Oblivious Secure Deletion with Bounded History Independence

Daniel S. Roche, Adam J. Aviv and Seung Geol Choi
United States Naval Academy
[roche,aviv,choi]@usna.edu

May 28, 2015

Abstract

We present a new secure cloud storage mechanism that combines three previously disjoint security properties: obliviousness, secure deletion, and history independence. The system maintains strong privacy guarantees against a cloud-observation attack, wherein an attacker learns all previous states of, and accesses to, the persistent cloud storage, as well as a catastrophic attack, where the decryption keys from erasable memory are also leaked. In the first scenario, the access pattern reveals nothing about the contents (obliviousness), and in both scenarios, no previously deleted data is recoverable (secure deletion), and the structure of the data leaks a bounded amount of information about previous states (history independence).

To achieve these goals we developed a new oblivious-RAM with variable-size storage blocks (vORAM) and a new history independent, randomized data structure based on B-trees (HIRB tree) stored within the vORAM. We prove that the vORAM+HIRB achieves obliviousness and secure deletion. We also show that any such system must inevitably leak a bounded amount of history information, and prove that our vORAM+HIRB construction matches this lower bound, up to logarithmic factors. Our system also provides better utilization of the ORAM buckets and reduces the amount of local storage requirements by $O(\log n)$ as compared to prior work for storing map data structures. Finally, we have implemented and measured the performance of our system using Amazon Web Services for a sample password-management application that maintains the privacy of login records, addition and removal of accounts, and password changes. The empirical performance is comparable to current state of the art in ORAM technologies and greatly outperforms naive approaches that provide the same security properties.

1 Introduction

Motivation. Cloud services provide clear advantages: virtually unlimited storage space; flexible allocation; backup and recovery; low cost; and, most importantly, high accessibility. However, the hidden cost of existing cloud services is that they often
provide minimal or opaque security guarantees where users have limited control over how and where their data is stored. Recent events, for instance the leaking of celebrity photos stored in the cloud [Isa14] and the trend of many government agencies, such as the CIA [Kon14], turning to third-party cloud storage providers such as Amazon, highlight the necessity of improved privacy and security settings for data stored in the cloud.

The scope of these challenges is not always apparent at first blush; the naive approach of just encrypting content locally and then uploading it to the cloud does not solve many practical security and privacy issues. Even if the database itself is encrypted with a secret key maintained locally away from the cloud, the cloud provider may not be trustworthy or may be compromised by an adversary. In such a situation, the adversary can conduct a *cloud observation attack* and observe each access to the encrypted database if not the contents of the database itself. Indeed, it was shown that the attacker may infer 80% of keyword search queries with a cloud observation attack on an encrypted email repository [IKK12]. To protect against such an attack, the cloud provider should provide *oblivious* access so that a passive observer cannot determine which data items are being accessed.

The situation worsens in the scenario of a *catastrophic attack* where the client’s device (and hence the secret key) is additionally compromised. In this case, the entire contents of the current database are inevitably leaked. However, as the user has little control over how data is maintained on the cloud, it could be the case that past instances of data persist, such as previously deleted encrypted documents. Following a catastrophic attack, the adversary might be able to decrypt content thought to be deleted, by combining the observed cloud access pattern or backup storage with the locally-stored decryption keys. To protect against such leakage, *secure deletion* is also required so that all past instances of the database are inaccessible.

Even that might not be enough. By inspecting the internal structure of the current database, the adversary may still be able to infer information about which fields were recently accessed or the likely prior existence of a record even if that record was previously securely deleted. For example, consider inserting 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. If an AVL tree is used, a different structure will result from each scenario, revealing information on the insertion order and previous deletions. Hence, we also need to provide *history independence* to ensure that the current structure of stored data will not leak information.

**Our work.** We analyze and defend against the two aforementioned attacks. A *cloud observation attack* occurs when the attacker can observe all the traffic at the cloud, including accesses to the persistent data segments as well as the content of the storage. In reality, this could be the cloud provider itself, an attacker who gains access to the cloud server, or a dishonest co-resident of a shared cloud server. We show that in this scenario the attacker learns nothing about currently or previously stored data other than the order of magnitude for the size of the data stored and the number operations that have occurred on the data (achieving obliviousness). A *catastrophic attack* occurs when an attacker observes both the previous cloud operations and has since coerced the decryption keys to reveal the current unencrypted state of the data store, perhaps due to theft, hacking, or a subpoena. In this scenario, we ensure that no previously deleted
data is recoverable by the attacker (achieving secure deletion) nor does the structure of the persistent storage fully reveal the previous states of the data (achieving history independence).

In the research community, the security of cloud storage has been the focus of numerous recent research efforts (e.g., [RRBC13, WNL+14, Gol09]) that individually can solve one but not all of the challenges described above. To the best of our knowledge, there is no previous work that considers all the properties simultaneously. In this paper we combine and advance upon these efforts using a richer set of security guarantees. In particular, we combined the security properties from obliviousness, secure deletion, and history-independence into a new, unified system for secure remote cloud storage that is well protected even against a powerful adversary.

To meet these goals, we have developed two new data structures. First, we have adapted path-based oblivious RAM (ORAM) [GO96, DMN11, GMOT12, KLO12, SSS12, SvDS+13, JSS14] and modified its structure to store variable-size data blocks. We call our modified ORAM vORAM (variable-length ORAM), and it has the same security properties as previous ORAM models with similar (and perhaps even improved) performance as compared to current state of the art ORAMs. Second, stored within the vORAM is a new data structure, called a HIRB tree (history independent, randomized B-tree), which has the property that the current state of the data structure does not reveal the order of accesses, insertions or removals. Together, the vORAM+HIRB construction achieves secure deletion and obliviousness, and we show that there is an inherent limitation in achieving history independence for any system in this model. Our system is nearly optimal with respect to a provable lower bound.

In order to empirically measure the performance of our construction, we developed a simple example application to store login credentials using Amazon Web Services as the cloud provider. First we analyzed the parameters of our vORAM bucket size and HIRB tree node size, to empirically determine the smallest constant overhead factor to achieve high performance with low likelihood of failure. Second, we measured the performance of our complete login credential storage system to demonstrate its practical utility, compared with a naive baseline approach that achieves the same security properties by transferring and re-encrypting the entire database on each access. These preliminary results demonstrate the practical value of our new construction, and the potential for further applications in secure cloud storage.

As the combination of obliviousness, secure deletion, and history independence has not been previously considered to the best of our knowledge, the current best alternative approach that provides the same level of protection is to re-encrypt and transfer the entire datastore on every access, which is why we applied it as a baseline in our experiments described above. Clearly, the naive approach is costly as each operation is \( O(n) \), and so the fact that local storage, communication bandwidth, and computational cost of each vORAM+HIRB operation is polylogarithmic represents an exponential improvement in performance. We demonstrate this fact empirically using our sample application. Furthermore, while there is an increase in the amount of cloud storage used for vORAM+HIRB as compared to the naive baseline, it is only a constant factor.

To summarize, the contributions of this paper are:

- New security definitions that combine the properties of secure deletion, history
independence, and obliviousness;
- 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 construction;
- The implementation and measurement of the vORAM+HIRB system for an account/password management application.

While we argue that the vORAM+HIRB construction is the first to effectively provide obliviousness, secure deletion, and history independence, there are also interesting takeaways when comparing the construction to systems that provide one or a few of the properties but not all. For example, compared with known data structures supporting secure deletion or history independence only, the additional security properties of our vORAM+HIRB incur an extra \( O(\log n) \) factor computational and communication cost, arising from the height of the ORAM storage tree. Compared with known data structures supporting obliviousness only, in particular [WNL+14], the addition of secure deletion and history independence in vORAM+HIRB incurs only a constant-factor increase in computational cost due to the increased number of cryptographic operations required on each access. However, the theoretical performance is otherwise the same or better compared to known results supporting the obliviousness property only.

Most interesting, the required amount of local, erasable storage in vORAM+HIRB is only \( O(\log n) \) as compared to \( O(\log^2 n) \) in [WNL+14]. Furthermore, the use of vORAM+HIRB removes the requirement that the ORAM bucket size be dictated by the size of the data; instead it can be set optimally according to the storage backend, allowing for larger bucket sizes and more efficient bandwidth utilization.

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. The goal of ORAM is to protect 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 [GO96], and since then, many works have focused on improving efficiency of ORAM in both the space, time, and communication cost complexities (for example [DMN11, GMOV12, KLO12, SSS12, SvDS+13, JSS14] 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 [ToF11], stacks and queues [MZ14], and graph algorithms [BSA13]. Recently, Wang et al. [WNL⁺14] 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. Wang et al. observed that when the data structure is a rooted tree, recursive positional map look-ups in ORAM can be avoided by each node storing its children’s positional tags.

Our ORAM construction, the variable block size ORAM, or vORAM, builds upon these prior constructions. In particular, we adopt the techniques from Wang et al. regarding positional mapping [WNL⁺14], and extend (non-recursive) Path ORAM [SvDS⁺13], to allow variable sized data items to be spread across multiple ORAM buckets. This design feature was developed to store dynamically sized B-tree nodes from the HIRB, but the applicability of the techniques of vORAM may be able to improve general ORAM performance.

**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. For example, an AVL tree does not provide history independence because tree rotations depend on the order of insertions.

Micciancio [Mic97] first considered history independence in the context of 2-3 trees, and the concept was further developed in [NT01, HHM⁺05, BP06]. The notion of strong history independence [NT01] 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 [HHM⁺05]. There have been uniquely-represented constructions for hash functions [BG07, NSW08] and variants of a B-tree (a B-treap [Gol09], and a B-skip-list [Gol10]).

We adopt the notion of unique representation for history independence when developing our history independent, randomized B-tree, or HIRB tree. Items in the HIRB are randomly assigned to levels in the B-Tree, which have variable sizes differing from standard B-trees. We show that, after initialization, the HIRB is uniquely represented and thus provides strong history independence.

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 traditional history independence and give a new, augmented notion of history independence for the cloud setting with a catastrophic attack.

**Secure deletion.** Secure deletion has been studied in many contexts [RBC13]. Here, we do not consider secure deletion mechanisms based on physical properties of the storage media, for example, properly overwriting previously stored data to ensure that it is unrecoverable. In the cloud, the user has little control over the storage media technology, and so other secure deletion techniques must be applied.

In particular, we build upon secure deletion techniques from the applied cryptography community. Typically, these systems assume that a user has a large amount of
persistent, non-erasable media and small amount of erasable media. Data stored in persistent media is encrypted with keys stored in the erasable memory, and thus deleting the keys from erasable memory will securely delete persistent media by rendering the data non-decryptable.

Boneh and Lipton [BL96] 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. [CFIJ99] showed a more efficient method for secure deletion using a tree structure applied in a setting of a large non-erasable persistent medium and a small erasable medium. Several works considered secure deletion mechanisms for a versioning file system [PBH+05], an inverted index in a write-once-read-many compliance storage [MWB08], and a B-tree (and generally a mangrove) [RRBC13]. There have been approaches to achieve secure deletion combined with revocation policies which depend on expiration time [Per05], a Boolean formula of policy attributes [TLLP12], and a general graph of policy attributes [CHHS13].

We apply the general idea of re-encryption to the vORAM buckets directly in order to achieve secure deletion, and although we do not directly use the same data structures as previous works, we do adopt the secure deletion terminology and properties. Particularly for the cloud setting, we consider the remote cloud storage to be persistent media because of the user’s lack of control over how the data is stored. Local storage, however, is considered to be erasable as the user can properly control the deletion process.

3 Preliminaries

We assume that readers are familiar with security notions of standard cryptographic primitives [KL07]. Let $\lambda$ denote the security parameter.

Types of storage. 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 that the size of erasable memory is small while the persistent storage has a much larger capacity. This mimics the cloud computing setting where cloud storage is large and persistent due to lack of user control, and local storage is more expensive but also controlled directly.

Data storage operations. We need to model many different kinds of data storage mechanisms: the erasable memory and persistent storage just described, existing ORAM constructions as well as our new vORAM, and any kind of mapping data structure. For simplicity, we assume all of these support these three operations:

- **insert**: $\text{data} \rightarrow \text{handle}$. Inserts the given $\text{data}$ and returns a $\text{handle}$ that can be used later to look up that item.
- **remove**: $\text{handle} \rightarrow \text{data}$. Removes the item identified by $\text{handle}$ and returns the original data $\text{data}$ as it was previously inserted.
• update\((\text{handle}, \text{callback}) \mapsto \text{handle}^+\). Locates the item specified by \text{handle} and passes it to the user-provided \text{callback} function. The return value from the \text{callback} replaces the original value, and a (possibly different) \text{handle}^+ for the new value is returned.

Because the user-provided \text{callback} can modify the data value or not, the update operation covers the typical operations of read-only lookup and modification, so hereafter, we will sometimes treat lookup as an operation, which is actually a mnemonic for update with the identity callback function. In fact update could be accomplished by a remove followed by an insert, but having it as a separate (single) operation is important as our privacy guarantees leak the number, but not the type, of operations performed.

Typically, the data being inserted would be a tuple \((\text{label}, \text{value})\), where the \text{handle} returned is simply the same label itself. In this situation, the \text{handle}^+ returned by update is also equal to the original \text{handle}. A call to insert\((\text{label}, \text{value})\) will replace any previously-stored value associated with that label. In the case of low-level storage (erasable or persistent), we assume each label will be an integer index, from 0 up to the size of the storage, and each value is a single byte.

The reason for our somewhat unusual definition of insert and update is to also accommodate non-recursive ORAMs (including our vORAM) that assign each \text{handle} as a random identifier which is changed on each operation.

An operation sequence on some data storage is a list of tuples \((\text{op}, \text{args})\), where \text{op} is insert, remove, or update, and \text{args} is a list of arguments each of which is \text{handle}, \text{data}, or \text{callback}. Here, \text{handle} can be a static value or a function of outputs of prior operations. An operation sequence represents the list of commands that are sent to be performed on the data store, but does not include their outputs. A complete historical record of a sequence of operations including both inputs and outputs is called an access pattern, and can be modeled as a series of tuples, where each tuple is \((\text{op}, \text{handle}, \text{data})\) if \text{op} is insert or remove, or \((\text{op}, \text{handle}, \text{data}, \text{handle}^+, \text{data}^+)\) if \text{op} is update.

Modeling a data structure. We define a data structure \(\mathcal{D}\) as a collection of data that supports initialization, insert, remove, and update, using both erasable memory and persistent storage. For a data structure \(\mathcal{D}\) stored in this model, let \(\mathcal{D}.\text{em}\) and \(\mathcal{D}.\text{ps}\) denote the contents of the erasable memory and persistent storage, respectively, and let \(\mathcal{D}.\text{data}\) denote the set of actual data items in \(\mathcal{D}\). For example, an encrypted graph structure may be stored in \(\mathcal{D}.\text{ps}\) while the decryption key resides in \(\mathcal{D}.\text{em}\). Therefore, it should be difficult to determine \(\mathcal{D}.\text{data}\) from access only to \(\mathcal{D}.\text{ps}\), but easy with access to both \(\mathcal{D}.\text{em}\) and \(\mathcal{D}.\text{ps}\).

For a sequence of operations \(\vec{\text{op}} = (\text{op}_1, \ldots, \text{op}_n)\), let \(\vec{\text{acc}} \leftarrow \mathcal{D}.\vec{\text{op}}()\) denote the access pattern to persistent storage resulting from executing the given operations on \(\mathcal{D}\). This access pattern will be a sequence of tuples \((\text{op}, \text{handle}, \text{data}, [\text{handle}^+, \text{data}^+])\), where \text{op} is insert, remove, or update; \text{handle} (resp., \text{handle}^+) is a physical address; and \text{data} (resp., \text{data}^+) is raw data inserted into or removed from persistent storage at that address. We note, however, that our vORAM does not actually require the use of the update operation on persistent storage; only the insert and remove operations are used. The access pattern to erasable memory is assumed to be hidden. Note that the length of \(\vec{\text{acc}}\) may be different than that of \(\vec{\text{op}}\), as typically a single command \text{op} will
result in some \( k \) operations on persistent storage. We note that the access pattern \( \text{acc} \) completely determines the state of persistent storage \( D.\text{ps} \).

### 3.1 Security Definitions

We first overview the requirement that our security definition aims to capture. The adversary is allowed to observe not only the contents of but also the access patterns over the persistent storage. Now, we consider two cases: (1) the adversary manages to observe just persistent storage, and (2) the adversary can observe persistent storage and has gained access to erasable memory. These cases match to the different attack models: cloud-observation attack and catastrophic attack, respectively.

**Obliviousness.** If the adversary does not have access to erasable memory, obliviousness requires that the adversary cannot obtain any information about actual operations performed on data structure \( D \) other than the number of operations (and thereby a rough estimate on the overall size of data items). This security notion is defined through an experiment \( \text{obl} \). In the experiment, the adversary chooses two sequences of operations on the data structure and then tries to make use of the access pattern to persistent storage in order to guess which sequence was chosen by the experiment. The data structure provides obliviousness if every polynomial-time adversary has only a negligible advantage.

**Definition 1.** For a data structure \( D \), consider the following experiment with adversary \( A = (A_1, A_2) \):

\[
\text{Experiment } \text{EXP}_A^{\text{obl}}(D, \lambda, b) =
\begin{align*}
\text{acc}_0 &\leftarrow D.\text{Init}(1^\lambda); \\
(\overrightarrow{\text{op}}(0), \overrightarrow{\text{op}}(1), \text{STATE}) &\leftarrow A_1(1^\lambda, \text{acc}_0); \\
\overrightarrow{\text{acc}} &\leftarrow D.\overrightarrow{\text{op}}(b); \\
\text{return } A_2(\text{STATE}, \overrightarrow{\text{acc}});
\end{align*}
\]

We call the adversary \( A \) admissible if \( A_1 \) always outputs two sequences with the same number of operations. We define the advantage of the adversary \( A \) in the experiment above as:

\[
\text{Adv}_A^{\text{obl}}(D, \lambda) = \left| \Pr[\text{EXP}_A^{\text{obl}}(D, \lambda, 0) = 1] - \Pr[\text{EXP}_A^{\text{obl}}(D, \lambda, 1) = 1] \right|.
\]

We say that the data structure \( D \) provides obliviousness if for any sufficiently large \( \lambda \) and any PPT admissible adversary \( A \), there is a negligible function \( \text{negl} \) such that \( \text{Adv}_A^{\text{obl}}(D, \lambda) \leq \text{negl}(\lambda) \).

**History independence.** Suppose the adversary manages to obtain erasable memory. In this case, the entire data collection in \( D \) is leaked; however, it is desirable that no additional information is leaked besides the current contents of \( D \) data. There exist several data structures that are history independent (see Section 2 and Section 5 for examples), meaning the current state of the data structure reveals nothing about the
sequence of operations that resulted in that state. However, these data structures assume that the underlying storage is fully in erasable memory. Our definition extends traditional history independence to the cloud setting with a catastrophic attack.

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 a limit on the allowable leakage of recent operations. One can interpret a history-independent data structure with leakage of $\ell$ operations as follows: Although the data structure may reveal some recent $\ell$ 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 security definition is defined through an experiment and is parameterized with a value $\ell$. In the experiment, the adversary chooses two sequences of operations (of the same length) on the data structure with the following restrictions: (i) the last $\ell$ operations must be identical; and (ii) the two sequences must result in the same data. The experiment then chooses one of the two sequences at random, executes the operations, and provides the adversary the sequence of resulting access pattern as well as the final state of erasable memory. The adversary guesses which sequence has been executed by the experiment, and the advantage of the adversary is defined as the probability that the guess is right. We say that the data structure provides history independence with leakage of $\ell$ operations, if every polynomial-time adversary has only negligible advantage.

Definition 2. For a data structure $D$, consider the following experiment with adversary $A = (A_1, A_2)$.

Experiment $\text{EXP}_{A}^{\text{hind}}(D, \lambda, n, b)$

- $acc_0 \leftarrow D.\text{Init}(1^\lambda)$;
- $(\overrightarrow{op}(0), \overrightarrow{op}(1), \text{STATE}) \leftarrow A_1(1^\lambda, n, acc_0)$;
- $\overrightarrow{acc} \leftarrow D.\overrightarrow{op}(b)$;
- return $A_2(\text{STATE}, \overrightarrow{acc}, D.\text{em})$;

We call the adversary $A$ $\ell$-admissible if $A_1$ always outputs sequences $\overrightarrow{op}(0)$ and $\overrightarrow{op}(1)$ which have the same number of operations and result in the same data with $n$ items, and the last $\ell$ operations of both are identical. We define the advantage of an adversary $A$ in the experiment above as:

$$\text{Adv}_{A}^{\text{hind}}(D, \lambda, n) = \left| \Pr_{\text{Exp}_{A}^{\text{hind}}(D, \lambda, n, 0)}[1] - \Pr_{\text{Exp}_{A}^{\text{hind}}(D, \lambda, n, 1)}[1] \right|.$$  

We say that the data structure $D$ provides history independence with leakage of $\ell$ operations if for any sufficiently large $\lambda$ and $n \in \text{poly}(\lambda)$, and any PPT $\ell$-admissible adversary $A$, there is a negligible function $\text{negl}$ such that $\text{Adv}_{A}^{\text{hind}}(D, \lambda, n) \leq \text{negl}(\lambda)$.

Observe that the definition of history independence is identical to that of secure deletion, with two important differences in the capabilities of the adversaries. The
“chooser” $A_1$ is further restricted in that the operation sequences it returns must share the last $\ell$ operations and result in the same size-$n$ data structure. On the other hand, the “guesser” $A_2$ is further enabled compared to experiment obl, in that $A_2$ gains access to the erasable memory as well as the access pattern.

**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 3.1.** Any oblivious RAM storage system that accesses $k$ bytes in persistent storage for every operation, where $k \in \omega(\sqrt{n})$, achieves at best history independence with leakage of $\Omega(n/k)$ operations in storing $n$ blocks.

The intuition behind the proof 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.

To motivate our HIRB tree construction, we consider briefly an AVL tree implemented within a standard ORAM, as has been done in prior work. Such a system cannot provide history independence with any amount of leakage less than the full operation sequence length. Using the fact that AVL tree shapes reveal information about past operations as illustrated in Section 1, the adversary can easily 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 lookup operations, as many as necessary, follow at the end. Under a 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. One of the motivations for our new vORAM+HIRB construction is to avoid this problem (see Sections 4 and 5).

**Secure deletion.** As the previous lower bound shows that complete history independence will not be possible in our setting, we consider a subset of history independence dealing only with the items which were inserted into the data structure and then deleted, prior to the catastrophic attack which reveals the contents of erasable memory to an adversary. In this case, the entire data collection $D.data$ is once again leaked; however, we require the system to provide strong security for the deleted data.

Note strict history independence with no leakage, which we have shown to be impossible, would imply secure deletion; the relaxed definition of history independence with leakage is complementary to the secure deletion definition below.

The security definition is defined through an experiment sdel. Before going over the experiment, we introduce a notation about sequences of operations. Let $\langle \delta, S \rangle$ denote a vector of operations with empty placeholders whose positions are specified by $S$, where $S = (s_1, s_2, \ldots, s_m)$ is a strictly monotonically increasing sequence. Now, we explain our experiment. In the experiment, the adversary chooses two data items $d_0$ and $d_1$ at random, and then chooses a sequence $\langle \delta, S \rangle$ of operations with placeholders
that contain neither \(d_0\) nor \(d_1\). We consider two resulting sequences \(\langle \overrightarrow{op}, S \rangle\) filling \(d_0\) or \(d_1\) in the empty placeholders: The operation to be in the first empty placeholder is \(\text{insert}(d_b) \mapsto \overrightarrow{handle}_0\), the intermediate ones \(\text{lookup}(\overrightarrow{handle}_i) \mapsto \overrightarrow{handle}_{i+1}\), and the last \(\text{remove}(\overrightarrow{handle}_k)\). We write this merging operation as \(\overrightarrow{op} \times_S d_b\) for \(b \in \{0, 1\}\). The experiment then chooses one of two merged sequences at random and executes the operations. We say that the data structure provides secure deletion if every polynomial-time adversary (even with access to erasable memory) makes a guess with negligible advantage.

**Definition 3.** For a data structure \(D\), consider the following experiment with adversary \(A = (A_1, A_2, A_3)\).

**Experiment** \(\text{EXP}_{\text{del}}(D, \lambda, b)\)

\[
\begin{align*}
\text{acc}_0 &\leftarrow D.\text{Init}(1^\lambda); \\
\langle \overrightarrow{op}, S \rangle &\leftarrow A_2(1^\lambda, \text{acc}_0, d_0, d_1); \\
\text{acc} &\leftarrow D.(\overrightarrow{op} \times_S d_b()); \\
\text{return } A_3(1^\lambda, \text{acc}_0, \text{acc}, D.\text{em}).
\end{align*}
\]

We call the adversary \(A\) admissible if for any \(b \in \{0, 1\}\), and for any handle output by \(D.\text{insert}(A_1(1^\lambda, b))\), it holds \(\Pr[D.\text{insert}(A_1(1^\lambda, b)) = \text{handle}] \leq \text{negl}(\lambda)\), that is, the set of handles forms a high-entropy distribution. We define the advantage of \(A\) as:

\[
\text{Adv}_{\text{del}}(D, \lambda) = \left| \Pr[\text{EXP}_{\text{del}}(D, \lambda, 0) = 1] - \Pr[\text{EXP}_{\text{del}}(D, \lambda, 1) = 1] \right|.
\]

We say that the data structure \(D\) provides secure deletion if for any sufficiently large \(\lambda\) and any PPT admissible adversary \(A\), there is a negligible function \(\text{negl}\) such that \(\text{Adv}_{\text{del}}(D, \lambda) \leq \text{negl}(\lambda)\). □

We note that \(A_3\) doesn’t receive any state information from \(A_1\) or \(A_2\), contrary to the other definitions. We could define a stronger security definition (that is, \(A_3\) is allowed to receive state information from \(A_1\) and \(A_2\), but we don’t know how to achieve this stronger security. Still, our definition captures the cloud scenario where the catastrophic adversary (modeled as \(A_3\)) doesn’t collude with the client (modeled as \(A_1\) and \(A_2\)).

## 4 ORAM with variable-size blocks

The design of variable-size ORAM (vORAM) is based on the non-recursive version of Path ORAM [SvDS$^+13$], but we are able to add more flexibility by allowing each ORAM bucket to contain as many 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 [CFIJ99, RRBC13].

**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 $2^{T+1} - 1$. $T$ will also control the total number of allowable blocks, which is $2^T$.
- The bucket size $Z$: Each bucket has $Z$ bytes, 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 $R$: 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 $R \cdot B$ bytes.
- Block collision parameter $\gamma$: If each block has an identifier $id$ chosen at random, then these identifiers will all be distinct at every step with probability $1 - \text{negl}(\gamma)$.

**Bucket structure.** Each bucket is split into two areas: header and data. See Figure 1 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 bytes, 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 different $l$ values 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 2). We further require the first bit of each identifier to be always 1 in 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 $\gamma$ 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 three primary operations insert, remove, and update, as defined in Section 3. For vORAM, the handle used in each case will be a vORAM $id$ that is assigned at random and can only be used once.

Each vORAM operation involves two phases:

1. 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.
Figure 2: 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.

2. 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 operations combine these in a single step, by first evicting the path from the initial $id$, retrieving the block from stash and passing it to the callback function, then assigning a new random $id^+$ for the block which is re-inserted into the stash, and finally calling writeback on the original $id$.

**Security properties.** For obliviousness, any insert, remove, or 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, while each insert evicts buckets along a bogus, randomly chosen path. Both insert and update return a fresh $id^+$ for the new block which has never been used (yet) for any evict or writeback call.

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. Therefore recovering any previously deleted data reduces to acquiring the old-root key, which was securely deleted from local, erasable memory.
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; hence the vORAM achieves history independence with leakage of $O(n \log n)$ operations. This is certainly not a strong security property, but considering the lower bound of $\Omega(n/k)$-history independence from Theorem 3.1, and where the bandwidth $k \in O(\log n)$ for the vORAM, it is only a factor at most $O(\log^2 n)$ from optimal.

The following theorems state these properties formally. Note that for Theorem 4.3, we also account for the security parameter $\lambda$ in the amount of leakage.

**Theorem 4.1.** The vORAM provides obliviousness.

**Theorem 4.2.** The vORAM provides secure deletion.

**Theorem 4.3.** The vORAM provides history independence with leakage of $O(n \log n + n\lambda)$ operations.

**Performance.** 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 4.4.** Consider a vORAM with $T$ levels, collision parameter $\gamma$, storing $n < 2^T$ 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

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

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 $40Bn$ bytes, and the bandwidth for each operation amounts to $40B \lg n$ bytes. However, this is a theoretical upper bound, and our experiments in Section 6 show a smaller constants suffice. Namely, setting $Z = 6B$ and $T = [\lg n - 1]$ stabilizes the stash, so that the actual storage requirement and bandwidth per operation are $6Bn$ and $12B \lg n$ bytes, respectively.

To avoid failure due to stash overflow or collisions in any of a series of $N$ operations, the client storage $R$ and collision parameter $\gamma$ should both grow slightly faster than $\lg N$, 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 (but without any arithmetic properties of pointers) because the vORAM uses random identifiers for storage blocks.
- The blocks and pointers in our data structure must form a directed graph that is arborescence, such that there exists at most one pointer to each block; otherwise when reading a block using a pointer, all other pointers be invalidated when the vORAM assigns a new random identifier to the read block.
- The memory access pattern for an operation (e.g., insert, remove, or update) must be bounded by a fixed 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 (label, 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 [HHM+05] for the desired security property.

In sum, we require a uniquely-represented, tree-based data structure with bounded height. A variety of uniquely represented (or strongly history independent) data structures have been proposed in the literature [NT01, Gol09], but we are not aware of any that satisfy all of 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 [Gol10]; 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 nodes, each with multiple (label, 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 3). 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 3: 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\(^1\) 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 anything about 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 \( \beta \), where \( \beta \) 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 \((\text{label}, \text{value})\) in the HIRB is determined by a series of biased coin flips initialized from the hash of the \( \text{label} \). The distribution of selected heights for insertions uniquely determines the structure of the HIRB tree because the process is deterministic (up to the choice of hash function at initialization), and thus the HIRB is uniquely-represented.

**Parameters and preliminaries.** Two parameters are fixed at initialization: the expected branching factor \( \beta \), and the height \( H \). In addition, throughout this section we will write \( n \) as the number of distinct labels stored in the HIRB tree at some point in time, and \( \gamma \) as a security parameter that affects the length of hash digests\(^3\).

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 4 where \( h_i \) indicates Hash\((\text{label}_i)\).

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 \( |\text{Hash}(\text{label})| = 2H \lg \beta + \gamma \),

\(^1\)The hash function is chosen at random from a family of hash functions at the initialization of the HIRB tree. In practice, we used a SHA1 hash initialized with a random string chosen when the HIRB tree is instantiated.

\(^2\)Note that this choice of heights is more or less the same as the randomly-chosen node heights in a skip list.

\(^3\)The parameter \( \gamma \) for HIRB and vORAM serves the same purpose in avoiding collisions in identifiers so for simplicity we assume they are the same.
and define $\text{nodesize}_k$ to be the size of a HIRB tree node with branching factor $k$, given as

$$\text{nodesize}_k = (2T + \gamma + 1) + k(2T + 2H + 2\gamma + |value|),$$

where we write $|value|$ as an upper bound on the size of the largest value stored in the HIRB. (Recall that the size of each vORAM identifier is $2T + \gamma + 1$ bits.) 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 $\text{nodesize}_k$.

As $\beta$ 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 4.4 in the previous section, and the HIRB node size defined above, one would choose $\beta$ according to the inequality $20\text{nodesize}_\beta \leq Z$, 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 $H \geq \log_{\beta} n$; 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 on $n$ is known a priori. If not, it will be necessary to periodically increase the height of the HIRB tree (and the number of levels $L$ in the underlying vORAM). This can be accomplished with an expected $O(\beta)$ number of vORAM operations as it would only involve splitting the root node of the HIRB. With a careful implementation, this would reveal only a rough estimate on $\lceil \log_{\beta} n \rceil$ in a cloud-observation attack, a so-called one-bit leakage.

**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.

Initially, an empty HIRB tree of height $H$ is created, as shown in Figure 5. Each node has a branching factor of 1 and contains only the single vORAM identifier of its child.

Performing an insert or remove operation on some label involves first computing the height of the label. The height is determined by sampling from a geometric distri-
Figure 6: HIRB insertion/deletion of $X = (\text{Hash}(\text{label}), \text{value})$: 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 $X$.

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 6.

An update operation to retrieve and/or modify the value associated to some label requires updating each HIRB node along the search path from the root to the node containing that label’s hash value.

In a HIRB tree with height $H$, each update operation requires updating at most $H + 1$ nodes from the vORAM, and each insert or remove operation requires accessing 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, as has been done in previous work [WNL+14], would be to read all $2H + 1$ HIRB nodes from the vORAM, store them in temporary memory, and then write back the entire path after any update. Each vORAM read requires a temporary stash storage of $O(\log n)$, and so this process would require $O(\log^2 n)$ temporary storage in total. 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, reducing the total temporary memory requirement to only $O(\log n)$.

Unfortunately, facilitating this extra efficiency requires considerable care in the im-
plementation 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. We note that this approach and implementation detail enables a significant improvement in the local storage (i.e. erasable memory) requirement for mapped data structures in ORAMs as compared to [WNL+14]. The full details of this approach and the HIRB operations can be found in Appendix D.

HIRB tree properties. Our analysis of the HIRB tree follows the same general approach as [Gol10] for the B-SkipList. First we need to understand the distribution of items among each level in the HIRB tree. We assume a subroutine chooseheight(label) computes a hash of the label, then uses this to seed a pseudo-random number generator to sample from a truncated geometric distribution with maximum value $H$ and probability $(\beta - 1)/\beta$.

Assumption 5.1. If $\text{label}_1, \ldots, \text{label}_n$ are any $n$ labels stored in a HIRB, then the heights
\[
\text{chooseheight}(\text{label}_1), \ldots, \text{chooseheight}(\text{label}_n)
\]
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 choice of the hash function upon creation of the HIRB.

This assumption will hold true if the hash function is chosen at random from an $n$-independent family upon HIRB initialization. For the security property proofs, we need the stronger condition anyway that the hash function is a random oracle, and in practice we used a cryptographic hash function.\(^4\)

We can now state the most important properties of the HIRB tree data structure.

Theorem 5.2. The HIRB tree is a dictionary data structure that associates arbitrary labels to values. 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.

Proof. 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. Assume the hash function is chosen from a 2-universal family, which is less than what Assumption 5.1 requires. 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 insert, remove, and update, and is crucial not only for the performance of the HIRB but also for the obliviousness property.

\(^4\)Specifically, in our experiments, we used SHA1 with digest size greater than $2H \lg \beta + 40$, initialized with a random salt.
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 D.

**Theorem 5.3.** Suppose a HIRB tree with \( n \) items and height \( H \) is stored within a vORAM with \( T \) levels, bucket size \( Z \), and stash size \( R \). Given choices for \( Z \) and \( \gamma > 0 \), set the parameters as follows:

\[
T \geq \log(4n + \log n + \gamma)
\]
\[
\beta \leq \frac{Z - (2T + \gamma + 1)}{20(4T + 2\gamma + |\text{value}|)}
\]
\[
R \geq \gamma \cdot \text{nodesize}_\beta
\]
\[
H \geq \log_\beta n
\]

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

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

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

**Theorem 5.4.** 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 in a straightforward way from the previous results on the vORAM and the HIRB. Note that the HIRB itself provides history independence with no leakage, but when combined with the vORAM may leak information about recent operations. The factor \( O(\log n) \) difference from the amount of leakage from vORAM in Theorem 4.3 arises because each HIRB operation entails \( O(\log n) \) vORAM operations.

### 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, that is, the ratio \( Z/B \). Second, we made a complete implementation of the vORAM+HIRB and used it as the core for a sample application to manage password/account-logon using a secure shadowfile stored in the cloud using Amazon Web Services. The complete source code of our implementation is available upon request.
The prototype password application for vORAM+HIRB may seem somewhat unusual — storing shadowfiles in the cloud is not typically done — but it has many interesting and practical security properties that make it useful as a sample demonstration of our construction. It would be important to hide which user is currently logging on, or when accounts are being created, modified, or deleted (obliviousness); to securely remove accounts and all evidence of their prior existence (secure deletion); and to limit a posteriori auditing of account activity (frequency of logons, for instance) if there ever were a compromise (history independence). The cloud model is also not a prerequisite for the password application, and the same properties would exist on a local, single-machine based application where the hard disk is considered persistent (or hard to erase) and RAM is considered erasable. Such a local implementation would provide protections against a powerful adversary wishing to conduct an extensive audit as might occur after a search and seizure.

The security properties of the password application for hiding data accesses, securely deleting data, and limiting auditing of access, do not change if the application instead were to store photos, music, or documents in the cloud. In fact, one of the strengths of our vORAM is the ability to flexibly adapt to differing data block sizes while preserving obliviousness. As we have determined in practice the bottleneck is network transfer speeds, our application would actually scale sub-linearly with increasing data block sizes, as increasing the size of each bucket by a factor of $k$ would also allow a decrease in the HIRB tree height by a factor of $O(\log k)$. This indicates that other applications beyond an account credentials database would scale well in the cloud setting.

### 6.1 Block-to-bucket ratio in vORAM

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 $B = \text{nodesize}_β$ when storing HIRB nodes within 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 [SvDS+13].

We performed a similar experimental analysis of the ratio $Z/B$ for the vORAM. Our best theoretical ratio from Theorem 4.4 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 overflow.** To analyze the likelihood of stash overflow 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 7.

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 $100 \lg n$.

**Bucket utilization.** Stash overflow is the most important failure point 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 8 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, $2^{15}$ elements were inserted, and the vORAM height varied between 14, 15, and 16. The graph shows that when the height is 15 or higher and $Z/B$ is 6 or higher, utilization stabilizes throughout all the interior levels, with a small spike only at the leaf level.

To be more specific, 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

Figure 7: Maximum stash size, scaled by $\lg n$, observed across 50 simulations of a vORAM for various $Z/B$ values.
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). The stabilization at $Z/B = 6$ continues through higher ratio values, and we observed stabilization all the way out to $Z/B = 13$ (not depicted in the figure).

In addition, our data shows that decreasing the number of levels from $\log n$ to $\log n - 1$ (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 $\log n - 1$, 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 [SvDS’13].

6.2 Sample vORAM+HIRB Application

Finally, we fully implemented our vORAM+HIRB map data structure using Python3 and Amazon Web Services as the cloud service provider. The encryption routine was AES256 for vORAM, and we 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. In our experiments, we found that the network transfer rates as well as TCP and SSL connection setup/teardown and not the vORAM encryption routines dominate performance,
and so we made a number of improvements and optimizations to the vORAM access routines to compensate.

In particular, the bucket transfers along a single vORAM path were performed in parallel over simultaneous connections in both the evict and writeback methods. A local buffer of $2^T$ ORAM buckets — which is proportional to the required size of stash — was used to facilitate asynchronous path reading and writing by our threads. 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. Our experiments used up to $T$ threads in parallel to fetch and send ORAM block files, and each maintained a persistent sftp connection. We did not include the cost of the $\approx 2$ second setup/teardown time for these SSL connections in our results as these were a one-time cost incurred at initialization. Many of these (and more sophisticated) techniques have been used in previous work to achieve similar performance gains [LPM+13, YRF+14].

**Sample application.** To test the performance of our implementation, we designed a sample application for password/account management. The application maintains a shadowfile of usernames, hashed passwords, and salts on the cloud server. The client application, running locally, will access that data to authenticate users. The shadowfile is stored in the vORAM+HIRB with usernames as labels and hashed passwords and salts as values. When a user logs in, an access is made from the client to the vORAM+HIRB on the server to retrieve the hash and salt and check that the user provided the correct password for a valid username.

The security properties of the password/account application are quite interesting and practical for users. The application would enable a user to access/logon-to their account obliviously; an observer monitoring the password file cannot know which user just accessed their account. Furthermore, a user can update their password without observation and their previous password value (hash+salt) is securely deleted, or delete their account entirely without leaving any evidence of its existence. Even a user who makes no changes to their shadowfile entry benefits because the history independence properties ensure privacy for the user’s account access history after a certain window of time. Further, the application model is not limited to the cloud setting (although that is how we evaluated it) and can easily be adapted for local machines where the hard disk is persistent storage and the RAM is erasable.

We also emphasize again that this application could be easily scaled to any number of other types of information being stored, such as multimedia files or documents. A strength of our vORAM+ HIRB is that the size of blocks can easily adapt to the size of underlying data items, and the performance gains should be similar as the data scales larger.

**Performance and measurement.** To test our application, we generated a shadowfile of approximately 300,000 real usernames and passwords culled from the php-BB password leak and the gmail email address leak [Lys12, Bow11]. We then hashed and salted all the passwords using MD5 and 16 byte salt. The resulting shadowfile, un-encrypted, constituted 22MB. In our experiments, we used some subset of the shadowfile for comparisons.
We compared the performance of our application to a naive baseline application that provides the same security properties. The baseline application simply uploads an encrypted file to the cloud, downloads the file on each access, and re-encrypts the file with a new key before uploading it back to the cloud. Obliviousness and secure deletion are achieved through the re-encryption, and history independence is achieved as the entries in the shadowfile are stored in a consistent order based on the cryptographic hash of the usernames. We chose this baseline comparison because, to the best of our knowledge, there does not exist any current method that provides obliviousness, secure deletion, and history independence in a non-trivial way, other than our proposed vORAM+HIRB construction. This fair, apples-to-apples comparison best demonstrates the performance improvements of our system compared to an alternative that provides the same level of security. Comparison to other systems that provide some, but not all, of our security properties could also provide some additional insight, but we leave this for future work. As noted previously in our analytic results, even in situations where not all properties are required, such as no history independence, the new vORAM+HIRB construction may provide performance improvements as well as stronger privacy guarantees.

A comparison of the baseline solution with vORAM+HIRB clearly shows that the baseline solution is quite costly with respect to the communication bandwidth. The costs for the baseline scale linearly with the total size of the shadowfile; each access requires the download, decryption, re-encryption, and upload of the entire file. There are also additional network costs associated with transferring large files (e.g., windowing of the TCP connection). Conversely, the vORAM+HIRB solution communication cost is logarithmic with the size of the shadowfile as only the paths in the vORAM that are associated with the HIRB tree blocks must be transferred. However, the vORAM+HIRB has disadvantages based on the need to fetch many small files, and based on the fact that the total persistent size of vORAM is a considerable (but constant) factor larger than the original shadowfile by itself.

To measure this fully, we implemented the baseline solution and tested it alongside the vORAM+HIRB for shadow files ranging in the number of entries between $2^8$ and $2^{18}$. The results of the latency in access times are presented in Figure 9, where we graph the median of 100 accesses of the sample application, presented in log-log scale. While the latency of an access for vORAM+HIRB is less than a second for all runs, the naive baseline solution incurs significant latency increases as the shadowfile size increases. In the largest run, it takes more than 100 seconds to perform an access as compared to less than 1 second for vORAM+HIRB. The advantages of vORAM+HIRB are clear.

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. If an attacker were to compromise remote, persistent storage in the cloud (a cloud observation attack), nothing is revealed other than the number
of operations, and if an attacker compromised both remote, persistent storage and local, erasable memory (a catastrophic attack), then all previously deleted data is unrecoverable and only a limited history of prior operations is leaked.

The theoretical performance of our vORAM+HIRB construction is competitive to existing systems which provide fewer security properties, even reducing by a factor of $O(\log n)$ the local storage requirement in some cases. We implemented an application of vORAM+HIRB for password/account management and measured its performance, which far outstripped that of a naive baseline application that achieves the same security properties. In our empirical analysis we also identified a number of key settings for the vORAM+HIRB; namely, that the ratio of bucket size to expected block size can be set to 6 and that only $\log n - 1$ levels are required to store $n$ items.

There is significant potential for future work in these directions. Besides other applications using vORAM+HIRB, one could also consider data structures that support a richer set of operations, such as range queries, while preserving obliviousness, secure deletion, and history independence. Finally, it is important to note that the vORAM construction in itself may provide novel and exciting new analytic results for ORAMs generally by not requiring fixed bucket sizes. Since buckets can be described as fractions or ratios instead of fixed integer multiples, there is a potential to improve the overall utilization and communication cost compared to existing ORAM models, even when storing only fixed-size blocks.

Figure 9: Median of 100 access times for different number of entries
References

[BG07] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In 48th FOCS, pages 272–282. IEEE Computer Society Press, October 2007.

[BL96] Dan Boneh and Richard J. Lipton. A revocable backup system. In Proceedings of the 6th USENIX Security Symposium, 1996.

[Bow11] Ron Bowes. Skull Security, 2011. https://wiki.skullsecurity.org/Passwords.

[BP06] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. Inf. Comput., 204(2):291–337, 2006.

[BSA13] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, ASIACCS 13, pages 207–218. ACM Press, May 2013.

[CFIJ99] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In STACS, pages 500–509, 1999.

[CHHS13] Christian Cachin, Kristiyan Haralambiev, Hsu-Chun Hsiao, and Alessandro Sorniotti. Policy-based secure deletion. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, ACM CCS 13, pages 259–270. ACM Press, November 2013.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, second edition, September 2001.

[DMN11] 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.

[GMOT12] 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.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. J. ACM, 43(3):431–473, 1996.

[Gol09] Daniel Golovin. B-treaps: A uniquely represented alternative to B-trees. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, ICALP 2009, Part I, volume 5555 of LNCS, pages 487–499. Springer, July 2009.

[Gol10] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. CoRR, abs/1005.0662, 2010.
[HHM+05] 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.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioğlu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, February 2012.

[Isa14] 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.

[JSS14] 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.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[KLO12] 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.

[Kon14] 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-clas-deal-with-amazon/374632/.

[LPM+13] 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.

[Lys12] T. Alexander Lystad. The Password Project, 2012. http://thepasswordproject.com/leaked_password_lists_and_dictionaries.

[Mic97] Daniele Micciancio. Oblivious data structures: Applications to cryptography. In 29th *ACM STOC*, pages 456–464. ACM Press, May 1997.

[MWB08] 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.

[MZ14] 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.
[NSW08] 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.

[NT01] Moni Naor and Vanessa Teague. Anti-presistence: History independent data structures. In *33rd ACM STOC*, pages 492–501. ACM Press, July 2001.

[PBH+05] 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.

[Per05] Radia Perlman. The ephemizer: Making data disappear. Technical report, Mountain View, CA, USA, 2005.

[RBC13] 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.

[RRBC13] Joel Reardon, Hubert Ritzdorf, David A. Basin, and Srdjan Capkun. Secure data deletion from persistent media. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 271–284. ACM Press, November 2013.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *NDSS*, 2012.

[SvDS+13] 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 Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 299–310. ACM Press, November 2013.

[SvDS+14] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher and Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *CoRR*, abs/1202.5150v3, 2014.

[TLLP12] Yang Tang, Patrick P. C. Lee, John C. S. Lui, and Radia Perlman. Secure overlay cloud storage with access control and assured deletion. *IEEE Trans. Dependable Sec. Comput.*, 9(6):903–916, 2012.

[Tof11] 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.
[WNL+14] 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.

[YRF+14] 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.
A Probability Distribution Bounds

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 [WNL+14].

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

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

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

Recall that a geometric random variable with expected value \( \mu \) is equivalent to the number of independent Bernoulli trials, each with probability \( p = 1/(\mu + 1) \), before the first success. If \( X \geq (\mu + 1)(an + b) \), 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

\[
\Pr[X \geq k] = \Pr[\text{Binomial}(k, p) \leq n - 1] < \exp(-2\epsilon^2k),
\]

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

\[
\epsilon = p - \frac{n-1}{k} = \frac{1}{\mu+1} - \frac{n-1}{k}.
\]

We do some manipulation:

\[
2\epsilon^2k = \frac{2k}{(\mu+1)^2} \cdot \left(1 - \frac{(n-1)(\mu+1)}{k}\right)^2
\]

\[
= \frac{2(\mu+1)}{\mu+1} \cdot \left(1 - \frac{n-1}{an+b}\right)^2.
\]

Because \( a \geq 2 \) and \( b \geq 0 \), we have

\[
\frac{n-1}{an+b} < \frac{n}{an} \leq \frac{1}{2},
\]

and so

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

The stated result follows with the constant defined by

\[
c_0 = \frac{1}{2(\mu+1)}. \tag{1}
\]
B Lower Bound on History Independence

B.1 Proof of Theorem 3.1

We prove the lower bound on history independence.

**Theorem 3.1.** Any oblivious RAM storage system that accesses $k$ bytes in persistent storage for every operation, where $k \in o(\sqrt{n})$, achieves at best history independence with leakage of $\Omega(n/k)$ operations in storing $n$ blocks.

**Proof.** 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 $\ell$ 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 $\text{op}^{(0)}$ returned by $\mathcal{A}_1$ consists of $n$ insertion operations for $a_1, a_2, \ldots, a_n$ 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 $\text{op}^{(1)}$ returned by $\mathcal{A}_1$ contains the same $n$ insertions, with only the order of the $j$’th and $(n-\ell)$’th insertions swapped:

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

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

The “guesser”, $\mathcal{A}_2$, looks back in the last $(\ell + 1)k$ entries in the access pattern history of persistent storage $\mathcal{A}^{acce}$, and tries to opportunistically decrypt the data in each access entry using the keys from $\mathcal{D}.em$ (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 $\text{EXP}^{\text{hind}}_{\mathcal{A}}(\mathcal{D}, \lambda, n, 1)$, $a_j$ 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 \( \Pr[\text{EXP}_{A}^{\text{hind}}(D, \lambda, n, 1) = 1] = 1 \).

In the experiment \( \text{EXP}_{A}^{\text{hind}}(D, \lambda, n, 0) \), 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

\[
(\ell + 1)(k-1) < (\ell + 1)k = \ell k + k \leq \frac{n}{4} + \frac{n}{12} = \frac{2n}{3}.
\]

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

C vORAM Proofs and Operation Details

The full detail of the vORAM helper functions is provided in Figure 10, and the three main operations are shown in Figure 11.

C.1 Proof of Theorem 4.1

Theorem 4.1. The vORAM provides obliviousness.

Proof. We prove the theorem using the standard hybrid argument. We consider the following hybrid experiments:

Hybrid 0: It’s \( \text{EXP}_{A}^{\text{obl}}(D, \lambda, 0) \).

Hybrid 1: It’s the same as \( \text{EXP}_{A}^{\text{obl}}(D, \lambda, 0) \), except that each \( \text{Enc}_{\text{key}}(\text{bucket}) \) in writeback is replaced with \( \text{Enc}_{\text{key}}(0^z) \), where \( z \) is the length of each bucket.

Hybrid 2: It’s the same as \( \text{EXP}_{A}^{\text{obl}}(D, \lambda, 1) \), except that each \( \text{Enc}_{\text{key}}(\text{bucket}) \) in writeback is replaced with \( \text{Enc}_{\text{key}}(0^z) \).

Hybrid 3: It’s \( \text{EXP}_{A}^{\text{obl}}(D, \lambda, 1) \).

We first show that Hybrids 0 and 1 are indistinguishable. Note each encryption key in our construction is used only once and the root-key is never observable by the adversary. Therefore, due to the security of symmetric encryption \( \text{Enc} \), the two hybrids are indistinguishable.

We next show that Hybrids 1 and 2 are identically distributed. This is mainly because the identifier of each block is used only once to retrieve that item, and then immediately discarded. For remove, the identifier is immediately discarded, of course.
location(id, t)
1: return the location of the node at level \( t \) along the path from the root to the leaf node identified by \( id \). This is simply the index indicated by the \((t + 1)\) most significant bits of \( id \).

idgen()
1: Choose \( r \leftarrow \{0, 1\}^{2T + \gamma} \).
2: return \( 1||r \).

evict(id)
1: \( key \leftarrow rootkey \) \quad \triangleright rootkey : encryption key for the root bucket
2: \( B \leftarrow \) empty list
3: for \( t = 0, 1, \ldots, T \) do
4: Remove bucket at location \((id, t)\) from persistent storage and decrypt it with \( key \)
5: Append all partial blocks in the bucket to the end of \( B \)
6: \( key \leftarrow \) child key from bucket according to location \((id, t + 1)\)
7: end for
8: for each partial block \((id^*, \ell, blk)\) in \( B \) do
9: if \((id^*, \ell, blk)\) is in stash already then replace with \((id^*, \ell + \ell, blk0 blk1)\) \quad \triangleright merge two partial blocks
10: else Add \((id^*, \ell, blk)\) to stash
11: end for

writeback(id)
1: \( key \leftarrow \) nil
2: for \( t = T, T - 1, \ldots, 0 \) do
3: \( W \leftarrow \{(id^*, \ell, blk) \in stash : location(id^*, t) = location(id, t)\} \quad \triangleright \) partial blocks storable in the bucket
4: create empty bucket with new child key \( key \) (other child key remains the same)
5: while \( W \) is not empty and bucket is not full do
6: \((id^*, \ell, blk)\) \leftarrow arbitrary element from \( W \)
7: \((id^*, \ell_1, blk_1)\) \leftarrow largest partial block that will fit in the bucket, where \( blk = blk_0 blk_1 \) and \( |blk_1| = \ell_1 \).
8: Add \((id^*, \ell_1, blk_1)\) to the bucket
9: if \( \ell_1 = \ell \) then Remove \((id^*, \ell, blk)\) from \( W \) and from stash
10: else replace \((id^*, \ell, blk)\) in stash with \((id^*, \ell - \ell_1, blk_0)\). \quad \triangleright split a partial block
11: end while
12: \( key \leftarrow \{0, 1\}^{\lambda} \) chosen uniformly at random
13: Insert Enc_{key}(bucket) at location\((id, t)\) in persistent storage.
14: end for
15: rootkey \leftarrow key

Figure 10: vORAM helper functions
**insert**(blk)

1:  \( id_0 \leftarrow \text{idgen}() \)
2:  \( \text{evict}(id_0) \)
3:  \( id^+ \leftarrow \text{idgen}() \)
4:  insert \((id^+, \|blk|, blk)\) into stash
5:  \( \text{writeback}(id_0) \)
6:  **return** \( id^+ \)

**remove**(id)

1:  \( \text{evict}(id) \)
2:  remove \((id, \ell, blk)\) from stash
3:  \( \text{writeback}(id) \)
4:  **return** \( blk \)

**update**(id, callback)

1:  \( \text{evict}(id) \)
2:  remove \((id, \ell, blk)\) from stash
3:  \( id^+ \leftarrow \text{idgen}() \)
4:  \( blk^+ \leftarrow \text{callback}(blk) \)
5:  insert \((id^+, \|blk^+|, blk^+)\) into stash
6:  \( \text{writeback}(id) \)
7:  **return** \( id^+ \)

Figure 11: vORAM operations
For insert, a bogus, fresh id\textsubscript{0} is used for evict and writeback, and another fresh id\textsuperscript{+} becomes the identifier of the inserted block. For update, the actual identifier id of the block evict and writeback, but a fresh id\textsuperscript{+} becomes the new identifier of the (updated) block. Therefore, in every operation, the identifier is uniformly distributed in both Hybrids.

The indistinguishability between Hybrids 2 and 3 holds, symmetrically to the case of Hybrids 0 and 1.

### C.2 Proof of Theorem 4.2

**Theorem 4.2.** The vORAM provides secure deletion.

**Proof.** We prove the theorem using the standard hybrid argument. We consider the following hybrid experiments:

**Hybrid 0:** It’s EXP\textsubscript{A}(D, λ, 0).

**Hybrid 1:** It’s the same as EXP\textsubscript{A}(D, λ, 0) except the following: The most recent encryption \text{Enc\textsubscript{key}}(bucket) in each bucket stays the same, but all the older encryptions are replaced with \text{Enc\textsubscript{key}}(0\textsuperscript{z}), where z is the length of each bucket.

**Hybrid 2:** It’s the same as EXP\textsubscript{A}(D, λ, 1) except the following: The most recent encryption \text{Enc\textsubscript{key}}(bucket) in each bucket stays the same, but all the older encryptions are replaced with \text{Enc\textsubscript{key}}(0\textsuperscript{z}).

**Hybrid 3:** It’s EXP\textsubscript{A}(D, λ, 1).

We first show that Hybrids 0 and 1 are indistinguishable. Note each encryption key in our construction is used only once, and the old root-key is never observable by the adversary since it had been in the erasable memory and was securely deleted. Therefore the attacker is only able to determine the most recent encryption of each bucket, upon learning the current root key in D.em. Therefore, due to the security of the symmetric encryption scheme \text{Enc}, the two hybrids are indistinguishable.

We next show that Hybrids 1 and 2 are identically distributed. As discussed in the proof of Theorem 4.1, in every operation, the identifier is uniformly distributed in both Hybrids. Moreover, all the most recent buckets contain identical items in both hybrids, and all the old buckets are identically distributed.

The indistinguishability between Hybrids 2 and 3 holds, symmetrically to the case of Hybrids 0 and 1.

### C.3 Proof of Theorem 4.3

**Theorem 4.3.** The vORAM provides history independence with leakage of \(O(n \log n + n\lambda)\) operations.
Proof. Let $\ell = \lceil n \ln n + n\lambda \rceil$ be the number of recent operations that are leaked. The operations before the $\ell$ operations are said to be “aged”. We prove the theorem using the standard hybrid argument. We consider the following hybrid experiments:

Hybrid 0: It’s $\text{EXP}^{\text{hind}}_A(D, \lambda, n, 0)$.

Hybrid 1: It’s the same as $\text{EXP}^{\text{hind}}_A(D, \lambda, n, 0)$ except the following: The most recent encryption $\text{Enc}_{\text{key}}(\text{bucket})$ in each bucket stays the same, but all the older encryptions is replaced with $\text{Enc}_{\text{key}}(0^z)$, where $z$ is the length of each bucket.

Hybrid 2: It’s the same as $\text{EXP}^{\text{hind}}_A(D, \lambda, n, 1)$ except the following: The most recent encryption $\text{Enc}_{\text{key}}(\text{bucket})$ in each bucket stays the same, but all the older encryptions is replaced with $\text{Enc}_{\text{key}}(0^z)$.

Hybrid 3: It’s $\text{EXP}^{\text{hind}}_A(D, \lambda, n, 1)$.

As shown in the proof of Theorem 4.2, Hybrids 0 and 1 are indistinguishable since data items are re-encrypted with a refreshed key in each writeback. Indistinguishability between Hybrids 2 and 3 holds symmetrically. From now on, we show indistinguishability between Hybrids 1 and 2.

Note that in Hybrids 1 and 2, old bucket encryptions are purely random strings. We first show that with overwhelming probability, all aged operations leave only old bucket encryptions; then, the adversary will see just random eviction path ids and encryptions of $0^z$ for those aged operations in the access pattern, even with knowledge of erasable memory. This event happens when each of $n$ randomly-chosen vORAM tree paths is repeated at least once in $\ell$ recent operations. This probability bound is essentially a coupon collector’s problem. We throw balls into $n$ bins at random, and seek an upper bound on the tail probability that any of the bins remain empty after $\ell \geq n \ln n + n\lambda$ balls are thrown. Since each ball goes into a given bin with probability exactly $\frac{1}{n}$, the probability that a single given bin remains empty after $\ell$ throws is at most

$$\left(1 - \frac{1}{n}\right)^{n \ln n + n\lambda} \leq e^{-\ln n - \lambda} = \frac{\exp(-\lambda)}{n}.$$ 

Here we used the fact that $1 - x \leq e^{-x}$ for any real $x$. Applying the union bound over all $n$ bins, we see the probability that any of them remains empty after $\ell$ throws is at most $\exp(-\lambda)$, as required.

The only concern left is whether the most recent bucket contents caused by the recent operations contain information about the past operation history. We argue that those contents reveal nothing about the order of aged operations. If an aged operation $\text{op}_i$ is item removal, clearly the current buckets reveal no information about $i$, except that it’s an aged operation. For insertion or update, suppose an item $x$ is inserted into $D$ at an aged operation $\text{op}_i$ with an assigned identifier $id^+$ upon writeback. Since each of $n$ tree paths is repeated at least once in $\ell$ recent operations, there must be a recent operation $\text{op}_j$ has the eviction path $id^+$ with overwhelming probability. Consider an equivalent sequence $\vec{\phi}$ where all of the same same random identifiers are chosen as in $\vec{p}$, with the only difference being that $\text{op}_i$ occurs at some other time. The only difference between these can be the distribution of partial blocks in the path given by
$x$’s identifier $id^+$. But since all of these partial blocks are evicted to the stash during the execution of $\text{op}_j$, writeback of $\text{op}_j$ and later operations $\text{op}_k$ will behave identically in both $\text{op}^+$ and $\text{op}^+\text{p}$. Therefore, we conclude that Hybrids 1 and 2 are statistically indistinguishable.

## C.4 Proof of Theorem 4.4

We prove the following theorem.

**Theorem 4.4.** Consider a vORAM with $T$ levels, collision parameter $\gamma$, storing $n < 2^T$ 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

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

**Proof outline.** We will mostly follow the proof of the small-stash-size theorem in Path ORAM [SvDS+13]. The proof of the theorem consists of several steps.

1. We recall the definition of $\infty$-ORAM (ORAM with infinitely large buckets) and show that stash usage in an $\infty$-ORAM with post-processing is the same as that in the actual vORAM.
2. We rely on results from the most recent version of [SvDS+14] 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 $\infty$-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 [SvDS+14].

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 4.4.** **Step 1: $\infty$-ORAM.** The $\infty$-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 $\infty$-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 \( \infty \)-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 [SvDS+14, Lemma 2], the stash size at this point is equal to the total overflow from some subtree of the \( \infty \)-ORAM buckets that contains the root. If we write \( \tau \) for that subtree, then we have

\[
|stash| > BR \iff \sum_{\text{node } v \in \tau} \text{(size of } v \text{ in } \infty\text{-ORAM)} \geq Z|\tau| + BR.
\]

**Step 3: Size of subtrees.** We prove a bound on the total size of all blocks in any subtree \( \tau \) in the \( \infty \)-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 \( \tau \), because the block sizes do not matter in the \( \infty \)-ORAM, we can simply recall from [SvDS+14, Lemma 5] that, for any subtree \( \tau \), the probability that \( \tau \) contains more than \( 5|\tau| + R/4 \) blocks is at most

\[
\frac{1}{4^{|\tau|}} \cdot (0.9332)^{|\tau|} \cdot (0.881)^R.
\]

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 A.1, the total size of all \( 5|\tau| + R/4 \) blocks exceeds \( 2(a(5|\tau| + R/4))B \) with probability at most

\[
\exp (-c_0a(a(5|\tau| + R/4))) \cdot (0.329)^{|\tau|} \cdot (0.883)^R.
\]

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 \( \exp (-2|\tau| - \frac{1}{8}R) \), which in turn is less than

\[
\frac{1}{4^{|\tau|}} \cdot (0.9332)^{|\tau|} \cdot (0.883)^R.
\]

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

\[
\frac{2}{4^{|\tau|}} \cdot (0.9332)^{|\tau|} \cdot (0.883)^R.
\]
Step 4: Stash overflow probability. As in [SvDS+14, 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 \( \tau \) having total block size greater than \((20|\tau| + R)B\) is at most

\[
\sum_{i \geq 1} \frac{d^i}{4^i} \cdot (0.9332)^i \cdot (0.883)^R < 28 \cdot (0.883)^R.
\]

D HIRB Proofs and Operation Details

D.1 HIRB Operation Details

We described the HIRB data structure in Section 5. The full details of the different subroutines are provided in Figures 12 and 13.

All the HIRB tree operations depend on a subroutine HIRBpath, which is a generator of tuples \((\ell, node_0, node_1, newcid)\), corresponding to the search path for the given label hash, for \(\ell = 0, 1, \ldots, H\). In each tuple, \(\ell\) is the level of \(node_0\), which is along the search path for the label. In the initial part of the search path, \(node_1\) is always nil, a dummy access used to preserve obliviousness. If the given label hash is actually found in the HIRB, the search path splits into two below it, and \(node_0\) and \(node_1\) will be the nodes on either side of that hash label, at level \(\ell\). Note that in the actual implementation of HIRBpath, \(node_0\) (resp. \(node_1\), if defined) corresponds to a vORAM block, evicted with identifier \(id_0\) (resp. \(id_1\)) and taken out from vORAM stash.

When each tuple \((\ell, node_0, node_1, newcid)\) 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 \(node_1\) 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 value \(newcid\) 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.

We now review the actual operations, starting with update which is the simplest among them. The update operation simply looks in each returned \(node_0\) 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 the update operation, 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 \((\ell, node_0, node_1)\) with \(\ell > \ell_h\), where \(\ell_h\) is the height of the label hash \(h\), if \(node_1\) is nil, then a new node \(node_1\) is created, and the items in \(node_0\) with a label greater than \(h\) are moved to a new node \(node_1\).
The remove operation works similarly, but instead of splitting each node below the level of the found item, the values in node0 and node1 are merged into node0, and node1 is removed by setting it to nil.

## D.2 Proofs on HIRB Trees

Next we turn to the analysis of the HIRB tree properties. Recall the following parameters and inequalities:

\[ |\text{Hash}(\text{label})| = 2H \log \beta + \gamma \]  \hspace{1cm} (5)

\[ \text{nodesize}_k = (2T + \gamma + 1) + k(2T + 2H + 2\gamma + |\text{value}|) \]  \hspace{1cm} (6)

\[ 20\text{nodesize}_\beta \leq Z \]  \hspace{1cm} (7)

We follow the same outline as [Gol10], presented here for completeness. The analysis depends entirely on the distribution of the heights for items chosen from a truncated geometric distribution over the range \{0, 1, 2, ..., H\} with probability \((\beta - 1)/\beta\).

Next we turn to the analysis of the HIRB tree properties. Recall the following parameters and inequalities:

\[ |\text{Hash}(\text{label})| = 2H \log \beta + \gamma \]  \hspace{1cm} (5)

\[ \text{nodesize}_k = (2T + \gamma + 1) + k(2T + 2H + 2\gamma + |\text{value}|) \]  \hspace{1cm} (6)

\[ 20\text{nodesize}_\beta \leq Z \]  \hspace{1cm} (7)

We follow the same outline as [Gol10], presented here for completeness. The analysis depends entirely on the distribution of the heights for items chosen from a truncated geometric distribution over the range \{0, 1, 2, ..., H\} with probability \((\beta - 1)/\beta\).

The geometric distribution is sampled pseudo-randomly and seeded by the hash of the label. To simplify the analysis, assume \(\beta\) is a power of 2. This allows us to specify exactly how the heights are chosen. These results should extend directly to the case when \(\beta\) is not a power of 2, under some assumptions on the quality of the pseudo-random number generator.

Recall from (5) that each \(\text{Hash}(\text{label})\) has \(2H \log \beta + \gamma\) bits, and thus we can say the chooseheight method chooses the random coins \(c_0, \ldots, c_{H-1} \in \{0, \ldots, \beta - 1\}\) by taking the first \(H \log \beta\) bits of \(\text{Hash}(\text{label})\), dividing into \(H\) bit strings of length \(\log \beta\), and interpreting each as an integer \(c_i\) between 0 and \(\beta - 1\), since \(\beta\) is a power of 2.

If two hash values \(\text{Hash}(\text{label}_1)\) and \(\text{Hash}(\text{label}_2)\) are chosen independent uniformly at random from \(\{0, 1\}^{2H \log \beta + \gamma}\), the corresponding HIRB levels are also chosen independently from a truncated geometric distribution over \{0, 1, ..., H\} with probability \((\beta - 1)/\beta\).

This is the motivation for the assumption on the distribution of item heights, which recall will hold whenever the hash function initializing the HIRB was chosen from a sufficiently independent random family.

### Assumption 5.1

If \(\text{label}_1, \ldots, \text{label}_n\) are any \(n\) labels stored in a HIRB, then the heights

\[
\text{chooseheight}(\text{label}_1), \ldots, \text{chooseheight}(\text{label}_n)
\]

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

We now utilize Lemma A.1 to prove the two lemmata on the distributions of the number and size of HIRB tree nodes.

### Lemma D.1

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
HIRBpath\((h, rootid, vORAM)\)

1: \((id_0, id_0^+) \leftarrow (rootid, vORAM.idgen())\)
2: \(rootid \leftarrow id_0^+\)
3: \((id_1, id_1^+) \leftarrow (vORAM.idgen(), vORAM.idgen())\) \quad \triangleright \text{dummy access}
4: \(found \leftarrow \text{False}\)
5: \(\text{for } \ell = 0, 1, 2, \ldots, H \text{ do}\)
6: \(vORAM.evict(id_0)\)
7: \(vORAM.evict(id_1)\)
8: \(\text{if } \ell = H \text{ then} (cid_0^+, cid_1^+) \leftarrow (\text{nil}, \text{nil})\)
9: \(\text{else} (cid_0^+, cid_1^+) \leftarrow (vORAM.idgen(), vORAM.idgen())\)
10: \(\text{remove } (id_0, |node_0|, node_0) \text{ from } vORAM.stash\)
11: \(\text{if } found = \text{True} \text{ then}\)
12: \(\text{remove } (id_1, |node_1|, node_1) \text{ from } vORAM.stash\)
13: \((cid_0, node_0.child_{last}) \leftarrow (node_0.child_{last}, cid_0^+)\)
14: \((cid_1, node_1.child_0) \leftarrow (node_1.child_0, cid_1^+)\)
15: \(\text{else}\)
16: \(node_1 \leftarrow \text{nil} \quad \triangleright node_1 \text{ only fetched after the target is found.}\)
17: \(i \leftarrow \text{index of } h \text{ in } node_0 \quad \triangleright node_0.h_{i-1} < h \leq node_0.h_i\)
18: \((cid_0, node_0.child_i) \leftarrow (node_0.child_i, cid_0^+)\)
19: \(\text{if } node_0.h_i = h \text{ then}\)
20: \(found \leftarrow \text{True}\)
21: \((cid_1, node_0.child_{i+1}) \leftarrow (node_0.child_{i+1}, cid_1^+)\)
22: \(\text{else}\)
23: \(cid_1 \leftarrow vORAM.idgen() \quad \triangleright \text{dummy access until } found = \text{True}\)
24: \(\text{end if}\)
25: \(\text{end if}\)
26: \(\text{yield } (\ell, node_0, node_1, cid_1^+)\) \quad \triangleright \text{Return to the caller, who may modify nodes.}\)
27: \(\text{insert } (id_0^+, |node_0|, node_0) \text{ into } vORAM.stash\)
28: \(\text{if } node_1 \neq \text{nil then} \text{ insert } (id_1^+, |node_1|, node_1) \text{ into } vORAM.stash\)
29: \(vORAM.writeback(id_0)\)
30: \(vORAM.writeback(id_1)\)
31: \((id_0, id_0^+) \leftarrow (cid_0, cid_0^+)\)
32: \((id_1, id_1^+) \leftarrow (cid_1, cid_1^+)\)
33: \(\text{end for}\)

Figure 12: Fetching the nodes along a search path in the HIRB
hirbinit($H, vORAM$)
1: $rootid \leftarrow \text{nil}$
2: for $\ell = H, H - 1, \ldots, 0$ do
3: node $\leftarrow$ new 1-ary HIRB node with child id $rootid$
4: $rootid \leftarrow vORAM.insert(node)$
5: end for
6: return $rootid$

chooseheight($\text{label}$)
1: $h \leftarrow \text{Hash}(\text{label})$
2: Initialize PRNG with $h$
3: Choose coins $(c_0, c_1, \ldots, c_{H-1}) \in \{0, 1, \ldots, \beta - 1 \}^H$ uniformly from the PRNG
4: return The largest integer $\ell \in \{0, 1, \ldots, H \}$ such that $c_1 = c_2 = \cdots = c_\ell = 0$.

insert($\text{label}, \text{value}, rootid, vORAM$)
1: $(h, \ell_h) \leftarrow (\text{Hash}(\text{label}), \text{chooseheight}(\text{label}))$
2: for $(\ell, node_0, node_1, newcid) \in \text{HIRBpath}(h, rootid, vORAM)$ do
3: $i \leftarrow$ index of $h$ in $node_0$ \hspace{1cm} $\triangleright node_0.h_{i-1} < h \leq node_0.h_i$
4: if $node_0.h_i = h$ then
5: $node_0.value_i \leftarrow \text{value}$
6: else if $\ell = \ell_h$ then
7: Insert $h, \text{value}, \text{and newcid}$ into $node_0.h_i, node_0.value_i,$ and $node_0.child_i$
8: $\triangleright$ Other items in $node_0$ are shifted over
9: else if $\ell > \ell_h$ and $node_1 \neq \text{nil}$ then
10: $node_1 \leftarrow$ new node with $node_1.child_0 \leftarrow newcid$
11: Move items in $node_0$ past index $i$ into $node_1$
12: end if
13: end for

remove($\text{label}, rootid, vORAM$)
1: $(h, \ell_h) \leftarrow (\text{Hash}(\text{label}), \text{chooseheight}(\text{label}))$
2: for $(\ell, node_0, node_1, newcid) \in \text{HIRBpath}(\text{label}, rootid, vORAM)$ do
3: if $h \in node_0$ then
4: Remove $h$ and its associated value and subtree from $node_0$
5: else if $\ell > \ell_h$ and $node_1 \neq \text{nil}$ then
6: Add all items in $node_1$ except $node_1.child_0$ to $node_0$
7: $node_1 \leftarrow \text{nil}$
8: end if
9: end for

update($\text{label}, \text{callback}, rootid, vORAM$)
1: $h \leftarrow \text{Hash}(\text{label})$
2: for $(\ell, node_0, node_1) \in \text{HIRBpath}(h, rootid, vORAM)$ do
3: $i \leftarrow$ index of $h$ in $node_0$
4: if $node_0.h_i = h$ then $node_0.value_i \leftarrow \text{callback}(node_0.value_i)$
5: end for

Figure 13: Description of HIRB tree operations.
hash function in initializing the HIRB. Then for any \( m \geq 1 \), we have

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

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 [Hoe63].

**Proof.** The HIRB has \( H \) nodes initially. Consider the \( n \) items \( \text{label}_1, \ldots, \text{label}_n \) 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 \( \text{label}_i \) into the HIRB, its height \( h = \text{chooseheight}(\text{label}_i) \) is computed from the label hash, where \( 0 \leq h \leq H \), and then exactly \( h \) existing HIRB nodes are split when \( \text{label}_i \) 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 5.1 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 A.1,

\[
\Pr[X \geq H + 4n + m] = \Pr[Y \geq 4n + m] \\
\leq \Pr[Y \geq (\mu + 1)(2n + m/2)] \\
< \exp(-c_0(2n + m/2)) \\
< \exp(-c_0m/2).
\]

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 D.2.** 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

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

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 in (6) to prove the statement of the lemma.

**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 \( m \cdot \text{nodesize}_\beta \).
First consider the items in the root node. These items all have height $H$, which according to Assumption 5.1 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 [CLRS01]) that a sample from such a distribution is at least $k$ with probability at most

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

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 $\ell$, and consider a hypothetically infinite list of possible label hashes from the HIRB which have height at least $\ell$ 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 $\ell$ before we find one whose height is at least $\ell + 1$. From Assumption 5.1, and the memorylessness property of the geometric distribution, these label heights are independent Bernoulli trials, and each height equals $\ell$ 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 branching factor greater than $m\beta$, is exactly

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

Here we use the fact that $(1 - \frac{1}{2})^x < \exp(-a)$ 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 $m \cdot \text{nodesize}_\beta$, which follows directly from $m \geq 1$ and the definition of $\text{nodesize}_\beta$ in (6).

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

**Theorem 5.3.** Suppose a HIRB tree with $n$ items and height $H$ is stored within a vORAM with $T$ levels, bucket size $Z$, and stash size $R$. Given choices for $Z$ and $\gamma > 0$, set the parameters as follows:

$$T \geq \lg(4n + \lg n + \gamma)$$

$$\beta \leq \frac{Z - (2T + \gamma + 1)}{20(4T + 2\gamma + |\text{value}|)}$$

$$R \geq \gamma \cdot \text{nodesize}_\beta$$

$$H \geq \log_\beta n$$

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

$$\Pr[\text{vORAM+HIRB failure}] \leq 30 \cdot (0.883)^7.$$  

45
Proof. We step through and motivate the choices of parameters, one by one.

The expected branching factor $\beta$ 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 D.1 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 4.4 with at least that probability.

The choice of $\beta$ is such that $Z \geq 20 \cdot \text{nodesize}_\beta$, using the inequality $H \leq \lg n < T$.

Therefore, by Lemma D.2, the size of blocks in the HIRB will be admissible for the vORAM according to Theorem 4.4.

This allows us to say from the choice of $R$ and Theorem 4.4 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 D.1 and D.2 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. ■

Theorem 5.4. 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.

Proof. Obliviousness follows from Theorems 4.1 on obliviousness of vORAM, plus Theorem 5.2 which guarantees a consistent access pattern for every HIRB operation.

For history independence, as the HIRB tree itself is uniquely represented, the only possible leakage of history information is through the vORAM pointers. Theorem 4.3 guarantees that this is limited to $O(n \log n + n\lambda)$ vORAM operations, which corresponds to $O(n + n\lambda/(\log n))$ vORAM+HIRB operations, as each HIRB operation requires $O(\log n)$ lookups in the vORAM.

We now show that vORAM+HIRB achieves secure deletion in the random oracle model. We prove the theorem using the standard hybrid argument. We consider the following hybrid experiments:

Hybrid 0: It’s $\text{EXP}_{A}^{\text{dels}}(D, \lambda, 0)$.

Hybrid 1: It’s the same as $\text{EXP}_{A}^{\text{dels}}(D, \lambda, 0)$ except the following change in the vORAM writeback operations: The most recent encryption $\text{Enc}_{\text{key}}(\text{bucket})$ in each bucket stays the same, but all the older encryptions are replaced with $\text{Enc}_{\text{key}}(0^z)$, where $z$ is the length of each bucket.

Hybrid 2: It’s the same as Hybrid 1, except that the random oracle is programmed such that the hash label for $d_0$ and $d_1$ HIRB items are the same random string.

Hybrid 3: It’s the same as $\text{EXP}_{A}^{\text{dels}}(D, \lambda, 1)$ except the following change in the vORAM writeback operations: The most recent encryption $\text{Enc}_{\text{key}}(\text{bucket})$ in each bucket stays the same, but all the older encryptions are replaced with $\text{Enc}_{\text{key}}(0^z)$.

Hybrid 4: It’s $\text{EXP}_{A}^{\text{dels}}(D, \lambda, 1)$. 
We first show that Hybrids 0 and 1 are indistinguishable. Note each encryption key in our construction is used only once, and the old root-key is never observable by the adversary since it had been in the erasable memory and was securely deleted. Therefore, due to the security of our symmetric encryption scheme Enc, the two hybrids are indistinguishable.

For Hybrids 1 and 2, recall from the definition of secure deletion that the data items \(d_0\) and \(d_1\) must be chosen so that the handles returned upon their insertion form a high-entropy distribution. In the case of the HIRB, each handle is simply the label of the data item that was inserted. Since the labels themselves are chosen from a high-entropy distribution, the hash values from the random oracle must also have a high-entropy distribution, and therefore Hybrids 1 and 2 are indistinguishable. (Note \(A_3\) doesn’t get the state information from \(A_1\) or \(A_2\).) Symmetrically, Hybrids 2 and 3 are also indistinguishable.

The indistinguishability between Hybrids 3 and 4 holds symmetrically to the case of Hybrids 0 and 1.

\[\square\]