, its height $h = \mathsf{chooseheight}(\mathsf{label}_i)$ is computed from the label hash, where $0 \leq h \leq H$, and then exactly h existing HIRB nodes are split when label_i is inserted, resulting in exactly h newly created nodes.

Therefore the total number of nodes in the HIRB after inserting all n items is exactly H plus the sum of the heights of all items in the HIRB, which from Assumption 6 is the sum of n iid geometric random variables, each with expected value $1/(\beta - 1)$. Call this sum Y.

We are interested in bounding the probability that Y exceeds $4n + m$. Writing $\mu = 1/(\beta - 1)$ for the expected value of each r.v., we have $\mu + 1 = \beta/(\beta - 1)$, which is at most 2 since $\beta \geq 1$. This means that $4n + m \geq (\mu + 1)(2n + m/2)$, and from Lemma 10,

$$\begin{aligned}
\Pr[X \geq H + 4n + m] &= \Pr[Y \geq 4n + m] \\
&\leq \Pr[Y \geq (\mu + 1)(2n + m/2)] \\
&< \exp(-c_0(2n + m/2)) \\
&< \exp(-c_0 m/2).
\end{aligned}$$

Because $\mu + 1 \leq 2$, $c_0 = 1/(2(\mu + 1)) \geq 1/4$. Numerical computation confirms that $\exp(-1/8) < 0.883$, which completes the proof. ■

Along with the bound above on the number of HIRB nodes, we also need a bound on the size of each node.

Lemma 12. *Suppose a HIRB tree with n items has height $H \geq \log_\beta n$, and let X, a random variable over the choice of hash function, be the size of an arbitrary node in the HIRB. Then for any $m \geq 1$, we have*

$$\Pr[X \geq m \cdot \mathsf{nodesize}_\beta] < 0.5^m.$$

The proof of this lemma works by first bounding the probability that the number of items in any node is at most $m\beta$ and applies the formula for node size, i.e.,

$$\begin{aligned}
&\mathsf{nodesize}_k = \\
&\quad (k+1)(2T + \gamma + 1) + k(|\mathsf{Hash}(\mathsf{label})| + |\mathsf{value}|).
\end{aligned} \tag{5}$$

Proof. We first show that the probability that any node's branching factor is more than $m\beta$ is at most 0.5^m. This first part requires a special case for the root node, and a general case for any other node. Then we show that any node with branching factor at most $m\beta$ has size less than $m \cdot \mathsf{nodesize}_\beta$.

First consider the items in the root node. These items all have height H, which according to Assumption 6 occurs for any given label with probability $1/\beta^H$. Therefore the number of items in the root node follows a binomial distribution with parameter $1/\beta^H$. It is a standard result (for example, Theorem C.2 in [9]) that a sample from such a distribution is at least k with probability at most

$$\binom{n}{k} \frac{1}{\beta^{Hk}} < \frac{n^k}{2^{k-1}\beta^{Hk}}.$$

From the assumption $H \geq \log_\beta n$, $n^k \leq \beta^{Hk}$, so the bound above becomes simply 2^{-k+1}. Setting $k = m\beta$, the probability that the root node has at least k items and hence branching factor greater than $m\beta$, is seen to be at most $2^{-m\beta+1}$, which is always at most 2^{-m} because $m \geq 1$ and $\beta \geq 2$..

Next consider any nonempty HIRB tree node at height ℓ, and consider a hypothetically infinite list of possible label hashes from the HIRB which have height at least ℓ and could be in this node. The actual number of items is determined by the number of those labels whose height is exactly equal to ℓ before we find one whose height is at least $\ell + 1$. From Assumption 6, and the memorylessness property of the geometric distribution, these label heights are independent Bernoulli trials, and each height equals ℓ with probability $(\beta - 1)/\beta$.

Therefore the size of each non-root node is a geometric random variable over $\{0, 1, \ldots\}$ with parameter $1/\beta$. The probability that the node contains at least $m\beta$ items, and therefore has banching factor greater than $m\beta$, is exactly

$$\left(\tfrac{\beta-1}{\beta}\right)^{m\beta} < \exp(-m) < 0.5^m.$$

Here we use the fact that $(1-\frac{1}{x})^{ax} < \exp(-a)$ for any $x \geq 1$ and any real a.

All that remains is to say that a node with branching factor $m\beta$ has size less than $m \cdot \mathsf{nodesize}_\beta$, which follows directly from $m \geq 1$ and the definition of $\mathsf{nodesize}_\beta$ in (5). ■

Finally, we prove the main theorems on the vORAM+HIRB performance and security.

Proof of Theorem 8. We step through and motivate the choices of parameters, one by one.

The expected branching factor β must be at least 2 for the HIRB to work, which means we must always have $H \leq \lg n$, and so $T = \lg(4n + \lg n + \gamma) \leq \lg(4n + H + \gamma)$. Then Lemma 11 guarantees that the number of HIRB nodes is less than $H + 4n + \gamma$ with probability at least $(0.883)^\gamma$. This means that T is an admissible height for the vORAM according to Theorem 5 with at least that probability.

The choice of β is such that $Z \geq 20 \cdot \mathsf{nodesize}_\beta$, using the inequality

$$H \leq \lg n < \lg(4n) < T.$$

Therefore, by Lemma 12, the size of blocks in the HIRB will be admissible for the vORAM according to Theorem 5.

This allows us to say from the choice of R and Theorem 5 that the probability of stash overflow is at most $28 \cdot (0.883)^\gamma$.

Choosing H as we do is required to actually apply Lemmas 11 and 12 above.

Finally, the probability of two label hashes in the HIRB colliding is at most $2^{-\gamma}$. The stated result follows from the union bound over the three failure probabilities.