

Hashing Beats Trees: Practical Oblivious Dictionaries in SGX

Erik-Oliver Blass¹ and Travis Mayberry²

¹ Airbus, Germany
erik-oliver.blass@airbus.com

² United States Naval Academy, USA
mayberry@usna.edu

Abstract. Oblivious RAM (ORAM) is a powerful cryptographic primitive that hides both the contents of outsourced data and the access pattern to the data. However, it offers only a restrictive array-based interface. A principal obstacle to its deployment is building an efficient oblivious dictionary upon it. To build such a dictionary, the state-of-the-art layers an oblivious AVL tree over ORAM, avoids an expensive local position map, but forces every operation to traverse a full root-to-leaf path, costing $\Theta(\log n)$ ORAM accesses.

We show this trade-off to be suboptimal, particularly when deployed in a common setup within a trusted execution environment like SGX. There, a small amount of client memory can reduce the position map recursion depth to three levels or less, rendering its overhead relatively small. With low position map cost, probabilistic hash tables prevail as dictionary data structures due to their small, constant number of accesses compared to $\Theta(\log n)$ path traversal of a tree.

We implement and analyze oblivious, probabilistic hash tables over Path ORAM, including several different strategies: chaining, power-of-two-choices, stash-less cuckoo, and de-amortized cuckoo hashing. Execution of these algorithms within a secure enclave requires "double obliviousness", which in turn needs strict concrete maximum cost limits for each access that current literature does not provide. We derive these concrete padding parameters bounding the hash table rebuild probability to $\leq 2^{-40}$ via large-scale simulation. In an SGX deployment, our constructions outperform oblivious trees by up to $20\times$ for lookups and $12\times$ for inserts.

1 Introduction

ORAM is a cryptographic primitive that enables a client to read from and write to an untrusted database server while hiding the plaintext values of the data as well as the access pattern: which records are read or written to at which times. Acting as a generic compiler, it can transform any algorithm into a secure outsourced equivalent by routing the algorithm's memory operations through the ORAM, enabling black-box construction of privacy-preserving protocols from non-private counterparts. This makes ORAM a very powerful technique. Yet,

despite three decades of research, culminating in constructions with asymptotically optimal overhead for over a decade [8, 11, 22, 23, 25, 29, 33, 39, 43, 47, 48], ORAM has seen limited real-world adoption outside of trusted-hardware deployments. This discrepancy can be attributed to three primary factors.

First, even asymptotically optimal ORAM constructions incur polylogarithmic overhead in the number of data blocks n for bandwidth [29] and require $\omega(1)$ rounds latency [11], usually $\Theta(\log n)$, under standard assumptions. Depending on the concrete ORAM technique used in practice, this overhead can be 100x or more compared to an unsecured access which is often cost-prohibitive.

Second, ORAM protocols are stateful. The client maintains local data that changes with each access. In recent ORAMs, this data is a *position map* tracking the location of each data block. As this position map scales linearly in n , protocols store it recursively in smaller ORAMs until it reaches constant size, at a significant cost to overall efficiency. This statefulness also confines ORAM to a single client, as multiple clients accessing would otherwise require separate ORAMs or a trusted proxy.

Third, the standard interface to an ORAM is array-based, where the client accesses a single block from a contiguous memory space. Yet, many real-world applications require a higher-level key-value (“dictionary”) interface at a minimum. The running example in this paper is the Signal messenger’s contact discovery [14, 34, 35] protocol. In this application, ORAM allows a user to determine which of their contacts is also a user of Signal, and hence that they can initiate communication with them over the service, without the server learning which users are contacts of each other. This requires obviously managing user contact information, keyed by unique per-user identifiers, in a large outsourced key-value storage.

Recent research [16, 36, 49] mitigates these issues using trusted execution environments (TEEs), i.e., hardware enclaves such as Intel’s SGX. An enclave on an untrusted server guarantees to the client that a specified program executes and that none of the program’s data leaks outside the enclave. It acts as a trusted proxy, executing queries from, potentially, multiple clients, without revealing information to the server, who is prevented from tampering with it. As client-to-enclave communication is protected by end-to-end encryption, the enclave serves as an agent that implements the dictionary data structure on top of ORAM and runs the ORAM protocol on the client’s behalf. This shifts the overhead from network round trips to enclave-host communication on the same machine, which is orders of magnitude faster.

The TEE approach has already seen practical adoption: Signal runs Path ORAM [47] inside an SGX enclave for the above contact discovery [14]. The dominating costs for the enclave are the number and size of accesses to the ORAM’s contents, stored, e.g., on a physical disk. To reduce the number and size of accesses, prior work, including Signal, builds a key-value storage over an array-based ORAM by using an AVL-tree-based Oblivious Data Structure (ODS), tailored specifically for ORAM [16, 36, 49]. Such a tailored tree-based ODS can be built without a position map, and reduces a naive oblivious tree’s

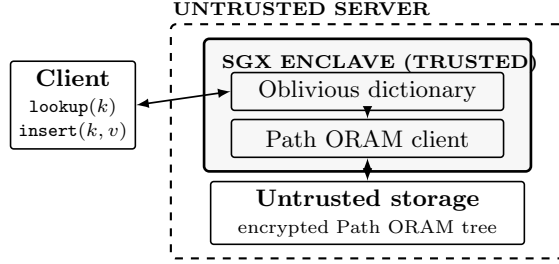


Fig. 1: System architecture. A remote client sends dictionary queries over a secure, attested channel to the trusted SGX enclave hosted by an otherwise untrusted server.

overhead over insecure access from $O(\log^3 n)$ to $O(\log^2 n)$ using only $O(1)$ client state [50]. Oblivious AVL trees thus dominate the ODS literature.

Our contributions. Our main observation is that with realistic enclave memory, Path ORAM’s position-map recursion can be made shallow, so the AVL tree’s $\Theta(\log n)$ root-to-leaf traversal, which is necessary in order to remove the position map, outweighs the savings you get from removing the position map in the first place. Hash tables, being far more efficient in terms of accesses, can then beat AVL trees in this setting. That observation motivates the following contributions:

- We construct oblivious dictionaries with four probabilistic hash tables: chaining, power-of-two-choices, stashless cuckoo, and deamortized cuckoo hashing on top of Path ORAM.
- We perform a large-scale empirical analysis establishing concrete parameters for each scheme at a failure probability of at most 2^{-40} . Such parameters have not been derived for the latter three of these data structures and may be of independent interest.
- Concrete Path ORAM position map and block size parameters, showing that 1 MByte client memory bounds recursion to 2 levels even for 2^{29} blocks, versus the standard $O(\log n)$.
- A double-oblivious SGX implementation, benchmarked against an AVL-tree ODS [49, 50] for n up to 2^{25} . The source code is published along with this paper [5].

Figure 1 summarizes the system model considered throughout this paper. A remote client issues dictionary operations to a trusted SGX enclave running within an otherwise untrusted server. The enclave implements an oblivious dictionary on top of a Path ORAM client, while the encrypted ORAM tree is stored outside the enclave in untrusted storage.

2 Background

Preliminaries and Notation. In this paper, we write $[n]$ as a shorthand for the n -element set $\{0, \dots, n-1\}$. For an array A of blocks, $A[i]$ denotes the i^{th} block of A . Moreover, n denotes the number of key-value pairs a data structure can store, m denotes the actual size of the underlying data structure, specifically the number of hash table buckets or number of nodes in a tree, and η denotes the number of blocks in an oblivious RAM. If not mentioned otherwise, all logarithms are taken with base 2. The terms dictionary, key-value storage, and map or associative map are used interchangeably.

We focus on dictionary data structures, i.e., a key-value map $D = (\text{init}, \text{insert}, \text{lookup})$ to efficiently store and retrieve key-value pairs (k, v) where keys $k \in \mathcal{K}$ and $v \in \mathcal{V}$. For simplicity, keys from universe $\mathcal{K}$ and values from universe $\mathcal{V}$ are represented as bit strings of fixed lengths $s_{\mathcal{K}}$ and $s_{\mathcal{V}}$. Dictionary D realizes the two operations **insert** and **lookup** in the natural way. That is, operation **insert** (k, v) stores key-value pair (k, v) . As D is a map, a key k is inserted at most once. Operation **lookup** (k) searches for key k and outputs either v (if it was stored before) or $\perp$. Operation **init** (n) initializes the dictionary data structure such that it can hold up to n key-value pairs. As previous work [50], we ignore deletions for clarity and discuss them later in Section E.4.

2.1 Oblivious Key-Value Storage

The adversary model we consider is that of a benign client that wants to store and access their data on an untrusted storage provider. A very general setup is that the storage provider offers the client an interface to **write** from and **read** into specific storage cells of the storage offered (RAM model). Again, these operations are natural with **write** (i, data) writing bit string **data** into storage cell number i , and **read** (i) reading the contents from storage cell i . To then implement the **init**, **insert**, and **lookup** operations for their specific data structure, the client performs sequences of calls to **write** and **read**.

The *trace* of a sequence $\mathcal{S}$ of **insert** (k, v) and **lookup** (k) operations is everything the adversary can observe from the resulting **write** (i, data) and **read** (i) operations on their data storage. That is, the adversary learns all operation types **write** or **read** and their parameters i and **data**.

Informally, the security challenge is now that, from the *trace*, the adversary should not learn anything non-trivial about $\mathcal{S}$. More formally, recall the definition of oblivious data structures from Wang et al. [50]:

Definition 1 (Oblivious Data Structure). Let λ be a security parameter, $D = (\text{init}, \text{insert}, \text{lookup})$ be a data structure, and $\text{op}_i \in \{\text{insert}(k_i, v_i), \text{lookup}(k_i)\}$ be operations.

Data structure D is oblivious, iff there exists a simulator Sim such that for all sequences $\mathcal{S} = (\text{init}(n), \text{op}_1, \dots, \text{op}_M)$ and distinguishers $\mathcal{A}$, there exists a

function ϵ and a sufficiently large λ with

$$|Pr[\mathcal{A}(1^\lambda, \text{trace}(\mathcal{S})) = 1] - Pr[\mathcal{A}(1^\lambda, \text{Sim}(1^\lambda, \mathcal{L}(\mathcal{S}))) = 1]| \leq \epsilon(\lambda),$$

where $n, M \in \text{poly}(\lambda)$, and ϵ is negligible in λ , and $\mathcal{L}$ is a leakage function.

Leakage function $\mathcal{L}$ defines what an adversary might learn about $\mathcal{S}$. In the context of the oblivious data structures in this paper, $\mathcal{L}(\mathcal{S})$ is n , M , and the type of each operation op_i , i.e., whether op_i is a `lookup` or `insert` [50]. We discuss a variation of this leakage in Section E.3.

We will come back to Definition 1 later in the description of the oblivious data structures. There, we will indicate why the data structures are oblivious. However, the main idea behind the argument will always be that we use them on top of an ORAM. So, if each `lookup` or `insert` operation of the oblivious data structures always translates to a fixed number of accesses to the underlying ORAM, obliviousness follows immediately.

We will now briefly revisit the main properties of Path ORAM, the concrete ORAM we use as a building block.

2.2 Path ORAM

An ORAM is a cryptographic protocol enabling a client to access an array A of η blocks, each of size B Bit, stored on an untrusted server [22]. Similar to the definition of obliviousness for oblivious data structures (Definition 1), an ORAM hides both the data contents and the access pattern, i.e., which block $A[i]$ is read or written. Unlike an ODS, ORAM also hides whether the access is a read or a write. The ORAM protocol’s efficiency is measured by its bandwidth overhead and the number of communication rounds between client and server required for each read or write operation.

Path ORAM [47] is a prominent, efficient ORAM protocol with multiplicative bandwidth overhead of $O(\log^2 \eta)$ and $O(\log \eta)$ rounds of communication. In the following, we summarize Path ORAM’s key ideas and refer to the original Path ORAM paper for exact details.

To represent an array A of η blocks in an oblivious way, Path ORAM creates a binary tree with a total of $2\eta - 1$ nodes and η leaves. Each node has space for Z data blocks. This binary tree is stored in an encrypted fashion on the server. Each data block $A[i]$ is assigned to a randomly chosen leaf L_i , and this assignment is stored in a *position map*, an array Pos of size η elements, i.e., $\text{Pos}[i] = L_i$. The invariant in Path ORAM is that at any time block $A[i]$ can always be found in a node of the tree on the path between the root and leaf $\text{Pos}[i]$.

A small, but important technicality in the context of Path ORAM is that η always has to be a power of 2. To ease the exposition, we do not explicitly mention this each time in the description of our hash tables in Section 3 below. However, whenever a data structure requires an underlying Path ORAM with

η blocks, η has to be always rounded to the next power of two. Our evaluation and benchmarks in sections 4 and 6 implicitly perform the rounding, too.

Position Map. A key technical challenge in the above protocol is to efficiently manage position map **Pos**. This position map has size $\eta \cdot \log \eta$ Bit which scales asymptotically faster than the η blocks of array A and has to be stored somewhere. In the special case where size $B = \log \eta$ (no payload), the size of **Pos** would even be equal to the size of the ORAM itself. If a client can store and access **Pos** locally, then one could argue that the client could also store and access array A locally, too, and there would be no need for an ORAM.

Thus, a key idea of Path ORAM is for the client to outsource and store position map **Pos** recursively in another Path ORAM that is called a *position map Path ORAM*. If the block size of this position map Path ORAM is $B' = c \cdot \log \eta$, $c > 1$, then you can store all η mappings from the main Path ORAM in $\frac{\eta}{c}$ blocks, each of size B' Bit. Observe that the position map **Pos'** of this position map Path ORAM has size $\frac{\eta}{c} \cdot \log \frac{\eta}{c}$ Bit. Now, the recursion would continue and store the new position map **Pos'** in another position map Path ORAM and so on. After $\log_c \eta$ levels of recursion, this position map recursion terminates with a position map of a size in $O(1)$. As this last position map is typically small (c blocks), the client finally stores it locally.

First Insight: Recursion Is Almost Free in a TEE. Each access to a block in the (main) Path ORAM also implies accesses to the recursive position map Path ORAMs, incurring a slowdown by a factor of $O(\log \eta)$ due to the larger amounts of data exchanged, the additional disk accesses, and the additional rounds of communication. For larger values of η , e.g., $\eta = 2^{20}$, this becomes significant in practice. Our *first insight* is that, in our setting, this asymptotic overhead is largely illusory in practice. A Path ORAM client, even when implemented inside a trusted execution environment like Intel SGX, features T Bit of local memory, where T is often much more than the few 32 Bit blocks assumed by prior work. Moreover, in real-world applications, block size B is much larger than $\log \eta$, e.g., $\log \eta$ plus a few hundred Bit payload for a public key or identifier in a Signal-like address book, or $\log \eta$ plus KBytes to MBytes for files. So, the position map ($\eta \cdot \log \eta$ Bit) is far smaller than the ORAM data ($\eta \cdot B$ Bit). Hence, although T is too small to hold the whole ORAM, it can store a relatively large position map, keeping the recursion depth small, e.g., at most two or three levels for many realistic values of η (Section 5). This eliminates nearly all of the $O(\log \eta)$ factor and significantly improves overall performance.

One might argue that granting T Bit of local memory to store a larger position map and then comparing to AVL trees which do not require such memory is unfair. For example, AVL tree performance might also benefit from local memory by standard techniques such as caching. We discuss in detail why this is not the case, and why AVL trees cannot meaningfully benefit from additional memory, in Section E.5.

2.3 SGX Trusted Execution Environment

Intel’s Software Guard Extensions [26] realize a trusted execution environment (TEE) protecting a program against an untrusted host computer and operating system that are running the program.

SGX provides hardware-enforced integrity and confidentiality for the enclave’s code and data. While the enclave’s RAM accesses are transparently encrypted by a memory encryption engine, its interface with the untrusted host is realized via two primary mechanisms. For I/O, the enclave issues *OCALLs* to the host OS, and the enclave program is responsible for the confidentiality and integrity of any data crossing this boundary. For network communication, an attestation mechanism allows remote clients to verify the enclave’s code and establish a secure channel, isolated from the host.

ORAM inside SGX. To outsource an array-based data structure to an untrusted server and still guarantee obliviousness, a popular approach [14, 16, 36, 49] is to move the core ORAM techniques into an SGX enclave. That is, the enclave implements the actual ORAM, e.g., Path ORAM, and offers a simple read and write API to a remote client who is connected over the secure network connection. The actual Path ORAM implementation is run inside the enclave, unobserved by the untrusted host computer. This Path ORAM implementation in the enclave performs disk accesses to implement reads and writes to large amounts of data in an oblivious fashion.

Enclave Memory. The memory available for the enclave and thus the Path ORAM implementation is the same as for any process running on the host, essentially the host’s entire RAM. However, there is a significant performance penalty for random memory accesses as soon as the amount of memory allocated by the enclave exceeds the size of a configurable *enclave page cache* (EPC) due to SGX paging [7]. While the EPC size was limited to ≈ 96 MByte for SGX V1, this restriction has been lifted to 512 GByte for SGX V2. In any case, when executed in an enclave, Path ORAM has far more than two blocks of 32 Bit memory T available as often assumed, e.g., in the order of MiB or even GiB (see Section 6).

2.4 Double Obliviousness

It is somewhat surprising, then, to say that when a Path ORAM client is run in a TEE, the untrusted host computer may be able to observe the Path ORAM algorithm’s internal memory access patterns, control flow, and learn secret information about the ORAM’s state [30, 36, 46, 51]. This is because, although the memory accesses are encrypted, a careful host device can potentially learn which branches of a program are being executed by observing memory accesses and cache timings. Branch behavior can reveal which part of the data structure is being accessed, which is exactly what ORAM is trying to hide.

A standard technique to mitigate learning from such TEE side-channels is called double obliviousness [36]. All code running in the enclave is implemented

in a way that, e.g., its loops run a public number of times, the code is either branch-free or branches do not depend on secret information, and accesses to internal data structures like arrays are implemented like a trivial, linear-scan ORAM. That is, for each access, all elements of the whole data structure are accessed to hide the element of the data structure.

Making code double oblivious is standard, mostly straightforward [1, 36, 45, 46], and even compiler extensions [32, 52] exist to automate this task. To keep the exposition clean, the description of the pseudo-code of the oblivious hash tables below omits such standard techniques for double obliviousness besides highlighting loops for the hash table parameters. Our implementation (available for inspection at [5]) used in the evaluation sections 4 and 6 is fully double oblivious.

2.5 Oblivious Data Structures

Wang et al. [50] present an AVL-based binary search tree (BST) on top of Path ORAM to realize an oblivious key-value storage. The AVL tree and its operations for a total of n nodes is implemented on top of an ORAM array with $\eta = n$ blocks in the textbook way [15].

The key idea of their work is that the underlying Path ORAM does not need a position map, but the nodes of the AVL tree store the leaf numbers for the Path ORAM tree. Specifically, each node in the AVL tree contains not only the pointers to their left and right children nodes in the AVL tree, but also the children’s leaf numbers (position map entries) in the Path ORAM tree. Consequently, searching for a key requires $O(\log n)$ accesses to the tree, each of them with $O(\log n)$ accesses (and not $O(\log^2 n)$), to the underlying non-recursive Path ORAM.

Second Insight: The Tree is the Real Bottleneck. To make the AVL tree oblivious following Definition 1, each **lookup** operation must be padded to the maximum possible search-path length $\log n$ in the AVL tree, and each **insert** operation must additionally perform as many accesses to the underlying Path ORAM as the most expensive AVL rebalancing, double rotation, requires five pointer updates. Our *second insight* is that this factor of $\Theta(\log n)$, iterating from root to leaf in the AVL tree, for every operation, is the main bottleneck of AVL-based ODS. Probabilistic hash tables introduce only a small constant factor of 2 or 3 for a **lookup** or **insert** as we will now see in Section 3. Moreover, once the position-map recursion is kept shallow (our first insight), oblivious hash tables outperform oblivious AVL trees, as we demonstrate in Section 6.

3 Probabilistic Hash Tables

Instead of using binary search trees to realize an oblivious key-value storage, our main idea is based on probabilistic hash table constructions. Informally, we will build hash tables on top of Path ORAM to achieve high efficiency. The actual hash table data, i.e., individual bucket contents, are going to be represented as

blocks of the ORAM. All accesses to hash table data will be translated into accesses to ORAM blocks.

While hash tables do not require the $\Theta(\log n)$ blow-up of AVL-trees for `lookup` or `insert` operations, they introduce two main technical challenges that we will discuss in detail below.

Discussion. Note that probabilistic hash table insertions might fail. However, we will set parameters such that the probability of a failure is extremely low at 2^{-40} in practice. We present parameter details in Section 4.1 and discuss more in Appendix E.1. We also stress that this paper considers performance at concrete values of n . For reference, however, we compare the asymptotic performance of all oblivious dictionaries in this paper in Appendix B and Table 1.

3.1 Hash Table Challenges

Like binary search trees, hash tables implement an associative array with `lookup` and `insert` operations. In scenarios where security does not matter, hash tables are more efficient than BSTs in expectation, because they provide an average-case time complexity of $O(1)$ for search (`lookup`), insertion (`insert`), and deletion operations. However, hash tables introduce two significant difficulties when used as an oblivious key-value store, i.e., in a setting where access pattern privacy is required.

First, the access pattern to the hash data structure is data-dependent. An adversary observing an access to the bucket with index $H(k)$ of a hash table $\mathcal{T}$ automatically learns the hash value of k . To mitigate this access pattern leakage, we will implement an oblivious hash table $\mathcal{T}$ with m buckets using an ORAM with m blocks in the natural way. The i^{th} bucket of $\mathcal{T}$ corresponds to the i^{th} block of the ORAM.

The second difficulty turns out to be more challenging: hash tables typically feature much fewer buckets m than the size of the universe $|\mathcal{X}|$ which leads to collisions, and collision resolution becomes both expensive and leads to access cost that depends on the block being accessed.

Collision Resolution. A standard collision resolution technique such as linear probing (open addressing) leads to a *worst case* complexity for access operations in $O(m)$. Since the oblivious key-value storage must hide access patterns, and all accesses must have the same length, this means that a naive implementation of an oblivious hash table with linear probing would always require the worst case of $O(m)$ accesses *per* operation to hide how many accesses were required for any specific operation and input. As a consequence, $O(m)$ access operations to the underlying ORAM are necessary, too.

Otherwise, simple distinguishing attacks on such hash tables on top of an ORAM become feasible. For example, consider the following two sequences of accesses: after inserting n keys into the hash table, one sequence retrieves the same key n times, and the other retrieves n different keys. In the first case, the size of the trace (the underlying ORAM accesses) would be smaller than in the second case, depending on how many collisions occurred. An adversary $\mathcal{A}$ could

easily distinguish between these two cases, breaking security guarantees of the oblivious key-value storage (Definition 1).

Consequently, we investigate other hashing techniques for collision resolution and avoidance than linear probing with the goal of better *worst-case* performance. While the actual hash data structure itself will be stored inside an ORAM similar to oblivious binary search trees (ODS), our focus will be on *probabilistic* hashing techniques. In this context, probabilistic hashing means that an *insert* operation might fail with a certain probability depending on hash table parameters. The idea is then to limit this probability of failure to a small value $2^{-\sigma}$, where σ is a failure parameter such as $\sigma = 40$, by choosing an appropriate parameter set. The advantage of accepting a tiny 2^{-40} failure probability for hashing on top of Path ORAM allows achieving better overall performance than oblivious binary search trees.

In the (rare) event of a failure, the client would rebuild the whole hash table with new hash functions. We discuss the implications of a failure during *insert* and why a hash failure is not a security issue in our specific situation in Appendix E.1.

The probabilistic hash tables techniques we use in this paper are chaining, power-of-two-choices hashing, stashless cuckoo hashing, and deamortized cuckoo hashing. We discuss these choices in Appendix E.2.

3.2 Chaining (CHAIN)

A standard collision resolution technique is chaining, where each of the m hash table buckets comprises all key-value pairs (k_i, v_i) with the same hash value $H(k_i)$ simply by chaining these key-value pairs. As the number of collisions mapping different keys to a specific bucket is data dependent and not known in advance, each bucket is typically realized as a linked list, so all key-value pairs of a bucket are stored as a chain of unbounded length.

If the length of a bucket's chain is data dependent, all chains must be padded to the same size β for obliviousness. Each *lookup* or *insert* operation must take β steps and β accesses to the underlying ORAM. In the worst case, all keys map to the same bucket, so $\beta = n$ which makes oblivious chaining inefficient. However, the probability that $\beta = n$ is small in practice. Raab and Steger [42] present a bound for the case that H is a cryptographic hash function and $m = n$. In that case, the length of the longest chain in any bucket is upper-bound by $\beta \in O(\frac{\log n}{\log \log n})$ with probability $1 - o(1)$.

Obliviousness. An oblivious hash table $\mathcal{T}$ with chaining can be implemented on top of a Path ORAM as follows. If the table comprises $m = n$ buckets, a Path ORAM with $\eta = \beta \cdot n$ blocks is initialized. To place (k, v) in bucket $i = H(k)$, always iterate over all β ORAM blocks $[i \cdot \beta, \dots, (i + 1) \cdot \beta - 1]$. Store (k, v) at the first empty block, but hide the location of this empty block by performing dummy write operations on the $\beta - 1$ remaining blocks. Similarly, to look up (k, v) , always iterate over these β ORAM blocks and search for k .

Algorithm 1 in Appendix A presents the pseudo-code for the resulting (probabilistic) chaining technique. The j^{th} block of bucket i , i.e., Path ORAM bucket $i \cdot \beta + j$, is denoted as $\mathcal{T}[i][j]$.

We again stress that our pseudo-code omits standard techniques for double obliviousness [36] to ease the exposition. In each iteration, the ORAM write to $\mathcal{T}[i][j]$ has to be performed regardless of whether the current entry is the first empty one. As discussed in Section 2.4, if the entry should not be updated, a dummy ORAM access is performed.

Concrete Parameters. To make this mechanism practical, we need a concrete value for β that results in a known failure probability. Exact values for β that result in low probabilities of failure are known, and we discuss them in Section 4.1.

3.3 Power of Two Choices Hashing (POTC)

To reduce the length β of the longest chain of any bucket, Azar et al. [9] suggest a simple load balancing mechanism during insertion. Instead of storing key-value pair (k, v) in table $\mathcal{T}$ at the bucket index given by hash output $H(k)$, let there be two different hash functions H_0 and H_1 . To store (k, v) , compute $i_0 = H_0(k)$, $i_1 = H_1(k)$ and store (k, v) either at bucket $\mathcal{T}[i_0]$ or at bucket $\mathcal{T}[i_1]$ depending on the buckets' current chain length. That is, if the chain of bucket i_0 contains fewer key-value pairs than the one of bucket i_1 , append (k, v) to bucket $\mathcal{T}[i_0]$, otherwise append to bucket $\mathcal{T}[i_1]$. Corresponding function $\text{lookup}(k)$ searches both buckets $\mathcal{T}[i_0]$ and $\mathcal{T}[i_1]$ for k .

This Power of Two Choices (POTC) paradigm reduces the length β of the longest chain exponentially. For a table with $m = n$ buckets, the length of the longest chain is upper-bounded by $\beta \in O(\log \log n)$ with probability $1 - o(1)$. Azar et al. [9] also show in the proof of their Theorem 3.2 that, as long as $\beta \leq \log \log n + O(1)$, the probability that a single bucket's chain exceeds length β is upper-bounded by $2^{-\Omega(2^\beta)}$. By the union bound, the probability that at least one of the n chains exceeds length $\beta \leq \log \log n + O(1)$ is then upper-bounded by $n \cdot 2^{-\Omega(2^\beta)}$.

Obliviousness. To make POTC hashing oblivious, we use the same setup as with the chaining technique. Each of the m buckets is represented as a fixed-size array of β ORAM blocks, i.e., we have an ORAM with total of $\eta = \beta \cdot n$ blocks. To insert a pair (k, v) , iterate over the β blocks of both buckets $H_0(k)$ and $H_1(k)$ and place (k, v) into the first available block. To lookup a pair (k, v) , also iterate over both buckets' blocks and find the matching key.

We formalize the resulting insert and lookup operations for this oblivious POTC hashing in Algorithm 2 in Appendix A.

Concrete Parameters. To make this mechanism practical, the crucial parameter that we will estimate in Section 4.3 is again β and its impact on the probability of failure.

3.4 Stashless Cuckoo Hashing (SLCH)

Another approach based on load balancing is Cuckoo hashing [19, 20, 27, 37, 38]. While there are several variations of Cuckoo hashing, we will focus on a “stash-less” version [40] with three hash functions in this paper.

Let there be u hash functions $H_0, \dots, H_{u-1}$ and an initially empty table of $m = \epsilon \cdot n$ buckets. Each bucket has space to store exactly one key-value pair and an index of a hash function that was used to store that pair. That is, a bucket is of the form $(key, value, index)$.

To insert a new pair (k, v) , do the following:

- Check the contents of bucket $H_0(k)$. If that bucket is empty, store $(k, v, 0)$ there.
- However, if bucket $H_0(k)$ is not empty, but contains (k', v', idx') , store $(k, v, 0)$ in bucket $H_0(k)$, evicting (“kicking out”) previous content (k', v', idx') .
- Now, continue by placing $(k', v', (\text{idx}' + 1) \bmod u)$ in bucket $H_{\text{idx}' + 1 \bmod u}(k')$ kicking out any key-value pair stored there before. Continue placing this key-value, and so on.

The process of evictions repeats until either an empty bucket is found or a threshold of **MaxKicks** evictions is exceeded. There are two reasons why an empty bucket is not found after **MaxKicks** evictions. First, inserting key k leads to a cyclic eviction chain (“second cycles” in the corresponding Cuckoo graph) which leads to an infinite loop of evictions. Second, **MaxKicks** evictions might not be enough to find the correct placement of the chain of keys. To cope with both cases, the pending key-value pair is typically placed into an auxiliary *stash* of size s .

To **lookup** a key-value pair (k, v) , simply check the u buckets given by $H_0(k), \dots, H_{u-1}(k)$ and the stash.

It has been shown that a stash with space for $s \in \Theta(\log n)$ is sufficient and no infinite loop occurs with probability $O(n^{-s})$, if **MaxKicks** $\in O(\log n)$ [23]. More recently, Pinkas et al. [40] have estimated the concrete impact of parameters ϵ and u on the size of the stash and the probability of no infinite loop. For the concrete case of $\epsilon = 1.27$ and $u = 3$, they even show that no infinite loop occurs during the insertion of n key-value pairs, and no stash is required ($s = 0$) with probability $1 - 2^{-40}$. It is this stash-less variant of cuckoo hashing that we will focus on.

In case no infinite loop occurs, **MaxKicks** has an exponential impact on the probability that a single key-value pair can be inserted within **MaxKicks** evictions. The probability that more than **MaxKicks** evictions are required is in $2^{-\Omega(\text{MaxKicks})}$ for any insertion [38]. By a simple union bound, the probability of failure during all n key-value insertions is then $\leq n \cdot 2^{-\Omega(\text{MaxKicks})}$.

Obliviousness. We apply this parameter setting of $\epsilon = 1.27$ and $u = 3$ to (I) get high hash table bucket utilization compared to $u = 2$ [40], thus reducing the number of ORAM blocks required, and (II) to avoid dealing with a separate stash in an oblivious manner at an extra cost. To make this stashless Cuckoo hashing oblivious, we represent the buckets of the hash table in a Path ORAM

of $\eta = 1.27 \cdot n$ blocks. To hide the exact number of kicks necessary to place a key-value pair, we need to always perform **MaxKicks** kick-out operations.

We formalize the resulting **insert** and **lookup** for stash-less cuckoo hashing with $\epsilon = 1.27$ and $u = 3$ hash functions in Algorithm 3 in Appendix A.

Concrete parameters. For stashless Cuckoo hashing, the only missing parameter that we will need to estimate in Section 4.4 is the number of **MaxKicks** and the resulting probability of a failure.

3.5 Deamortized Cuckoo Hashing (DEAM)

Cuckoo hashing without a stash has asymptotically optimal complexity for lookup operations and constant space overhead with low constants, e.g., with $u = 3$ hash functions and $\epsilon = 1.27$. However, as we will see in the evaluation in Section 4, a main performance issue of stash-less cuckoo hashing is the value of **MaxKicks** which becomes large in practice, e.g., **MaxKicks** = 60 for $n = 2^{20}$. Thus, an oblivious cuckoo hash table has to always perform the worst-case of 60 kick-out operations even though in expectation there is no kick-out necessary, making **insert** expensive.

Consequently, there have been constructions that [6, 27] deamortize the Cuckoo hashing operation **insert**. Let there be two hash functions H_0 and H_1 , two hash tables of $m = \epsilon \cdot n$ buckets each, and an auxiliary data structure called a queue. The main idea is that for each **insert**(k, v) operation, $(k, v, 0)$ is first appended to the queue. Here, 0 indicates the hash function index, i.e., (k, v) is going to be placed in $\mathcal{T}_0$ using the hash function H_0 .

Then a constant number of $L > 1$ key-value pairs is dequeued and placed in one of the two tables as follows.

Run these two steps L times:

- Remove key-value pair (k', v', idx') from the head of the queue and place it in bucket $H_{\text{idx}'}(k')$ in Table $\mathcal{T}_{\text{idx}'}$.
- In case there already was an element (k'', v'', idx'') residing in $\mathcal{T}_{\text{idx}'}[H_{\text{idx}'}(k')]$, place $(k'', v'', (\text{idx}'' + 1) \bmod 2)$ at the front of the queue.

As $L > 1$, in expectation there are more elements being removed from the queue than added during an **insert**. So over a long sequence of operations, there should be pressure on the queue pushing it toward zero occupancy. However, in the short term there can be chains of kicks that do not resolve in the L steps allocated by the algorithm. These are temporarily stored in the queue. Also, as discussed above, cuckoo hash graphs can form cycles of kicks that result in some elements never being permanently placed in either table. To detect cycles of elements kicking-out each other infinitely, a second auxiliary data structure, the cycle detection mechanism (CDM), is used as follows.

- Before placing key-value pair (k, v, idx) by kicking out another key-value pair, check whether (k, v, idx) is already in the CDM. If it is, add (k, v, idx) to the back of the queue and empty the CDM.

- Otherwise, place (k, v, idx) at its intended location, kicking-out the previous key-value pair, add (k, v, idx) to the CDM.

To **lookup** key-value pair with key k , check in the two locations $\mathcal{T}_0[H_0(k)]$, $\mathcal{T}_1[H_1(k)]$ and in the queue. Arbitman et al. [6] show that both the queue and the CDM can be joined together in a larger queue, containing both temporary items that will eventually be flushed from the queue as well as permanently sequestered cycles that will stay there permanently. The size ℓ of this larger queue is shown to be $\ell \in O(\log n)$ with high probability. As ℓ is tiny in practice, and in our setting the cost of **insert** is dominated by the number of external memory accesses, we represent the queue as an array stored in the enclave’s memory. An access with a linear scan over this array only incurs minor overhead. Crucially, we must bound the size ℓ as it has to be capped at initialization time and linear scanned during access. Asymptotic guarantees are not sufficient.

The queue-size bound ℓ has two components: the number of elements in the queue due to cycles, and the number of elements in the queue due to temporary backups in the normal cuckoo kicking process. Elements that cannot be placed because they are in a cycle are accounted for using the stash bound of Kirsch et al. [27]. For a table of size n , the probability that at least ℓ such elements exist is $O(n^{-\ell})$. For the second contribution to the queue, regular backlog, Arbitman et al. [6] argue over a connected-component bound over batches of $\log n$ insertions, sums of connected-component sizes in the cuckoo graph, whose tails decay as $2^{-\Omega(\ell)}$ for $\epsilon > 1$.

This bound cannot be used to generate concrete parameters, as the probability only depends on n , queue size ℓ is given asymptotically, and the bound only applies existentially for *some* value of L . Consequently, Arbitman et al. provide an evaluation with $\epsilon = 1.2$ and $L = 3$. However, they only show the maximum queue occupancy for each value of n , so it is difficult to extrapolate to higher confidence values.

Obliviousness. To realize an oblivious version of deamortized Cuckoo hashing, we store both tables in a Path ORAM with $\eta = 2\epsilon n$ blocks. We set parameters $\epsilon = 1.2$ and $L = 3$ as Arbitman et al. and formalize the resulting **insert** and **lookup** operations in Algorithm 4 in Appendix A. Note that operations **add2QueueBack**, **add2QueueFront**, and **add2CDM** could overflow leading to a failure.

Concrete parameters. Section 4.5 will empirically estimate ℓ using the known relationship of $\ell \in O(\log n)$, while gathering statistics about failure rates at different sizes over many runs, to also relate ℓ to the probability of failure.

3.6 Optimization: Larger ORAM Block Sizes

To keep the exposition clear, we have omitted two straightforward optimizations. They apply to both hash table chaining and the power of two choices hashing.

As we will see in our evaluation in Section 6, the larger the number η of blocks in a Path ORAM, the larger the time for an access operation. More specifically, β accesses to an ORAM with $\eta = \beta \cdot n$ blocks of size s each are slower than 1 access to an ORAM with $\eta = n$ large blocks of size $\beta \cdot s$ each.

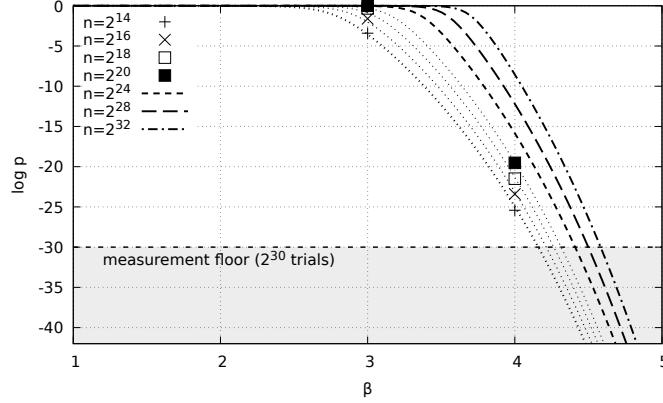


Fig. 2: Maximum β for power-of-two-choices hashing to achieve failure probability p

For hash table chaining, we can easily reduce the total number of blocks for the underlying ORAM. Every operation always accesses all β ORAM blocks $[i \cdot \beta, \dots, (i + 1) \cdot \beta - 1]$ in a row. So, idea is to simply increase the block size by a factor of β and perform the ORAM access operation only once. We then need to only perform one access to an ORAM of $\eta = n$ large blocks. Placing v into the right block (insert) or finding k in a large block (lookup) is done using standard techniques for double obliviousness.

The same big block optimization as with hash table chaining also applies for POTC. Instead of β ORAM blocks per bucket and a total of $\eta = n \cdot \beta$ blocks, we can set the block size to β leading to a smaller ORAM of $\eta = n$ blocks.

4 Hash Table Parameter Estimation

Roadmap: We estimate concrete hash-table parameters, tune Path ORAM’s position map, and benchmark the resulting dictionaries against an AVL-tree ODS in SGX.

4.1 Hash Table Parameters

Besides hash table chaining, concrete hash table parameters to achieve a failure probability of $2^{-\sigma}$ during insert operations are neither known nor easy to obtain analytically. Moreover, a failure event with probability 2^{-40} is, by construction, unobservable at any feasible experiment count, as witnessing it would require on the order of 2^{40} trials. Consequently, extrapolating from measurements and experiments is a standard tool for concrete security estimation at cryptographic targets. We follow this established methodology and run a large number of experiments for different parameter settings and then extrapolate using regression.

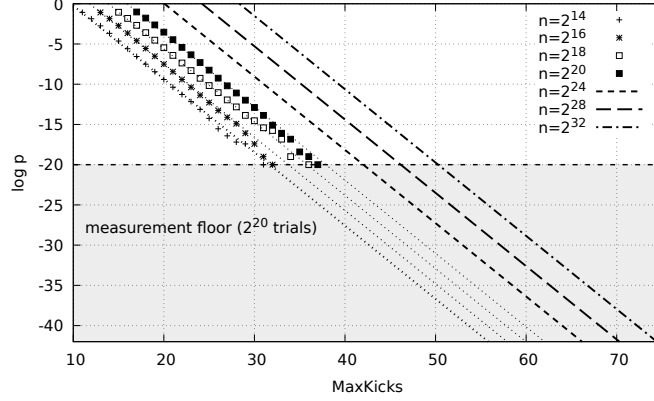


Fig. 3: Maximum number of kicks for stash-less cuckoo hashing ($\epsilon = 1.27$, $u = 3$) to achieve failure probability p

For each scheme and $n = 2^x$, $8 \leq x \leq 20$, an experiment initializes an empty table, inserts the n keys in $[n]$, and records the maximum bucket load, number of evictions, or queue occupancy. We repeat the experiment 2^R times, with $R = 30$ for POTC and $R = 20$ for SLCH and DEAM. For a candidate parameter, the fraction of experiments whose recorded maximum exceeds it estimates the failure probability. The plots mark parameter values with no observed failure at the nominal measurement floor 2^{-R} ; the rule-of-three bound below quantifies the corresponding statistical confidence.

We then make a *log-linear regression* argument, depending on the asymptotic relationship between parameter and failure probability, to extrapolate and scale the parameters so they achieve a failure probability of 2^{-40} . Essentially, we extrapolate for each n the parameters that would be required to reach a 2^{-40} failure probability and also extrapolate for huge values of n that we cannot benchmark due to hardware limitations, e.g., $n = 2^{32}$.

Discussion: Capacity-Planning. We target a per-build failure probability $p \leq 2^{-\sigma}$, $\sigma = 40$, as *capacity-planning* bound, not a security parameter. Under oblivious padding, parameters β , MaxKicks, and ℓ are fixed once at init and cannot be resized or probed adaptively during the data structure’s lifetime. An under-provisioned parameter is therefore an unrecoverable overflow, forcing a rebuild with fresh hash keys (Appendix E.1) at cost $\Theta(n)$, while an over-provisioned one is paid on every operation. Parameters must be chosen tight and once. Asymptotic bounds are insufficient, the concrete constants are required. Failure is thus a liveness, not a security, event. As discussed in Appendix E.1, a rebuild is data-independent and leaks nothing, so the bound optimizes cost, not obliviousness.

We adopt an empirical method widely used for concrete parameters across applied cryptography, e.g., the Path ORAM stash-size bound [14, 47], the cuckoo-

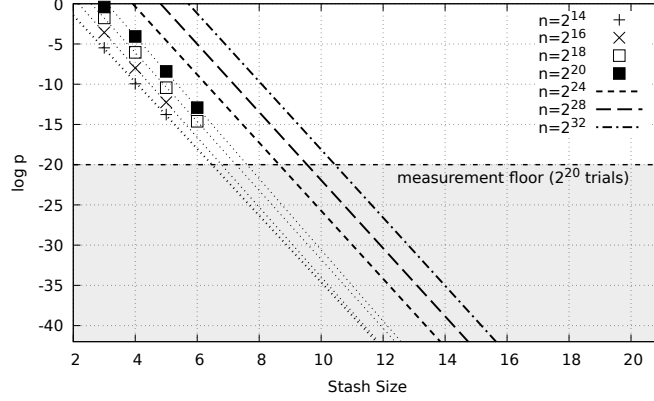


Fig. 4: Maximum queue size for de-amortized cuckoo hashing to achieve failure probability p , $\epsilon = 1.2$, $L = 3$

hashing overflow probability in Private Set Intersection [13, 21, 40, 41], cryptographic key-size selection [31] or the concrete security of the LWE-based NIST PQC finalists [2, 3]. We stress that we only reuse the method, a union-bound-derived probability, regression, a rule-of-three floor, but not its rationale. In the other works, a failure breaks correctness or security while it only costs an amortized rebuild under fixed padding in this paper.

Additionally, we verify the quality of our extrapolation. For each scheme, we refit the extrapolation on smaller inputs ($n \leq 2^{16}$) and use it to predict the failure probabilities for larger inputs ($n > 2^{16}$) that were not included in the fit, but for which we have the failure probabilities.

While more details are in Table 3 in Appendix D, we summarize that the extrapolation models for smaller inputs $n \leq 2^{16}$ predict the failure probabilities for larger n to within 0.04 (POTC), 0.35 (DEAM), and 0.53 (SLCH) parameter units Root-mean-Squared error. The fitted form thus extrapolates accurately across the 4 binary orders of n it was not fitted on. The extrapolation is moreover over-determined: discarding three quarters of the data (all n larger than 2^{16}) does not shift the parameter at $n = 2^{32}$ and $p = 2^{-40}$ for POTC. For SLCH, it shifts by $\Delta_{2^{32}} = 1.25$, and for DEAM it shifts by $\Delta_{2^{32}} = 0.64$. This is at or below the integer rounding we apply. So, our extrapolation is insensitive to which measurements it is fit and thus robust for the tail we are interested in.

For the parameters we use in our benchmarks, we can also complement the concrete extrapolation by a rigorous, assumption-free bound. The failure probability is monotonically decreasing in each parameter. At the smallest parameter value where no failure occurs across all 2^R experiments, the “rule of three” [24] bounds $p \leq 3 \cdot 2^{-R}$ at 95% confidence. Due to monotonicity, this also bounds every larger value we benchmark. This gives $p < 2^{-28.4}$ (POTC) and $p < 2^{-18.4}$ (SLCH, DEAM) for every simulated n . The regression is thus needed only to extend this bound from $3 \cdot 2^{-R}$ to 2^{-40} and to values of n we cannot simulate.

Note that the asymptotics we fit in the sections below derive from union bounds over the n buckets (respectively insert operations), a deliberately loose over-count of failure events. Such extrapolation ensures failure probabilities no smaller than the truth. Finally, rounding every parameter up to the next integer then adds a further, explicit margin on top. The concrete estimate is thus conservative in both the model it fits and the rounding it applies.

4.2 CHAIN

The failure probability p , i.e., the probability that there is at least one bucket out of $m = n$ buckets with a chain longer than $\beta \in O(\frac{\log n}{\log \log n})$ is in $o(1)$. In contrast to the other hashing techniques in this paper, a concrete parameter analysis exists for chaining. As shown before [40], p is upper bounded by

$$p \leq n \cdot \left(\sum_{i=\beta}^n \binom{n}{i} \cdot \left(\frac{1}{n}\right)^i \cdot \left(1 - \frac{1}{n}\right)^{n-i} \right) \quad (1)$$

which allows numeric derivation of β for different n and a target probability of $p \leq 2^{-40}$. For example, with $n = 2^{20}$ key-value pairs, $\beta = 19$ with probability $p \geq 1 - 2^{-40}$ from Inequality 1. So, the length of the largest chain of any bucket in the hash table exceeds $\beta = 19$ with probability $\leq 2^{-40}$.

For our evaluation, we use values for β as the maximum chain length and ORAM block size exactly as established by Pinkas et al. [40].

4.3 POTC

For the Power of Two Choices Hashing (POTC), the only hash table parameter we have to estimate is β , the maximum size of each bucket and also the size of each ORAM block.

With $m = n$ buckets and $R = 30$, we observe overflows only for $\beta \leq 4$. For $\beta = 5$, no experiment fails for any n , placing it at the nominal 2^{-30} measurement floor. This agrees with the experimental evaluation in Table 7.1 of Azar et al. [9].

Recall that the POTC failure probability decreases doubly exponentially in β . A union bound over the n buckets gives $p \in n \cdot 2^{-\Omega(2^\beta)}$, so $\log p \leq \log n - \Omega(2^\beta)$ in the tail. We will estimate the constants hidden in this asymptotic decay by regression, as detailed in the following.

The above bound holds only as long as $\beta \leq \log \log n + O(1)$ (Theorem 3.2 of [9]), and the constant inside $O(1)$ is unknown. The value $\beta = 5$ lies at exactly this scale, since $\log \log n \leq 5$ for all $n \leq 2^{32}$, consistent with the maximum load never exceeding 5 in any of our 2^{30} experiments. As we cannot certify formally that $\beta = 5$ falls within $\log \log n + O(1)$, we instead treat the double exponential decay as an empirical model whose form we confirm on the measurable points $\beta \leq 4$ and calibrate by regression. This is pessimistic, because the union bound over the n buckets over-counts failures, so the fitted p upper-bounds the true probability. The extrapolation is relied on only to reach 2^{-40} and $n > 2^{20}$.

Specifically, to extrapolate to target $p = 2^{-40}$ and values of n beyond 2^{20} , we fit the free parameters a , b , and c of relation

$$\log p = a \cdot \log n - b \cdot 2^\beta + c \quad (2)$$

to the measured $\log p$. Unlike SLCH and DEAM below, the double exponential decay is so steep that, for each n , at most two values of β fall in the measurable band (below $p \approx 1$, above the 2^{-30} floor), and these shift with n . For example, $\beta \in \{3, 4\}$ at $n = 2^{14}$, but only $\beta = 4$ at $n = 2^{20}$, as $\beta = 3$ makes $p \approx 1$ there. Only $\beta = 3$ and $\beta = 4$ yield tail values with $p \ll 1$ which leaves the 2^β decay of a , b , and c poorly constrained from the tail alone. We therefore also include the head points where $p \approx 1$, i.e., $\beta = 2$ for $n \leq 2^{11}$ and $\beta = 3$ for larger n . However, now the offset $a \cdot \log n + c$ of Equation 2 would push $\log p$ above 0, i.e., $p > 1$. We thus cap the fitted $\log p$ at 0. To compute $\min(0, a \cdot \log n - b \cdot 2^\beta + c)$, we fit its smooth counterpart

$$\log p = -\log(1 + 2^{-(a \cdot \log n - b \cdot 2^\beta + c)}), \quad (3)$$

the logarithm of the logistic (sigmoid) function $\frac{1}{1+2^{-x}}$. Equation 3 matches Equation 2 in the tail where β is large ($p \ll 1$), but bends smoothly to $\log p = 0$ as p goes to 1.

We stress that we apply the sigmoid only to flatten the head (keep $p \leq 1$) so that these points can enter the fit and constrain a , b , and c , which the two tail values $\beta = 3, 4$ alone leave poorly determined. In the tail $p \ll 1$, the sigmoid becomes Equation 2, so it does not affect the extrapolation. To confirm this, we have refitted the plain tail relation of Equation 2 to only the tail points where $p < \frac{1}{2}$. This fit shifts the extrapolated β at $p = 2^{-40}$ by less than 0.01 for all n up to 2^{32} , far below our integer rounding, and still yields $\beta = 5$.

Figure 2 shows the measured points for n between 2^{14} and 2^{20} together with the resulting fitted curves also for larger values of n . We stress that the curves are only to guide the eye, and β must be rounded to the next largest integer which we do for our benchmarks in Section 6. There, we set $\beta = 5$ for all n .

4.4 SLCH

We set $\epsilon = 1.27$ and $u = 3$ for Stash-less Cuckoo hashing (SLCH) and only need to find concrete values for **MaxKicks**, i.e., the maximum number of evictions. We use $R = 20$ because an SLCH experiment may perform many more evictions than a POTC experiment.

As $p \in n \cdot 2^{-\Omega(\text{MaxKicks})}$, we have $\log p \leq \log n - \Omega(\text{MaxKicks})$ in the tail. We thus adopt a log-linear model and, in contrast to POTC, directly fit parameters a , b , and c of

$$\log p = a \cdot \log n + b \cdot \text{MaxKicks} + c$$

to the measured $\log p$ by (linear) regression, restricted to the tail ($p < \frac{1}{2}$). This direct fitting is possible, because the single exponential decay populates the tail densely with measurable points, so we can directly fit the tail and ignore the

head. Figure 3 shows the measured points for n between 2^{14} and 2^{20} together with the resulting fitted curves, also for values of $n > 2^{20}$ and down to target probability $p = 2^{-40}$. The measured points fall closely onto the fitted curves, confirming the log-linear relationship expected from $p \in n \cdot 2^{-\Omega(\text{MaxKicks})}$.

As for POTC, we round MaxKicks up to the next integer in our benchmarks in Section 6.

4.5 DEAM

For de-amortized cuckoo hashing (DEAM), we fix the table expansion and insertion work parameters to match the construction used by Arbitman et al. [6]: two tables $\mathcal{T}_0, \mathcal{T}_1$, each with $m = 1.2 \cdot n$ buckets, and $L = 3$ cuckoo moves performed per insert. The remaining concrete parameter is the capacity ℓ of the auxiliary queue. This queue stores both ordinary de-amortization backlog and elements that are temporarily deferred after the cycle-detection mechanism identifies a second cycle. In our implementation, the queue is stored inside the enclave and scanned double-obliviously up to its fixed capacity ℓ , so increasing ℓ affects local enclave work but does not increase the number of external Path ORAM accesses, unlike our other schemes. However, concrete parameter estimation for ℓ is independently interesting as there are other applications where the size of the queue may be more impactful.

The asymptotic analysis of Arbitman et al. [6] shows that, for suitable constant L , the auxiliary data structure has size $O(\log n)$ with overwhelming probability over the initialization randomness. More concretely, their proof separates the queue size into two sources: ordinary backlog from long cuckoo insertion paths, controlled by a connected-component bound over batches of $\log n$ insertions, and elements deferred by second cycles, controlled by the stash bound of Kirsch et al. [27]. This implies logarithmic queue capacity asymptotically, but it does not give useful concrete constants for our parameter setting $L = 3$. We therefore estimate ℓ empirically.

We use $R = 20$ because DEAM, like SLCH, is substantially more expensive to simulate than POTC.

Arbitman et al. [6] do not establish a concrete tail bound on the total queue size. However, its two contributions each decay exponentially in ℓ . Regular backlog is governed by sums of connected-component sizes in the cuckoo graph, whose tails decay as $2^{-\Omega(\ell)}$ for $\epsilon > 1$. Second cycles are governed by stash bound $O(n^{-\ell})$ [27]. Both yield $\log p$ linear in ℓ , resulting in a log-linear model, which fits the measured queue-occupancy tail closely. To get the concrete parameters, we fit the parameters a, b, c in

$$\log p = a \cdot \log n + b \cdot \ell + c$$

to the measured failure probabilities, restricting the regression to points with $p < \frac{1}{2}$. As with SLCH, we fit only this densely populated tail and ignore the head, where it would likewise exceed $p = 1$. So, again no saturation term as in POTC is needed. Since $b < 0$, the fitted model gives larger queue capacities for

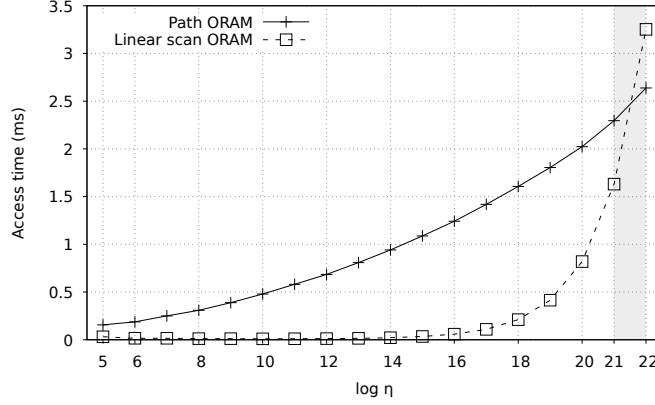


Fig. 5: Access time of linear-scan ORAM vs. Path ORAM position-map update. The shaded region marks where the cost crosses over, between an 8 MiB and 16 MiB position map.

larger tables and for smaller target failure probabilities. We then extrapolate the fitted curves to table sizes larger than those directly simulated, including $n = 2^{24}$, $n = 2^{28}$, and $n = 2^{32}$, and choose the smallest integer ℓ such that the fitted failure probability is at most 2^{-40} .

Figure 4 shows the measured failure probabilities and the fitted extrapolations. The measured data fall close to straight lines in the tail, consistent with the fitted log-linear model. For the benchmark parameters in Section 6, we round the resulting capacity upward to the next integer.

5 Path ORAM Position Map

Inside an SGX enclave, the residual position map must be linearly scanned on every access to prevent cache leakage. Larger residual maps reduce recursion but increase local computation, creating a trade-off between these costs.

We therefore cap the residual map at T MByte. If $\eta_i \log \eta_i$ Bits exceeds T , we store it in a subsequent position-map ORAM with $\eta_{i+1} = \eta_i/c$ blocks of size $B_{i+1} = c \log \eta_{i+1}$ Bit, repeating until the residual map fits.

To choose the optimal value of T , we have determined the threshold where, in our SGX evaluation environment (see Section 6.2 for the specification), a linear scan of the position map becomes more expensive than an extra Path ORAM access. Figure 5 shows the empirical results of this comparison, run inside an SGX enclave. The crossover happens between $\log \eta = 21$ and 22, corresponding to a position map between 8 and 16 MiB.

Since we are comparing the efficiency of storing the entire position map (an associative array) in memory and doing a linear scan over it for each access against storing the same array in an ORAM, this figure also shows the exact point

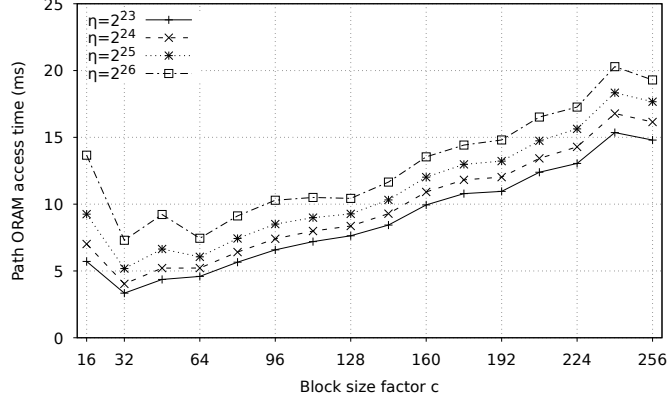


Fig. 6: Path ORAM access time as function of block size factor c . Each series fixes the position-map recursion depth at one level. Access time is the mean of the measured insert and lookup averages.

at which it becomes advantageous to use an oblivious data structure algorithm in the first place compared to just storing the database in enclave memory.

The block size $B_i = c \cdot \log \eta_i$ also has to be tuned for the SGX environment. Our implementation fixes position map addresses to a fixed length of $\log \eta_i = 32$ Bit for all levels i of the recursion. Now, initially, it might seem that $c = 2$ is the best option for setting the block size. Increasing it from 2 to 4, for instance, makes the resulting ORAM tree one level shorter, but causes twice as much data to be read for each access. However, there is a static cost to doing an OCALL in SGX and reading a chunk of data from the disk. Increasing the amount read, up to a point, does not significantly increase the total cost of the read operation, but it does reduce the total number of external reads done slightly. Therefore, the optimal value should be larger than 2.

To determine it more precisely, we have performed benchmarks sweeping block sizes B from $c = 2$ times 32 Bit to $c = 256$ times 32 Bit in increments of 16. Figure 6 shows the results for $c \in \{16, \dots, 256\}$ as a representative sample. As noted in Section 2.2, c must be a powers of two, so the only viable settings are 16, 32, 64, 128, and 256. Among these, 32 is a clear minimum, representing an optimal tradeoff between external access cost and Path ORAM tree height.

As a result of these micro benchmarks, we standardize on $c = 32$ and a position-map threshold of $T = 10$ MiB. For comparison, we also evaluate the same $c = 32$ configuration with a constrained threshold of $T = 1$ MiB, representing a minimally-sized enclave, e.g., for the case the host computer wants to run several in parallel or alongside other enclaves.

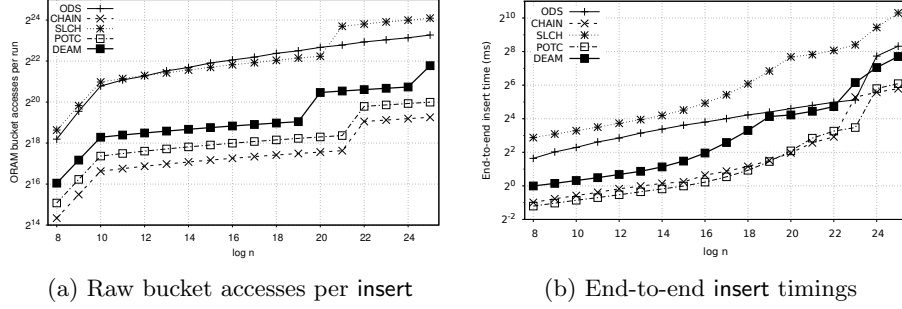


Fig. 7: Benchmarks for $T = 10$ MiB, $c = 32$

6 Data Structure Benchmarks

We now turn to the main benchmarks which compare the different oblivious data structures, oblivious AVL trees vs. our oblivious, probabilistic hash tables.

6.1 Storage Overhead

Let $|kv|$ denote the Bit length of a key-value pair. The AVL-tree ODS uses $2n - 1$ Path ORAM nodes with $Z = 4$ blocks of size $|kv|$, for storage overhead $\frac{8n-4}{n} \approx 8\times$. CHAIN instead uses n blocks of size $\beta|kv|$ after the optimization in Section 3.6, giving overhead $\frac{\beta(8n-4)}{n} \times$. Figure 8 in the appendix includes all schemes and recursive position maps. All hash tables except SLCH require more storage than the AVL tree; CHAIN is the most expensive at approximately $2.5\times$ the tree’s storage. Its steps occur when β or the number of recursion levels increases.

6.2 Access Benchmarks

To empirically compare these data structures, we ran experiments on an Alibaba Cloud Elastic Cloud Service instance of type `ecs.g7t.xlarge`. This instance uses a 3rd generation Xeon Ice Lake processor with SGX, 16 GiB of RAM, and runs Ubuntu 24.04 [4]. The size of the EPC was pre-configured to 8 GiB. That is, as soon as the enclave would require more than 8 GiB of RAM, random memory accesses would become slower. However, our benchmark enclaves, including space for T to store a reduced position map, were in the order of only a few dozen MiB. The first comparison, shown in Figure 7a, shows the number of underlying Path ORAM accesses that are incurred per individual `insert` call on the oblivious data structure. This allows for an objective comparison between data structures that abstracts away all hardware and operating system behavior. As external memory accesses are the main bottleneck with SGX-implemented oblivious data structures, one would expect this to tightly correlate with the actual *end-to-end*

runtimes. Indeed, the end-to-end insert runtimes in Figure 7b follow the number of raw bucket accesses closely.

Appendix C shows these empirical end-to-end runtimes in detail. There, Table 2 compares data structures timings for `lookup` and `insert` operations for various values of n and different values of T (1 MiB and 10 MiB). Observe that in Figure 7, we present benchmarks for $T = 10$ MiB, but Figure 9 in the appendix also present benchmarks for $T = 1$ MiB.

6.3 Analysis

Our end-to-end timing results show SLCH having notably the worst insert timings of the data structures we compare, being *slower* than ODS for medium to large database sizes. DEAM outperforms ODS for all database sizes except 2^{23} .

All schemes have a noticeable jump in runtime at either 2^{23} or 2^{24} . This is the point at which the ORAM database can no longer fit in the page cache of our machine and significant disk accesses start to occur. There is a discontinuity here because both deamortized cuckoo hashing and chaining have larger ORAM trees than the other schemes, because the number of buckets is multiplied by a larger constant. They therefore end up reaching this threshold at 2^{23} while the remaining schemes reach it at 2^{24} .

At the highest database sizes, where the performance is more impactful, CHAIN and POTC both have similar performance and a significant lead over the other schemes and ODS. This is due to their very simple algorithmic complexity requiring only a small number of bucket accesses for each `lookup` or `insert` operation.

Between these two schemes, POTC has a slight advantage in storage requirement while CHAIN has a slight advantage in runtime. Both significantly outperform ODS by, approximately, $6\times$ and $10\times$ for `lookup` and $5\times$ and $6\times$ for `insert`.

7 Related Work

Oblivious data structures over ORAM. Wang et al. [50] introduce oblivious data structures and show that a pointer-based AVL tree can be layered over a non-recursive Path ORAM. This AVL-tree ODS is the design realized inside TEEs by Oblix [36], OblIDB [16], and EnigMap [49]. Roche et al. [44] give an oblivious map with history independence and secure deletion. Concurrent oblivious data structures for high throughput have recently been investigated [25, 48]. All of these realize the dictionary interface through a tree with $\Theta(\log n)$ ORAM accesses per operation.

Oblivious hashing. Oblivious variants of hashing are a well-studied primitive used as building block inside the hierarchical ORAM of Goldreich and Ostrovsky [22]. Together with oblivious sorting, their goal is to asymptotically improve the amortized access cost [8, 12, 23, 39] of hierarchical ORAM. Their focus is on

hash tables built statically on a fixed set of keys, but target asymptotic performance of the enclosing ORAM. Cuckoo hashing is also central to the private set intersection (PSI) literature [21, 40, 41]. Oblivious key-value stores (OKVS) [21] formalize the encode-once/decode interface used there. They again target a static key set, a non-oblivious-access setting, and without our TEE padding constraint. Our contribution to the above is orthogonal and complementary. We take hash table constructions as given, instantiate them as double-oblivious dictionaries over Path ORAM, and show empirically that they dominate the AVL-tree ODS.

Alternative data structures. Filters such as Bloom or cuckoo filters [17] answer membership queries obviously and compactly, but do not return an associated value and admit false positives. They do not realize the key-value interface required in this paper, so we do not consider them further.

Open Science

Availability. Our artifact is available from the anonymized repository [5]. It can be browsed online or downloaded without credentials.

Contents. The repository contains the SGX host application and enclave implementations of CHAIN, POTC, SLCH, DEAM, the AVL-tree ODS baseline, and Path ORAM. It also includes the CPU and CUDA simulations used to estimate hash-table parameters; raw and summarized measurements for the data-structure, position-map, and ORAM experiments; and the analysis and plotting scripts used to reproduce the reported results.

Evaluation. The top-level README documents prerequisites, builds, the executable interface, and commands for both quick validation and measurement reproduction. Full end-to-end measurements require an x86-64 Linux system with the Intel SGX SDK, runtime, and an SGX-capable CPU. The checked-in measurements can be parsed and the parameter-estimation plots regenerated without rerunning the hardware experiments.

Bibliography

- [1] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [2] M. R. Albrecht. Security Estimates for Lattice Problems, 2026. <https://github.com/malb/lattice-estimator>.
- [3] M. R. Albrecht, B. R. Curtis, A. Deo, A. Davidson, R. Player, E. W. Postlethwaite, F. Virdia, and T. Wunderer. Estimate all the LWE, NTRU schemes! In *Security and Cryptography for Networks: 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, page 351–367, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-319-98112-3. https://doi.org/10.1007/978-3-319-98113-0_19. URL https://doi.org/10.1007/978-3-319-98113-0_19.
- [4] Alibaba. Features and specifications of ecs instance families, 2026. <https://www.alibabacloud.com/help/en/ecs/user-guide/overview-of-instance-families>.
- [5] Anonymous. Oblivious data structures artifact. <https://anonymous.4open.science/r/oblivious-data-structures-artifact-4326/>, 2026. Anonymous artifact repository.
- [6] Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikolettseas, and W. Thomas, editors, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, Lecture Notes in Computer Science, pages 107–118. Springer, 2009.
- [7] S. Arnaudov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’keeffe, M. L. Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 689–703, 2016.
- [8] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi. OptORAMa: Optimal oblivious RAM. In *Advances in Cryptology - EURO-CRYPT 2020*, volume 12106 of *Lecture Notes in Computer Science*, pages 403–432. Springer, 2020.
- [9] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.
- [10] M. Bellare. New proofs for NMAC and HMAC: security without collision resistance. *J. Cryptol.*, 28(4):844–878, 2015.
- [11] D. Cash, A. Drucker, and A. Hoover. A lower bound for one-round oblivious ram. In *Theory of Cryptography Conference*, pages 457–485. Springer, 2020.
- [12] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Ad-*

- vances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 660–690. Springer, 2017.
- [13] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1243–1255. ACM, 2017.
 - [14] G. Connell. Technology deep dive: Building a faster oram layer for enclaves, 2022. URL <https://signal.org/blog/building-faster-oram/>.
 - [15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 1990.
 - [16] S. Eskandarian and M. Zaharia. Oblidb: oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, Oct. 2019. ISSN 2150-8097.
 - [17] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.
 - [18] M. Farach-Colton, A. Krapivin, and W. Kuszmaul. Optimal bounds for open addressing without reordering. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 594–605. IEEE, 2024.
 - [19] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2): 229–248, 2005.
 - [20] A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random-walk cuckoo hashing. In I. Dinur, K. Jansen, J. Naor, and J. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 490–503, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
 - [21] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. Oblivious key-value stores and amplification for private set intersection. In T. Malkin and C. Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 395–425. Springer, 2021.
 - [22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996. ISSN 0004-5411. <https://doi.org/10.1145/233551.233553>. URL <https://doi.org/10.1145/233551.233553>.
 - [23] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of out-sourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, *Lecture Notes in Computer Science*, pages 576–587. Springer, 2011.

- [24] J. A. Hanley and A. Lippman-Hand. If nothing goes wrong, is everything all right?: Interpreting zero numerators. *JAMA*, 249(13):1743–1745, 04 1983. ISSN 0098-7484.
- [25] W. He, F. Wang, and D. Feng. H2oram: Low response latency optimized oram for hybrid memory systems. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 405–408. IEEE, 2020.
- [26] Intel. Intel Software Guard Extensions, 2026. <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html>.
- [27] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. In D. Halperin and K. Mehlhorn, editors, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, Lecture Notes in Computer Science, pages 611–622. Springer, 2008.
- [28] D. E. Knuth. *The art of computer programming: Seminumerical algorithms, volume 2*. Addison-Wesley Professional, 2014.
- [29] K. G. Larsen and J. B. Nielsen. Yes, there is an oblivious ram lower bound! In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*, page 523–542, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-319-96880-3. https://doi.org/10.1007/978-3-319-96881-0_18. URL https://doi.org/10.1007/978-3-319-96881-0_18.
- [30] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017.
- [31] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *J. Cryptol.*, 14(4):255–293, 2001.
- [32] Y. Luo and M. Gao. obliv-clang: Real-world oblivious programming in c++, 2026. <https://github.com/tsinghua-ideal/obliv-clang>, <https://arxiv.org/abs/2606.16187>.
- [33] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS ’13*, page 311–324, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. <https://doi.org/10.1145/2508859.2516692>. URL <https://doi.org/10.1145/2508859.2516692>.
- [34] M. Marlinspike. The Difficulty Of Private Contact Discovery , 2014. <https://signal.org/blog/contact-discovery/>.
- [35] M. Marlinspike. Technology preview: Private contact discovery for signal, 2017. <https://signal.org/blog/private-contact-discovery/>.
- [36] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and*

- Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 279–296. IEEE Computer Society, 2018.
- [37] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, Lecture Notes in Computer Science, pages 121–133. Springer, 2001.
 - [38] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2): 122–144, 2004.
 - [39] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 871–882. IEEE Computer Society, 2018.
 - [40] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), Jan. 2018. ISSN 2471-2566. <https://doi.org/10.1145/3154794>. URL <https://doi.org/10.1145/3154794>.
 - [41] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient circuit-based PSI with linear communication. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.
 - [42] M. Raab and A. Steger. "balls into bins" - A simple and tight analysis. In M. Luby, J. D. P. Rolim, and M. J. Serna, editors, *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, Lecture Notes in Computer Science, pages 159–170. Springer, 1998.
 - [43] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious {RAM}. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.
 - [44] D. S. Roche, A. J. Aviv, and S. G. Choi. A practical oblivious map data structure with secure deletion and history independence. In *IEEE Symposium on Security and Privacy, SP 2016*, pages 178–197. IEEE Computer Society, 2016.
 - [45] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotracer : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
 - [46] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In X. Chen, X. Wang, and X. Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 317–328. ACM, 2016.

- [47] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, page 299–310, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. <https://doi.org/10.1145/2508859.2516660>. URL <https://doi.org/10.1145/2508859.2516660>.
- [48] N. Talur and I. Demertzis. SONIC: Concurrent oblivious RAM & data structures for low-latency and high-throughput. Cryptology ePrint Archive, Paper 2026/1577, 2026. URL <https://eprint.iacr.org/2026/1577>.
- [49] A. Tinoco, S. Gao, and E. Shi. EnigMap: External-Memory oblivious map for secure enclaves. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4033–4050, Anaheim, CA, Aug. 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/tinoco>.
- [50] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.
- [51] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015.
- [52] S. Zahur and D. Evans. Obliv-c: A language for extensible data-oblivious computation, 2015. <https://github.com/samee/obliv-c>, <https://eprint.iacr.org/2015/1153.pdf>.

Table 1: Asymptotic complexities of different oblivious data structures when running on top of Path ORAM. n : number of key-value pairs data structure must hold, **insert** and **lookup**: number of external memory access operations per call (with recursive position map, see text). ODS: Oblivious Data Structure/AVL tree, CHAIN: hash table chaining, SLCH: stash-less Cuckoo hashing, POTC: power of two choices hashing, DEAM: de-amortized Cuckoo hashing. *: using the optimization of Section 3.6.

	ODS	CHAIN	CHAIN*	SLCH	POTC	POTC*	DEAM
Communication insert	$O(\log^2 n)$	$O(\frac{\log^3 n}{\log \log n})$	$O(\log^2 n)$	$O(\log^3 n)$	$O(\log^2 n \cdot \log \log n)$	$O(\log^2 n)$	$O(\log^2 n)$
blowup lookup	$O(\log^2 n)$	$O(\frac{\log^3 n}{\log \log n})$	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log^2 n \cdot \log \log n)$	$O(\log^2 n)$	$O(\log^2 n)$
Storage expansion	$O(1)$	$O(\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\log \log n})$	$O(1)$	$O(\log \log n)$	$O(\log \log n)$	$O(1)$

A Probabilistic Hash Table Algorithms

This appendix presents the complete pseudo-code for the four probabilistic hash tables described in Section 3.

Algorithm 1 Oblivious CHAIN insert and lookup

```

1: Parameters: Table  $\mathcal{T}$  with  $m = n$  buckets, every bucket  $\mathcal{T}[i]$  holds  $\beta$  key-value
   pairs  $\mathcal{T}[i][j]$  (initially empty), hash function  $H : \{0, 1\}^* \rightarrow \{0, \dots, m - 1\}$ 
2: function insert( $k, v$ )
3:    $i = H(k)$ ;  $placed = false$ ;
4:   for  $j = 0$  to  $\beta - 1$  do
5:     if  $\mathcal{T}[i][j] = (\perp, \perp)$  and  $placed = false$  then
6:        $\mathcal{T}[i][j] = (k, v)$ ;  $placed = true$ ;
7:   if  $placed = false$  then
8:     failure ▷ bucket overflow
9: function lookup( $k$ )
10:   $i = H(k)$ ;
11:   $v' = \perp$ ;
12:  for  $j = 0$  to  $\beta - 1$  do
13:    if  $\mathcal{T}[i][j] = (k, v)$  then ▷ Key matches
14:       $v' = v$ ; ▷  $v$  can be  $\perp$ 
15:  return  $v'$ ;

```

B Asymptotic Performance

Table 1 summarizes the resulting asymptotic performance of oblivious key-value storage realized by probabilistic hash tables on top of Path ORAM. As Table 1 presents asymptotic complexities, this includes a multiplicative factor of $O(\log n)$ for recursion. While oblivious hash tables are asymptotically inferior to trees, for many realistic values of n a factor of $O(\log n)$ can be replaced by a small

Algorithm 2 Oblivious POTC insert and lookup

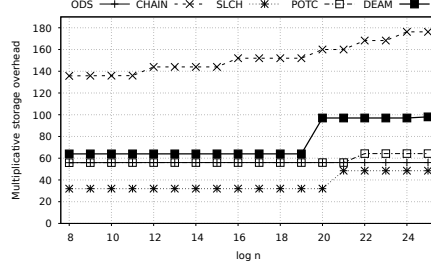
1: **Parameters:** Table $\mathcal{T}$ with $m = n$ buckets, every bucket holds β key-value pairs (initially empty), hash functions $H_0, H_1 : \{0, 1\}^* \rightarrow \{0, \dots, m-1\}$
2: **function** insert(k, v)
3: $i_0 = H_0(k); \quad i_1 = H_1(k); \quad placed = false;$
4: **for** $j = 0$ **to** $\beta - 1$ **do**
5: **if** $placed = false$ **then**
6: **if** $\mathcal{T}[i_0][j] = (\perp, \perp)$ **then**
7: $\mathcal{T}[i_0][j] = (k, v); \quad placed = true;$
8: **else if** $\mathcal{T}[i_1][j] = (\perp, \perp)$ **then**
9: $\mathcal{T}[i_1][j] = (k, v); \quad placed = true;$
10: **if** $placed = false$ **then**
11: failure ▷ bucket overflow
12: **function** lookup(k)
13: $i_0 = H_0(k); i_1 = H_1(k);$
14: $v' = \perp;$
15: **for** $j = 0$ **to** $\beta - 1$ **do**
16: **if** $\mathcal{T}[i_0][j] = (k, v)$ **then** ▷ Key matches
17: $v' = v;$
18: **else if** $\mathcal{T}[i_1][j] = (k, v)$ **then** ▷ Key matches
19: $v' = v;$
20: return $v';$ ▷ v can be $\perp$

Algorithm 3 Oblivious SLCH insert and lookup

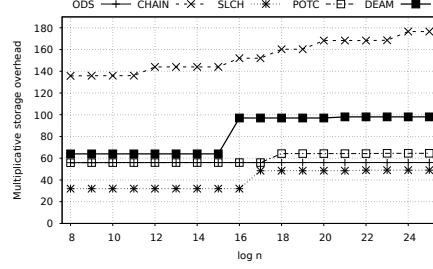
1: **Parameters:** Table $\mathcal{T}$ with $m = 1.27 \cdot n$ buckets, each (initially empty) bucket holds one key-value pair and hash function index ($key, value, idx$), hash functions $H_0, H_1, H_2 : \{0, 1\}^* \rightarrow \{0, \dots, m-1\}$, maximum number of kicks MaxKicks.
2: **function** insert(k, v)
3: $carry = (key, value, idx) = (k, v, 0);$
4: **for** $j = 0$ **to** MaxKicks $- 1$ **do**
5: **if** $carry \neq (\perp, \perp, \perp)$ **then**
6: $i = H_{carry.idx}(carry.key);$
7: swap($carry, \mathcal{T}[i]$; ▷ $\mathcal{T}[i]$ is either (k', v', idx') or $(\perp, \perp, \perp)$
8: $carry.idx = (carry.idx + 1) \bmod 3;$
9: **if** $carry \neq (\perp, \perp, \perp)$ **then**
10: failure ▷ bucket overflow
11: **function** lookup(k)
12: $i_0 = H_0(k); \quad i_1 = H_1(k); \quad i_2 = H_2(k); \quad v' = \perp;$
13: **if** $k = \mathcal{T}[i_0].key$ **then**
14: $v' = \mathcal{T}[i_0].value;$
15: **else if** $k = \mathcal{T}[i_1].key$ **then**
16: $v' = \mathcal{T}[i_1].value;$
17: **else if** $k = \mathcal{T}[i_2].key$ **then**
18: $v' = \mathcal{T}[i_2].value;$
19: return $v';$

Algorithm 4 Oblivious DEAM insert and lookup

```
1: Parameters: Tables  $\mathcal{T}_0$  and  $\mathcal{T}_1$  with  $m = 1.2 \cdot n$  buckets each, every (initially  
   empty) bucket holds one key-value pair and hash function index  $(key, value, idx)$ ,  
   queue (initially empty) of size  $\ell$ , hash functions  $H_0, H_1 : \{0, 1\}^* \rightarrow \{0, \dots, m-1\}$ .  
2: function insert( $k, v$ )  
3:   add2QueueBack( $k, v, 0$ ); ▷ could fail  
4:    $carry = (key, value, idx) = (\perp, \perp, \perp)$ ;  
5:   for  $i = 0$  to  $L - 1$  do  
6:     if  $carry = (\perp, \perp, \perp)$  then  
7:        $carry = \text{PopQueue}()$ ;  
8:     if  $carry \neq (\perp, \perp, \perp)$  then  
9:       if  $\mathcal{T}_{carry.idx}[H_{carry.idx}(carry.key)] = (\perp, \perp, \perp)$  then  
10:         $\mathcal{T}_{carry.idx}[H_{carry.idx}(carry.key)] = carry$ ;  
11:        ResetCDM();  $carry = (\perp, \perp, \perp)$ ;  
12:       else  
13:         if LookUpInCDM( $carry$ ) then  
14:           add2QueueBack( $carry$ ); ▷ could fail  
15:           ResetCDM();  $carry = (\perp, \perp, \perp)$ ;  
16:         else  
17:           add2CDM( $carry$ ); ▷ could fail  
18:           swap( $carry, \mathcal{T}_{carry.idx}[H_{carry.idx}(carry.key)]$ );  
19:            $carry.idx = (carry.idx + 1) \bmod 2$ ;  
20:       if  $carry \neq (\perp, \perp, \perp)$  then  
21:         add2QueueFront( $carry$ ); ▷ could fail  
22: function lookup( $k$ )  
23:    $i_0 = H_0(k)$ ;  $i_1 = H_1(k)$ ;  $v' = \perp$ ;  
24:   if  $k = \mathcal{T}_0[i_0].key$  then  
25:      $v' = \mathcal{T}_0[i_0].value$ ;  
26:   else if  $k = \mathcal{T}_1[i_1].key$  then  
27:      $v' = \mathcal{T}_1[i_1].value$ ;  
28:   else  
29:      $v' = \text{LookUpInQueue}(k)$ ; ▷  $v'$  can be  $\perp$   
30:   return  $v'$ ;
```

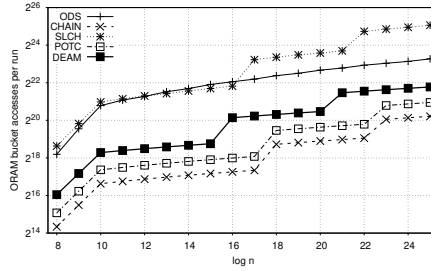


(a) $T = 10$ MiB, $c = 32$

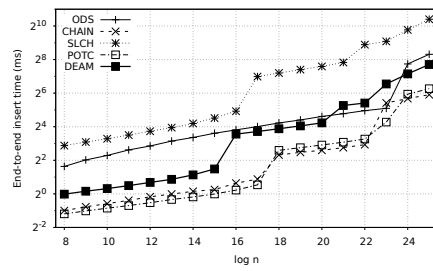


(b) $T = 1$ MiB, $c = 32$

Fig. 8: Empirical storage overhead, measured as a multiplicative factor over storing the database in a flat array.



(a) Raw bucket accesses per insert



(b) End-to-end insert timings

Fig. 9: Benchmarks for $T = 1$ MiB, $c = 32$

constant such as 2, so in practice you can shave-off one factor of $\log n$ in the hash table complexities of Table 1.

C Path ORAM Recursion and Page Cache

Table 2 shows the runtime of lookup and insert operations tested on an `ecs.g7t.xlarge` Alibaba Cloud Elastic Computer Service instance with SGX extensions [4]. We indicate where each hash table construction requires an extra recursion in the underlying Path ORAM and therefore demonstrates a discontinuous jump in runtime. Similarly, we also mark where size of the ORAM on disk becomes larger than the host computer’s page cache, resulting in another runtime increase.

D Extrapolation Quality

To check the quality of the fitted models of Section 4.1, we perform a cross-validation that leaves out measurements for the largest values of n . For each scheme, we refit the model on the smaller input sizes ($n \leq 2^{16}$) and use it to

Table 2: Benchmarks including recursion levels and page cache. A “———” line indicates an increase in position map recursion, a “=====” line indicates running out of page cache (8 GByte), a bold “=====” line indicates both.

lookup(k, v)									
n	Tinoco et al. [49] ODS	Ours							
		CHAIN	$T = 10$ MiB, $c = 32$ SLCH	POTC	DEAM	CHAIN	$T = 1$ MiB, $c = 32$ SLCH	POTC	DEAM
2^8	3.1	0.3 (12.2×)	0.4 (6.9×)	0.3 (10.6×)	0.3 (9.2×)	0.3 (12.0×)	0.4 (6.9×)	0.3 (10.6×)	0.3 (9.2×)
2^9	4.1	0.3 (13.7×)	0.5 (8.1×)	0.3 (12.4×)	0.4 (10.8×)	0.3 (13.7×)	0.5 (8.1×)	0.3 (12.2×)	0.4 (10.8×)
2^{10}	4.9	0.3 (14.4×)	0.6 (8.7×)	0.4 (13.2×)	0.4 (11.6×)	0.3 (14.4×)	0.6 (8.7×)	0.4 (13.1×)	0.4 (11.6×)
2^{11}	6.1	0.4 (15.9×)	0.6 (9.8×)	0.4 (14.9×)	0.5 (13.0×)	0.4 (15.9×)	0.6 (9.8×)	0.4 (14.8×)	0.5 (13.0×)
2^{12}	7.2	0.4 (16.1×)	0.7 (10.2×)	0.5 (15.6×)	0.5 (13.4×)	0.4 (16.2×)	0.7 (10.2×)	0.5 (15.5×)	0.5 (13.4×)
2^{13}	8.8	0.5 (17.8×)	0.8 (11.0×)	0.5 (17.1×)	0.6 (14.3×)	0.5 (17.7×)	0.8 (11.0×)	0.5 (16.7×)	0.6 (14.3×)
2^{14}	10.4	0.6 (18.6×)	0.9 (11.2×)	0.6 (17.6×)	0.7 (13.9×)	0.6 (18.7×)	0.9 (11.2×)	0.6 (17.7×)	0.7 (13.9×)
2^{15}	12.2	0.6 (20.4×)	1.1 (11.0×)	0.7 (18.4×)	1.0 (12.9×)	0.6 (20.5×)	1.1 (11.0×)	0.7 (18.4×)	1.0 (12.8×)
2^{16}	13.9	0.8 (17.8×)	1.4 (9.8×)	0.8 (17.9×)	1.3 (10.5×)	0.8 (17.8×)	1.4 (9.7×)	0.8 (17.9×)	3.9 (3.5×)
2^{17}	16.1	0.9 (17.6×)	2.0 (8.1×)	1.0 (16.7×)	2.0 (8.0×)	0.9 (17.5×)	5.9 (2.8×)	1.0 (16.7×)	4.4 (3.7×)
2^{18}	18.8	1.1 (16.9×)	3.0 (6.2×)	1.3 (14.8×)	3.3 (5.6×)	2.5 (7.5×)	6.6 (2.9×)	4.0 (4.7×)	4.9 (3.8×)
2^{19}	21.3	1.4 (14.9×)	5.0 (4.3×)	1.8 (11.7×)	5.9 (3.6×)	2.8 (7.6×)	7.3 (2.9×)	4.5 (4.7×)	5.5 (3.9×)
2^{20}	24.6	1.9 (12.7×)	8.8 (2.8×)	2.8 (8.7×)	6.3 (3.9×)	3.0 (8.1×)	8.3 (3.0×)	5.0 (4.9×)	6.2 (3.9×)
2^{21}	27.7	3.0 (9.4×)	9.4 (3.0×)	4.8 (5.8×)	7.3 (3.8×)	3.4 (8.2×)	9.4 (2.9×)	5.6 (4.9×)	12.8 (2.2×)
2^{22}	31.6	3.8 (8.3×)	10.9 (2.9×)	6.4 (5.0×)	9.1 (3.5×)	3.8 (8.3×)	19.2 (1.7×)	6.4 (4.9×)	14.1 (2.2×)
2^{23}	34.7	15.0 (2.3×)	13.4 (2.6×)	7.4 (4.7×)	24.4 (1.4×)	18.0 (1.9×)	21.5 (1.6×)	13.0 (2.7×)	27.1 (1.3×)
2^{24}	166	19.5 (8.5×)	24.3 (6.8×)	34.0 (4.9×)	37.3 (4.4×)	26.2 (6.3×)	26.8 (6.2×)	39.2 (4.2×)	37.9 (4.4×)
2^{25}	275	28.1 (9.8×)	38.4 (7.2×)	47.6 (5.8×)	56.4 (4.9×)	29.8 (9.2×)	40.3 (6.8×)	47.3 (5.8×)	53.5 (5.1×)

insert(k, v)									
n	Tinoco et al. [49] ODS	Ours							
		CHAIN	$T = 10$ MiB, $c = 32$ SLCH	POTC	DEAM	CHAIN	$T = 1$ MiB, $c = 32$ SLCH	POTC	DEAM
2^8	3.1	0.5 (6.2×)	7.3 (0.4×)	0.4 (7.2×)	1.0 (3.1×)	0.5 (6.2×)	7.3 (0.4×)	0.4 (7.1×)	1.0 (3.2×)
2^9	4.1	0.6 (6.9×)	8.5 (0.5×)	0.5 (8.3×)	1.1 (3.7×)	0.6 (6.9×)	8.5 (0.5×)	0.5 (8.2×)	1.1 (3.6×)
2^{10}	4.9	0.7 (7.3×)	9.7 (0.5×)	0.6 (8.9×)	1.2 (3.9×)	0.7 (7.3×)	9.7 (0.5×)	0.6 (8.8×)	1.2 (3.9×)
2^{11}	6.1	0.8 (8.0×)	11.3 (0.5×)	0.6 (10.0×)	1.4 (4.4×)	0.8 (8.0×)	11.3 (0.5×)	0.6 (10.0×)	1.4 (4.4×)
2^{12}	7.2	0.9 (8.1×)	13.3 (0.5×)	0.7 (10.4×)	1.6 (4.5×)	0.9 (8.1×)	13.3 (0.5×)	0.7 (10.4×)	1.6 (4.5×)
2^{13}	8.8	1.0 (8.9×)	15.4 (0.6×)	0.8 (11.3×)	1.8 (4.8×)	1.0 (8.9×)	15.4 (0.6×)	0.8 (11.2×)	1.8 (4.8×)
2^{14}	10.4	1.1 (9.4×)	18.3 (0.6×)	0.9 (11.8×)	2.2 (4.8×)	1.1 (9.4×)	18.3 (0.6×)	0.9 (11.8×)	2.2 (4.8×)
2^{15}	12.2	1.2 (10.4×)	22.9 (0.5×)	1.0 (12.3×)	2.8 (4.4×)	1.2 (10.3×)	22.9 (0.5×)	1.0 (12.3×)	2.8 (4.4×)
2^{16}	13.9	1.6 (9.0×)	30.4 (0.5×)	1.2 (11.9×)	3.9 (3.6×)	1.6 (8.9×)	30.4 (0.5×)	1.2 (11.9×)	11.8 (1.2×)
2^{17}	16.1	1.8 (8.8×)	43.0 (0.4×)	1.5 (11.1×)	6.0 (2.7×)	1.8 (8.7×)	127 (0.1×)	1.5 (11.1×)	13.1 (1.2×)
2^{18}	18.7	2.2 (8.4×)	67.5 (0.3×)	1.9 (9.8×)	9.9 (1.9×)	5.0 (3.8×)	147 (0.1×)	6.0 (3.1×)	14.7 (1.3×)
2^{19}	21.0	2.9 (7.4×)	115 (0.2×)	2.7 (7.7×)	17.5 (1.2×)	5.6 (3.8×)	169 (0.1×)	6.7 (3.1×)	16.6 (1.3×)
2^{20}	24.3	3.9 (6.2×)	206 (0.1×)	4.2 (5.7×)	18.7 (1.3×)	6.0 (4.0×)	193 (0.1×)	7.5 (3.2×)	18.8 (1.3×)
2^{21}	27.7	5.9 (4.7×)	226 (0.1×)	7.2 (3.9×)	21.8 (1.3×)	6.7 (4.1×)	227 (0.1×)	8.4 (3.3×)	38.4 (0.7×)
2^{22}	31.6	7.6 (4.2×)	268 (0.1×)	9.6 (3.3×)	26.4 (1.2×)	7.6 (4.1×)	474 (0.1×)	9.6 (3.3×)	42.5 (0.7×)
2^{23}	34.7	39.0 (0.9×)	339 (0.1×)	11.1 (3.1×)	71.2 (0.5×)	42.7 (0.8×)	543 (0.1×)	19.4 (1.8×)	93.5 (0.4×)
2^{24}	212	47.7 (4.5×)	694 (0.3×)	55.5 (3.8×)	132 (1.6×)	51.2 (4.2×)	872 (0.2×)	61.4 (3.5×)	142 (1.5×)
2^{25}	319	55.3 (5.8×)	1261 (0.3×)	68.1 (4.7×)	209 (1.5×)	60.1 (5.3×)	1353 (0.2×)	77.0 (4.1×)	209 (1.5×)

predict the failure probabilities of the larger input sizes ($n > 2^{16}$), which the fit never saw. Table 3 reports the resulting prediction errors, in units of the estimated parameters β , MaxKicks, and queue size. It also shows the difference $\Delta_{2^{32}}$ between the restricted fit of $n \leq 2^{16}$ and the full-fit when extrapolated to $n = 2^{32}$ at target $p = 2^{-40}$. Across all three schemes, the restricted fit reproduces the held-out measurements and the full-data extrapolation to within about one parameter unit, i.e., at or below the integer rounding we anyways apply for the benchmarks in Section 6. Note that $\Delta_{2^{32}}$ compares two extrapolations of the same model and thus measures the fit’s stability rather than the correctness of that model in the tail which is derived asymptotically.

Table 3: Validation of parameter extrapolation. Each extrapolation is refit on $n \leq 2^{16}$ only and used to predict the larger inputs which we have measured. RMS and Max errors are over the held-out points, in units of the estimated parameter. $\Delta_{2^{32}}$ is the absolute difference between the restricted-fit and full-fit parameter extrapolated to $n = 2^{32}$ at $p = 2^{-40}$.

Scheme	RMS error	Maximal error	$\Delta_{2^{32}}$
POTC (β)	0.04	0.12	0
DEAM (queue size)	0.35	0.98	0.64
SLCH (MaxKicks)	0.53	3.59	1.25

E Discussion

E.1 Random Hash Functions and Re-Hashing

All of the above hashing schemes require one or more randomly chosen hash functions $H_i : \{0, 1\}^* \rightarrow [m]$ from a family of hash functions. A standard technique used in practice is to model the hash function family as a family of PRFs, choose random keys k_i , and set $H_i(x) = \text{HMAC}_{K_i}(x)$, while adjusting the HMAC output to map into $[n]$ [10, 28].

In Section 4, we estimate and then use hash table parameters such that the failure probability during an `insert` operation is $\leq 2^{-40}$. Still, in the event of a failure in our scenario, the client would then need to choose new hash functions, i.e., new keys k_i and completely rebuild the whole hash table. As a consequence, the total cost of n insertions holds with probability at least $1 - 2^{-40}$. In case of a failure, the probability increases to $1 - 2^{-80}$ at the cost of a second build.

We stress that a hashing failure does *not* imply a security issue, and a re-hash is *not* a data-dependent leakage. The client chooses the k_i randomly (in the enclave), hidden from the adversary who cannot derive any information about the access pattern in the rare event (2^{-40} for the whole build) of a hashing failure. So, a failure only depends on information output by function $\mathcal{L}$, i.e., n , M , and the (public) hash table configuration parameters. Consequently, simulator `Sim` can trivially reproduce failures. It samples the identical, data-independent failure coin and emits the re-hash trace on failure. More formally, the simulator can run the same operations derived from $\mathcal{L}$ on a hash table with the same public parameters using random inputs. In case of a failure of an operation, the simulator outputs a failure, too. As the choice of the hash functions is unknown to the adversary, real and simulated trace distributions are thus identical with respect to failures.

E.2 Choice of Hash Tables

The design of efficient hashing techniques is a very active area of research with a flurry of recent results. Our first hash table algorithm is simple hashing with separate chaining. We chose this not only because it is a classic baseline, but because in the ORAM setting the cost of a scheme is highly dominated by the

number of bucket accesses, and separate chaining has only one bucket access per logical access. In exchange for this efficiency, though, it has very large storage overhead.

Among load-balancing schemes, we concentrate on power-of-two-choices hashing and Cuckoo hashing, the seminal and most thoroughly analyzed constructions. We stress, however, that other schemes exist that asymptotically outperform them in *expectation*. For instance, the recently proposed Elastic hashing and Funnel hashing [18] achieve expected worst-case insert probe complexities of $O(\log \delta^{-1})$ and $O(\log^2 \delta^{-1})$, where δ is the fraction of free buckets.

We do not consider such schemes for two reasons. First, their guarantees bound the *expected* worst-case number of probes, whereas an oblivious implementation must pad every operation tightly to the *true worst case*. For Funnel and Elastic hashing, this worst case has an additive $O(\log \log n)$ complexity from a power-of-two-choices table that absorbs overflows. Second, establishing concrete failure probabilities for these schemes requires precisely the empirical estimation we develop in this paper. We therefore leave their concrete analysis to future work. Importantly, any such advance in hash table design carries over to our ORAM-based hash table constructions.

E.3 Hiding Operation Types

Wang et al. [50] discuss in their Section E.1 different leakage functions $\mathcal{L}$ for Definition 1. The leakage in this paper comprises M , n , and the operation type, but one might envision stronger security guarantees. For example, we could hide the operation type by padding all operations for all data structures (tree, hash tables) to the more expensive operation. For all data structures discussed in this paper (AVL-trees and probabilistic hash tables), this is the **insert** operation. Furthermore, depending on the type or size of the input, a different data structure might be chosen, so one might even consider hiding which data structure is used by padding all operations to the most expensive data structure (AVL trees) [50].

E.4 Deletions

As is common in the ODS literature [50], we focus on **insert** and **lookup** operations and omit deletions for clarity of exposition. A deletion is trivial to realize in hash tables as it accesses the same locations as a **lookup**, clearing the matching element. Since every Path ORAM access rewrites the fetched block regardless, a deletion has the same access pattern and cost as a **lookup**. Note that unlike an AVL-tree deletion, which may trigger rotations, a hash table delete is significantly cheaper, not requiring a rebalance. Regarding the failure probability, it is easy to see that **inserts** are the worst case: a deletion removes the key from its bucket (and queue), monotonically decreasing every padded quantity β , **MaxKicks**, and ℓ . As the live set never exceeds n , the full build dominates, so no re-parameterization is needed.

E.5 Memory for AVL trees

It might seem that comparing an oblivious AVL tree which requires only a small, constant amount of client storage with an oblivious hash table with a larger amount of T Bit client memory available is not a direct comparison. For instance, the AVL tree could somehow also make use of extra client memory, if it were available, in order to *cache* the top of the tree and speed up accesses. Caching is a common optimization with Path ORAM, which has a similar tree structure [33, 43, 49].

Unfortunately, such optimizations do not work well with an AVL tree in an SGX enclave, as the cache accesses themselves have to be oblivious. Since the nodes that are at the top of an AVL tree change throughout the tree’s lifetime, and depend on the access pattern in which they were inserted, such a cache would have to have many additional dummy accesses throughout the tree to hide when nodes are rotated in or out of the cache area. The prior work that we compare against, Tinoco et al. [49], therefore uses a cache at the Path ORAM level, not the AVL level, which can be implemented identically for oblivious hash tables as well.

Moreover, caching in an SGX enclave has, somewhat paradoxically, only limited value. As detailed in Section 5, when large amounts of data are stored in the enclave, the only way to obviously access them is to do a linear scan over the entire array. Operations within the enclave are much faster than external memory accesses, but the larger a cache is, the less efficient it is due to $O(n)$ access cost. Having multiple separate smaller caches, i.e., the position map and separate caches for the top of each recursive Path ORAM, provides more benefit than having a single large cache, and a single cache would be all that is available to an oblivious AVL tree.

The end result is that hash tables can shore up their weakness of having a larger position map with access to a modest amount of client memory, while AVL trees get no special benefit. The constant number of Path ORAM accesses then outweighs the logarithmic number in AVL trees and results in better performance.