# $H_2O_2$RAM: A High-Performance Hierarchical Doubly Oblivious RAM

Leqian Zheng
*City University of Hong Kong*

Zheng Zhang
*ByteDance Inc.*

Wentao Dong
*City University of Hong Kong*

Yao Zhang
*ByteDance Inc.*

Ye Wu
*ByteDance Inc.*

Cong Wang
*City University of Hong Kong*

## Abstract

The combination of Oblivious RAM (ORAM) with Trusted Execution Environments (TEE) has found numerous real-world applications due to their complementary nature. TEEs alleviate the performance bottlenecks of ORAM, such as network bandwidth and roundtrip latency, and ORAM provides general-purpose protection for TEE applications against attacks exploiting memory access patterns. The defining property of this combination, which sets it apart from traditional ORAM designs, is its ability to ensure that memory accesses, both inside and outside of TEEs, are made oblivious, thus termed doubly oblivious RAM ($O_2$RAM). Efforts to develop $O_2$RAM with enhanced performance are ongoing.

In this work, we propose $H_2O_2$RAM, a high-performance doubly oblivious RAM construction. The distinguishing feature of our approach, compared with the existing tree-based doubly oblivious designs, is its first adoption of the hierarchical framework that enjoys inherently better data locality and parallelization. While the latest hierarchical solution, FutORAMa, achieves concrete efficiency in the classic client-server model by leveraging a relaxed assumption of sublinear-sized client-side private memory, adapting it to our scenario poses challenges due to the conflict between this relaxed assumption and our doubly oblivious requirement. To this end, we introduce several new efficient oblivious components to build a high-performance hierarchical $O_2$RAM ($H_2O_2$RAM). We implement our design and evaluate it on various scenarios. The results indicate that $H_2O_2$RAM reduces execution time by up to $\sim 10^3$ times and saves memory usage by a factor of $5 \sim 44$ compared with state-of-the-art solutions.

## 1 Introduction

With the growing adoption of cloud computing, there have been rapidly arising concerns about the security and privacy of data outsourced to the cloud. In this evolving landscape, trusted execution environments (TEEs) play an increasingly prevalent role due to their ability to provide enhanced security features (*e.g.*, isolation, confidentiality, and integrity) without significant performance overhead, reliance on trusted third parties, or the need for non-colluding servers. Furthermore, virtual machine-based TEEs (*e.g.*, Intel TDX [44], AMD SEV [4]) require minimal adaption efforts to migrate existing applications into secure environments. Consequently, many cloud providers [23, 36, 69] have incorporated TEEs as part of their infrastructure offerings.

In brief, TEEs [4, 44, 68] provide high-level security by isolating the execution of selected code and data from the main operating system, thus shielding them even from system administrators. Within the processor, a memory encryption engine transparently encrypts and decrypts confidential data as it moves to and from the main memory, using keys derived from a root key permanently embedded in the processor. Furthermore, users can verify the integrity of the application or the system that operates within the TEEs via an attestation process [47], thus ensuring that computations are executed correctly and privately in the untrusted cloud.

However, many works [12, 13, 14, 22, 55, 56, 59, 71, 105] have shown the shortcomings of TEEs that compromise user privacy. Some of them violating the intended security model (*e.g.*, those from speculative execution [14, 22] and power analysis [59]) are due to design or implementation flaws and will be patched by the corresponding manufacturers. In contrast, side-channel attacks [55, 56, 105] related to memory access patterns generally remain beyond the security goal of mainstream TEEs, representing a persistent challenge.

This work focuses on concealing the memory access pattern that has been shown to be devastating in many applications [15, 39, 46, 60, 65, 104, 111] in TEEs. Since the seminal work of Goldreich and Ostrovsky [34], oblivious RAM (ORAM) has been recognized as a general and standard solution toward this goal. After decades of continuous development, ORAM has undergone significant advances with considerable efforts dedicated to performance optimization. A promising line of work [18, 24, 27, 70, 84, 88, 89, 96, 110] leverages TEEs to deploy a trusted client in the TEE to access the untrusted server with minimal latency and high band-

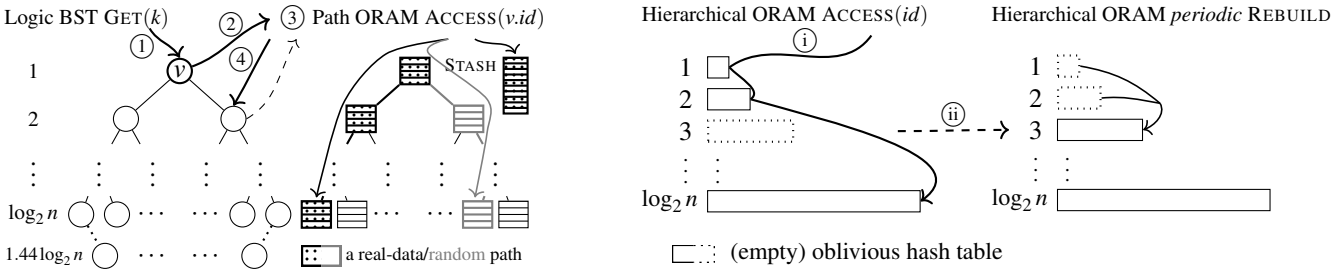



Figure 1: Simplified tree-based/hierarchical oblivious map access with $n$ data blocks.

width. This effectively mitigates one of the critical performance bottlenecks in the classic client-server model, *i.e.*, network latency due to high bandwidth demands and round-trip complexity. Meanwhile, the use of TEEs, in turn, brings a new consideration, *i.e.*, ensuring that data accesses are oblivious both outside and within the TEE itself. Thus, such solutions [18, 70, 84, 88, 96] are coined as *doubly oblivious* RAM (DORAM). To better distinguish it from distributed ORAM or differentially ORAM, we rename it as $O_2$RAM.

**ORAM roadmap.** Informally, ORAM conceals on which data blocks a client operates and whether these operations are reads or writes. This is achieved by accessing extra dummy blocks, shuffling the data periodically, and consistently writing the data back after each read. As discussed in [8], one prevalent category of tree-style ORAM designs, derived from Path ORAM [95], excels in concrete efficiency, but falls short of achieving optimal theoretical complexity. To get the value of a key $k$ in such designs as shown in Fig. 1, one has to *recursively* execute the following four steps until reaching the maximal possible height of a binary search tree, *e.g.*, $\lceil 1.44 \log_2 n \rceil$ for an AVL tree. ① starting from the root node, it obliviously compares $k$ with the current node value $v$ to determine whether the left/right node is the next one. ② to get $v$, it invokes Path ORAM's access protocol using the path identifier *id* stored in the node. ③ the access protocol retrieves a path by *id* together with a randomly picked path. It linearly scans all the values retrieved and its stash to get the value $v$. Each node in Path ORAM, referred to as a bucket, contains $\sim 4$ data blocks, and the stash holds $\omega(\log n)$ blocks. ④ afterwards, an eviction operation obliviously re-arranges the real data blocks over two retrieved paths and the stash as close to the leaves as possible w.r.t. the path invariance.

The other category derived from the foundational square-root ORAM [33] draws more theoretical interest in the past. It typically comprises $\log n$ levels, with each level increasing geometrically in size and structured as *oblivious hash tables* (to be precise, oblivious w.r.t. non-recurrent lookups). To retrieve a value associated with *id*, the access protocol ⓘ, as shown in Fig. 1, visits each nonempty level and re-writes the target data block back to the first one. A lookup in an oblivious hash table generally completes in $\tilde{O}(1)$ time. ⓘⓘ to maintain obliviousness, it extracts data from the first $i$ levels and rebuilds them into the $(i+1)$-th level after every $2^i$ accesses. This rebuilding process, essentially relying on oblivious shuffling, dominates the computation overhead of hierarchical ORAM designs [6, 8, 77]. This line of work has recently achieved groundbreaking progress in achieving asymptotic optimality [6, 25]. However, its practical efficiency is largely constrained by the huge constant hidden in the big-O notation. Asharov et al. [8] continue to optimize along this path, making it the first concretely efficient hierarchical scheme by allowing a sublinear yet reasonable-size client-side memory that does not expose access patterns.

**Insight:** *Under the premise of comparable complexity, hierarchical ORAM exhibits better concrete efficiency than tree-style ORAM due to its better data locality and parallelization.* As described above, data blocks are scattered along paths in tree-style ORAM, whereas in hierarchical ORAM, data blocks are continuously aligned at each level, facilitating better utilization of the cache and bandwidth of the chip. Meanwhile, hierarchical ORAM is more amenable to parallelization as the most time-consuming rebuilding process ⓘⓘ, which mainly relies on oblivious shuffling, can be easily parallelized. Although the eviction process in tree-style ORAM can also be partially parallelized, the volume of data involved is too small to justify the performance overhead caused by threading. Furthermore, while there are indeed some studies [17, 24, 102] exploring the parallelization and scaling of tree-style ORAMs, these efforts mainly focus on scaling *a batch of* operations rather than parallelizing the internal computation for *individual* operations. Such a batch process technique is not ideal for many tasks, such as computing shortest paths, determining maximum flow, or data accesses with sequential oder.

Given the achievable similar asymptotic complexity between hierarchical and tree-style ORAMs, we hence prefer the hierarchical one as our technical roadmap for further optimization. In specific, our work builds upon FutORAMa [8] that represents a leading advancement in hierarchical ORAM. In essence, their approach assumes a reasonable yet non-constant, local memory that inherently does not expose access patterns (which does not apply to our scenario). In addition, they have implemented several elegant optimizations in the oblivious rebuilding and extraction processes of *large* hash tables, approaching asymptotically optimal complexity.

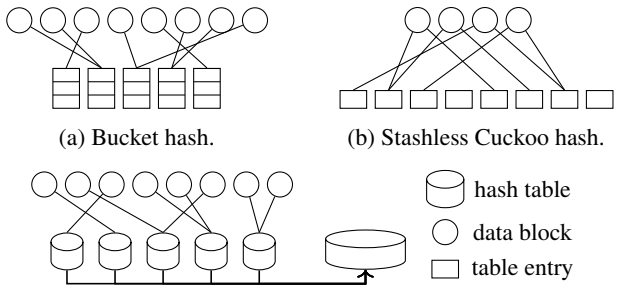
**The challenge.** Developing an efficient doubly oblivious version of FutORAMa [8] is non-trivial, as its high performance is heavily dependent on the use of sublinear-sized private memory that inherently conceals any access pattern. Without delving into excessive specifics, FutORAMa [8] randomly distributes the data blocks at each level into bins (configured as hash tables), and then secretly selects a small subset of these blocks from each bin to a secondary and similarly configured structure. The key point is that the client could leverage its private local memory to handle both *small levels* and *multiple bins* at large levels in a *non-oblivious* (*i.e.*, highly efficient) way. A straightforward approach that replaces local memory with existing $O_2$RAM designs [18, 70, 84] would incur substantial overhead. Our empirical evaluation shows that even a simple look-up in such a naïve replacement (*i.e.*, with a Path ORAM) incurs much more overhead than our final solution.

Therefore, the primary challenge lies in constructing *efficient* doubly oblivious bins, *i.e.*, hash tables, that are of small to moderate size (concretely, less than $\sim 2^{18}$ data blocks or tens to hundreds of megabytes). This challenge arises for two main reasons: 1) each hash bin leveraged in large levels must be implemented efficiently and obliviously; 2) while individual access times of small levels are significantly lower than those of large ones, optimizing their efficiency is also crucial as they are much more frequently accessed and rebuilt.

While several studies [6, 19, 35, 52, 77] have investigated efficient hash table designs for ORAM, most focus on optimizing large hash tables by decomposing them into several moderately sized ones, which are typically implemented as oblivious Cuckoo hash tables [19, 35]. These approaches are hence not applicable to our scenario. As for the existing oblivious Cuckoo hashing schemes, they face two main issues: 1) a time-consuming oblivious build process; and 2) the necessity for a linear scan over a stash for a lookup to achieve negligible overflow or failure probability. As acknowledged in [26, 99] and further confirmed in [70, 73], the minimum size requirement of a stash significantly slows down the overall lookup of a Cuckoo hash table. This is because, for the oblivious access of a few dozen data blocks, a linear scan remains the most efficient method. In concrete terms, existing Cuckoo hashing designs require a linear scan over *hundreds or thousands* of data blocks for a *single* operation, severely affecting their concrete performance. In short, *existing approaches are inadequate to construct doubly oblivious hash tables w.r.t. non-recurrent lookups that are concretely efficient, especially for those of small to moderate size*.

**Our approach.** To address the above issues, we tailor three oblivious hashing schemes, as shown in Fig. 2, along with a naïve linear scan to handle varying scenarios.

In specific, the bucket hash scheme, as shown in Fig. 2a, uses a pseudorandom function (PRF) to assign each block across buckets of uniform size. Each lookup requires a linear scan within the bucket identified by the PRF. The key aspect here is to derive the minimal bucket size to achieve a negligible bucket overflow probability caused by hash collisions. These bucket hash tables, detailed in §3.1, are well-suited for managing small to moderately sized levels.




(a) Bucket hash. (b) Stashless Cuckoo hash.

(c) Two-tier hash, where a secret number of blocks from the main hash tables will be (obliviously) relocated to a secondary hash table.

Figure 2: Tailored hashing schemes used in this work.

Our Cuckoo hash scheme, adapted from the elegant design [106] as shown in Fig. 2b, removes the stash without sacrificing its negligible overflow probability. The key modification in the stashless design is simple yet powerful: use slightly more hash functions, each enjoying disjoint table entries. However, there are still two technical issues to be addressed: 1) the work does not specify how to *obliviously* build and access such hash tables. The existing designs for oblivious Cuckoo hash tables [19, 35] are also not suitable for our scenario, as they are tailored for mechanisms with only two hash functions. To address this, we propose a new oblivious bipartite graph matching algorithm that may also be of independent interest in other domains. We note that the complexity of this (relatively) expensive process remains independent of data block size, thus providing substantial benefits to our design in scenarios involving large blocks. 2) the asymptotic failure probabilities in [106] are only tight for sufficiently large $n$, which deviates from our focus on moderate $n$. We hence derive a more concrete bound and employ numerical methods to compute more precise and relevant estimates, proving that $3 \sim 6$ hash functions are sufficient to achieve a negligible failure probability. These stashless Cuckoo hash tables, detailed in §3.2, excel in scenarios where the number of lookups exceeds that of the data blocks they store.

The two-tier hash scheme, adapted from FutORAMa [8] and illustrated in Fig. 2c, closely resembles the bucket hash. Its performance benefits from the ability to safely place data blocks into major hash bins/tables in a *non-oblivious* (*i.e.*, efficient) way, given that 1) the input data are randomly shuffled, and 2) a small portion of each major bin is secretly and obliviously relocated to a secondary hash table. This approach effectively avoids the need for a relatively costly oblivious bin placement process. Each lookup involves accessing a major bin identified by the PRF and the secondary hash table. Due to the minimum size requirement for the major bin to prevent both data overflow and underflow, this design is preferable

for large levels. Further details are provided in §3.3.

In addition, we derive theoretical complexity analysis for the three oblivious hashing schemes mentioned above. However, asymptotic analysis alone is insufficient for identifying the practically fastest scheme and parameters across various real-world scenarios, as factors such as parallelization, cache friendliness, and other physical considerations are hard to formalize and analyze as a whole to derive the optimal choice. We hence develop a hash scheme planner, as discussed in §4, to empirically select approximately optimal ones.

**Implementation and evaluation.** We implement, evaluate, and open source our $O_2$RAM design, along with some applications in doubly oblivious data structures and algorithms, *e.g.*, $O_2$Map and single-source shortest paths. To the best of our knowledge, it is the first open-source $O_2$RAM design based on the hierarchical roadmap (the one in FutORAMa is implemented in Python and serves more as a simulation). We compare our design with state-of-the-art tree-based $O_2$RAM designs, EnigMap [96] for $O_2$RAM and $O_2$Map, and GraphOS [18] for oblivious single-source shortest path computation. The results show that `H2O2RAM` outperforms EnigMap [96] by factors up to $10^3\times$ in map operations. When integrated into specific applications, `H2O2RAM` achieves up to a $142.3\times$ speedup in doubly oblivious single-source shortest path computations compared to GraphOS [18] (with its results taken directly from Figure 6c of their paper). In addition, `H2O2RAM` also enjoys lower memory overhead, *e.g.*, saving up to $44\times$ the memory space compared to EnigMap [96].

**Contributions.** We summarize our contributions as follows:

- We develop the first *doubly oblivious* RAM based on the hierarchical roadmap, which not only shows concrete efficiency but also outperforms tree-based designs in practice.
- Some of the building blocks, including oblivious stashless Cuckoo hashing, oblivious bipartite matching, and concrete parameter selection, are of independent interest.
- We provide an open-source implementation of `H2O2RAM` on https://doi.org/10.5281/zenodo.14648338 along with empirical evaluations to show its concrete performance.

## 2 Preliminaries

**Balls into bins problem.** The process of throwing $m$ balls into $n$ bins in a uniformly random way perfectly simulates the data allocation of a bucket hash scheme, with only a negligible probability of deviation introduced by the PRF it uses. Our objective to find the minimum bucket size for a negligible overflow probability can be formalized as finding a threshold $k$ s.t. the probability of any bin that contains more than $k$ balls is negligible. A general answer for this problem when $m = n$ is $k = \lceil e \cdot \log_2 n \rceil$. However, this general answer is unsatisfactory for us to build an efficient bucket hash table. We hence choose to state the following lemma to show a more precise overflow probability for finding a better bucket size:

**Lemma 1.** *If $n$ balls are thrown independently and uniformly into $m$ bins, the probability of a bin having at least $k$ balls is:*

$$\Pr[\text{overflow}] \leq \sum_{i=k}^{n} \binom{n}{i} \frac{(m-1)^{n-i}}{m^n}.$$

**Bipartite graph and matching.** A bipartite graph is a graph $G = (L \cup R, E)$ where the vertex set $V$ is divided into two disjoint sets $L$ and $R$, s.t. every edge connects a vertex in $L$ and a vertex in $R$ while no edge connects vertices within the same set. A *matching* in a bipartite graph is a subset of edges $M \subseteq E$ where no two edges share a common vertex. A matching is called left-perfect matching if the edges of $M$ cover all vertices in $L$.

**Cuckoo hashing and its variants.** Cuckoo hashing [76] in its simplest form has 2 hash tables, $T_1$ and $T_2$, each with $m/2$ entries of unit capacity. The hash function $h_i$ for each table $T_i$ is chosen independently and is assumed to distribute inputs uniformly in $\{1, \ldots, m/2\}$. Given an item $x$, it is stored in either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. To put it simply, we assign sequential indices from 1 to $m$ to the entries of both hash tables. The outputs of hash functions are then adjusted with an appropriate offset.

As our focus is on scenarios where all input blocks are provided in advance rather than arriving online, constructing the Cuckoo hash table could be reduced to a bipartite matching problem. Specifically, $n$ input data blocks and $m$ table entries form the disjoint vertex sets $L$ and $R$, respectively. Each hash function establishes an edge $(x, h_i(x))$ for each input data block $x$. It is clear that a matching directly corresponds to a table assignment and vice versa.

However, a left-perfect matching is not guaranteed to exist on such random graphs, indicating the input data blocks may overflow the hash table. To address this, many variants have been proposed [31, 50, 57, 73, 106], including one of the most prevalent ones that place all overflowed data blocks in a stash. And it has been shown in [31, 73, 106] that the stash must be of $\Omega(\log n)$ size to prevent the overflowed blocks from exceeding its capacity. Noble [73] lists some concrete figures that for a single hash table of size $2^{16}$, the stash size has to be $\geq 14$ to obtain a $2^{-40}$ failure probability. As discussed in §3.2, such stash sizes will bring notable performance drops.

Fortunately, Yeo [106] propose a novel variant that requires no stash but slightly more PRFs. In specific, it divides the entire table into $k$ sub-tables, each comprising $\lfloor m/k \rfloor$ entries. Every data block is then given $k$ candidate entries, one in each sub-table, selected by $k$ independent PRFs. It is clear that the construction process is still equivalent to bipartite matching. The following lemma states its failure probability:

**Lemma 2** (derived from [106]). *The failure probability of the above cuckoo hashing scheme with n input data, k independent PRFs, and m table entries is upper bounded by:*

$$\Pr[\mathsf{fail}] \leq \sum_{t=k+1}^{n} \binom{n}{t}\binom{m}{t-1}\left\lfloor\frac{m}{k}\right\rfloor^{-kt}\prod_{i=1}^{k} a_i^t, \text{ with} \sum_{i=1}^{k} a_i = t-1.$$

The bound converges to $n^{-\Theta(k^2)}$ for sufficiently large $n$ and $k$. For practical usage, we employ numeric methods to compute appropriate $n$ and $k$, with detailed results provided in §5. In brief, when $m = 2n$, three to six hash functions are sufficient to reduce the failure probability to $2^{-64}$.

## 2.1 Threat Model

We employ a threat model similar to those used in previous studies [18, 70, 84] that integrate oblivious primitives with Trusted Execution Environments (TEEs). The key distinction between our model and the classic client-server model adopted in FutORAMa and other purely cryptographic designs lies in the fact that the client in the traditional model operates a fully trusted machine actively involved in the computation. In contrast, in our model, the client encrypts its data and uploads them to an untrusted server. The computation is then fully outsourced to the TEE, which physically resides on the untrusted server and may be exposed to various attacks. Therefore, it is necessary to consider the presence of strong adversaries who can control the entire server's software stack and the operating system, and even gain physical access to the server. While we do trust the secure processor and exclude cases in which an adversary could extract information from within the processor. We also assume that the TEE upholds its claimed security properties. Although in reality, such an assumption may not always hold, just as in many other real-world applications, we can apply patches released by the manufacturer in a timely manner to maintain these standards.

In specific, we assume a remote attestation process [47] that can help the client verify the identity and integrity of the TEE, and build a secure and authenticated communication channel. Data confidentiality is offered by a memory encryption engine within the processor that *transparently* encrypts and decrypts private data as it travels to and from the chip cache, using keys derived from a root key burned during manufacturing. Adversaries are also unable to tell whether two ciphertexts in and out the chip originate from the same plaintext. We hence omit explicit descriptions of data encryption/decryption. In line with prior works [18, 27, 70, 84, 96, 110], we consider timing and power analysis attacks [59], rollback attacks [67], and denial-of-service attacks as orthogonal issues. The interested readers may seek for works [82, 87, 89, 90] for mitigations.

However, adversaries can observe *which* memory data the TEE accesses, *i.e.*, memory access patterns, using various methods and at different granularities. First, the untrusted host operation system can observe page-level access patterns via the page tables it maintains [4, 44]. A malicious system administrator can hence easily observe page-level access patterns. Second, shared resources, such as caches, can reveal which memory locations have been accessed. In addition, an attacker can mount an affordable hardware attack [55] to obtain the exact data addresses accessed by the TEE by snooping on the physical memory bus. It is hence essential to fully conceal the memory access patterns for general usage.

## 2.2 Oblivious Primitives

Due to space constraints, we defer some formal definitions to Appendix A, but provide an overview of their concepts here.

**Obliviousness.** A RAM (or Turing machine) consists of a CPU containing a constant number of registers that are oblivious to adversaries, and a memory containing $n$ words indexed by $\{1,\ldots,n\}$. To execute a program that has some inputs and outputs, the CPU will interact with the memory in the form of reads and writes. Adversaries can observe *which* memory words the CPU accesses, named access pattern. We say that the program is oblivious if its access pattern is (statistically/computationally) independent of the content of input data. Namely, its access pattern can be simulated given only the input length. We omit the formal definitions for the oblivious algorithms as they can be found in the referenced papers [6, 8, 77]. Meanwhile, we slightly abuse some terms by treating data blocks (items) the same as memory words of bit-length $w$. In implementation, we will consistently access the entire data block that may span several memory words.

**Oblivious operations.** We follow some standard assumptions [18, 70, 86] that random bits can be generated obliviously. In our pseudocode, we leverage more readable descriptions like "if $x > y$ then $a$ else $b$", while they are implemented obliviously with some instruction-trace oblivious techniques [61, 62, 96].

**Oblivious sorting/shuffling.** We adopt a classic yet concretely efficient sorting algorithm, bitonic sort, in our design. It achieves $O\left(n\log^2 n\right)$ complexity for $n$ data blocks. Oblivious shuffling is implemented via bitonic sorting with uniformly random keys. A recent advancement, WAKSSORT [86], outperforms bitonic sort through an offline preprocessing phase. However, we choose not to adopt this design because the offline phase generates some randomness, resulting in considerably large space overhead as our work relies heavily on frequent invocation of oshuffle/osort.

**Oblivious compaction.** Given $n$ data blocks where some of the inputs are marked, it obliviously moves all the marked blocks to the front. There exist concretely efficient oblivious compaction algorithms [58, 85] taking $O(n\log n)$ time. By further assuming that the input array is randomly shuffled with exactly half of the items marked, Asharov et al. [8] achieve $O(n)$ time with a negligible error probability $2n\exp(-Z/256)/Z$, where $Z$ denotes the local memory size. The original design compacts $Z$ items locally in $O(Z)$ time. However, in our

model, this process must be performed obliviously, introducing an additional $O(\log Z)$ multiplicative overhead. The total time hence becomes $O(n \log Z)$. When $n$ is small, the former is faster; otherwise, the latter is more efficient (a concrete threshold depends on the machine and block size). In our implementation, we always choose the better one.

**Oblivious intersperse.** Originated in [77] and further optimized in [6], intersperse refers to the process of "merging" two randomly shuffled arrays into a single one that is also randomly shuffled. It is more efficient than naïvely shuffling two arrays collectively and achieves the asymptotical optimality of $O(n)$ time. In brief, it generates a random bit array, where each bit indicates whether the element at that location originates from the first or the second input array. By reversing the process of *obliviously compacting* the bit array, we can shuffle the items appropriately. By slightly abusing the term, we also refer to it as "oblivious shuffling". Whether it is implemented as bitonic sort or intersperse, similar to the aforementioned ocompaction, depends on the specific context.

**Oblivious bin placement.** Given $n$ items, each tagged with a destination bin, the functional goal is to put items into their respective bins. Assuming $m$ bins of uniform size $k$, we adopt the method from [5] with $O\big(N \log^2 N\big)$ time, where $N := n + m \cdot k$. In essence, it involves one round of osorting and one round of ocompaction over $N$ items.

**Oblivious hash table.** Given $n$ possibly dummy items in the form of $(k, v)$ pairs, an oblivious hash table supports the following three algorithms:

- build takes as inputs $n$ elements and creates a data structure;
- lookup receives a possibly dummy key $k$ and outputs the value $v$ corresponding to the key $k$ in the data structure or $\perp$ if $k$ is dummy or it is not found in the table;
- extract destructs the data structure and returns all elements that have never been looked up.

The security goal is to ensure that each of the three algorithms, both individually and in combination, remains oblivious. Besides, we never call lookup with duplicate keys. In practice, we may represent distinct real keys as positive integers and dummies as negative integers, utilizing a dummy counter to avoid duplicate dummy lookups, under an input assumption of non-recurrent real keys.

**Hierarchical ORAM.** We opt to provide a detailed, end-to-end description of $\mathtt{H_2O_2RAM}$, adapted from FutORAMa [8], in Appendix A. Instead, we provide a concise discussion of its process and components here, along with a simplified illustration of $\mathtt{H_2O_2RAM}$ in Fig. 1. This discussion is for completeness and will also incorporate some minor modifications necessitated by our threat model.

Without loss of generality, we assume that the capacity $N$ of the ORAM (*i.e.*, the maximum number of data blocks) is a two-power, otherwise, we can pad it to the next two-power. Let $L = \log N$, we create $L$ levels with each level $i$ instantiated as an oblivious hash table having a capacity of $2^i, \forall i \in \{1, \ldots, L\}$ and initially marked as *empty*. The bottom level, initiated with $N$ data blocks indexed from 1 to $N$, is marked as non-empty.

In the access phase, given an operation $\mathtt{op} \in \{\mathtt{read}, \mathtt{write}\}$, an address $\mathtt{addr} \in [N]$, and a data block $v \in \{0,1\}^w$, we first initialize $\mathtt{res} := \perp$, then for each nonempty level $i$: we lookup $\mathtt{addr}$ (or $\perp$) in its hash table if $\mathtt{res} = \perp$ (or $\mathtt{res} \neq \perp$); we then obliviously write $\mathtt{res}$ with the returned data if its key matches $\mathtt{addr}$. After the loop, we obliviously write $\mathtt{res}$ with $v$ if $\mathtt{op} = \mathtt{write}$. If the first level is full, we find the first empty level $i^*$ or set $i^* = L$ if all levels are full, then extract data from levels 1 to $i^* - 1$, and build $i^*$-th level with the extracted data as input. Finally, we return $\mathtt{res}$ to the client.

## 3 Oblivious Hash Schemes

As discussed in §1 and §2, oblivious hash tables play a central and foundational role in hierarchical ORAM. In this section, we elaborate on the design of three tailored hash schemes as well as their parameter planners for better efficiency. Let $n$ denote the number of data blocks a hash table handles, and $t$ denote the total number of lookups over its lifetime.

**Concrete failure probability computation.** As described in §2, we opt to compute more precise overflow/error probabilities for better performance without compromising the system's security. Specifically, given that the computation primarily involves combinatorial and factorial numbers, we choose to compute their logarithms rather than their original values for better efficiency and numerical stability. We will exponentiate the results back to their original values when necessary. In addition, as the targeted overflow/error probabilities exceed native floating-point precision, we use high-precision mathematical libraries to ensure accurate results.

### 3.1 Oblivious Bucket Hash

**Tight bucket size computation.** In oblivious bucket hash tables, we first have to determine the tight bucket size $\ell$ given $n$ and the number of buckets $m$. Equipped with Lem. 1 and methods for calculating accurate probabilities, we can easily derive these sizes by binary search.

**Construction.** We then describe its construction as follows:

•$\mathsf{build}(A, m)$: given $A$ containing $n$ input data blocks with unique keys and $m$ denoting the number of buckets, it begins by calculating the tight bucket size $\ell$ and initializing the table $T$ with $m$ empty buckets. A PRF $\mathsf{PRF}$ with range $\{1, \ldots, m\}$ and its key $\mathtt{sk}$ are sampled to compute the target bucket $\mathsf{PRF}_{\mathtt{sk}}(k_i)$ for $i$-th data block, after which the data in $A$ are obliviously placed into their target buckets (see §2).

$\mathsf{lookup}(k)$: it performs a linear scan of the bucket identified by $\mathsf{PRF}_{\mathtt{sk}}(k)$ to obliviously select the matching data, which is subsequently marked as a dummy entry.

•$\mathsf{extract}()$: oblivious compaction is performed to retain exactly $n$ data entries, including all real ones.

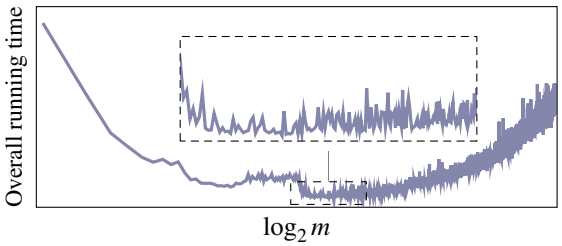



Figure 3: Trend in overall running time for oblivious bucket hash tables as the number of buckets $m$ varies.

**Time complexity.** The build process runs in the time complexity of $O\big((n+m\ell)\log^2(n+m\ell)\big)$ due to the oblivious bin placement operation. Assuming a hash table size of $m\ell = \Theta(n)$, it can be simplified as $O\big(n\log^2 n\big)$. Each lookup requires $O(\ell)$ time due to the linear scan performed over the bucket. The extraction mainly involves an ocompaction process, resulting in $O(m\ell \cdot \log(m\ell))$ time. The overall time complexity for a bucket hash table handling $t$ lookups is hence $O\big((n+m\ell)\log^2(n+m\ell)+t\ell\big)$.

**Optimal bucket number.** So far, we have not yet specified how to determine the number of buckets $m$ for optimal running time. As discussed in §1, the above asymptotic analysis serves only as a reference and does not directly derive the optimal $m$. In fact, the actual running time, as shown in Fig. 3, exhibits an approximately convex shape with significant fluctuations, where even slight changes in $m$ can lead to notable running-time variations. Therefore, the highly time-consuming brute-force search is likely the only feasible method to find the optimal number of buckets. However, if we relax the objective to an *approximately* optimal solution, a variety of classic optimization methods can be employed. In specific, we adopt the golden-section search [49] due to its simplicity.

## 3.2 Oblivious Stashless Cuckoo Hash

We first claim that no solution practically surpasses naïve linear scans when dealing with small sets of data blocks. This has been empirically shown in several studies [26, 70, 99]. We supplement these findings by highlighting two critical factors that contribute to the observed deficiencies: 1) the oblivious build process typically involves several rounds of osorting over data sets that are two to three times larger than the input one; and 2) the lower bounds on the number of entries that must be scanned during each lookup to achieve a negligible collision probability closely align with the size of the input data. Concretely, for up to 256 items, both bucket hash and Cuckoo hash with a stash [73] require scanning nearly 50 entries per lookup. Similarly, a doubly oblivious map based on Path ORAM [18, 70] requires a comparable number of scans, not to mention its higher building complexity.

The above finding places Cuckoo hash *with* a stash in an awkward position in handling moderate-sized levels, as its stash typically holds hundreds of data blocks. It is worth noting that combining stashes [35] across the entire ORAM does not effectively mitigate this issue, as the combined one is still relatively small. Fortunately, Yeo [106] proposed an elegant design that removes the need for a stash and proved its asymptotic optimality. However, making this design oblivious remains an unresolved challenge. To this end, we introduce an oblivious bipartite matching algorithm as follows.

### 3.2.1 Oblivious Bipartite Matching

We remark that although our design resembles the existing oblivious Cuckoo hash build algorithm [19], there are fundamental differences. In their scheme, the two disjoint vertex sets comprise entries in the two hash sub-tables, with edges defined by $(h_1(x), h_2(x))$ for each item $x$ where $h$ denotes PRFs. While in our design, the left vertex set consists of $n$ input items, the right vertex set consists of $2n$ table entries, with $k$ edges $(x, h_i(x)), i \in [k]$ for each item $x$.

In fact, our algorithm is an oblivious version of the Hopcroft-Karp algorithm [43]. In brief, all edges within the matching are directed from right to left, while those outside the matching are directed from left to right. We call a pair of edges that share the same vertex, with exactly one of them included in the matching, an alternating path. With slightly abusing the terms, we also refer to the edges in the match as *alternating* edges. Adding a new item (*i.e.*, a new vertex in $L$) is equivalent to *recursively* finding an alternating path from the new vertex to any right vertex that is currently free. For free vertices in $R$, we also name the edges incident to them *free* edges. We give a toy example of this process in Fig. 4.

It is evident that the above recursive process will expose the structure of the bipartite matching, thereby exposing sensitive information of the input data. Conceptually, our oblivious algorithm transforms this recursive process into a *propagation*-based way. Namely, a vertex in $L$ propagates its available *free* and *alternating* edges to the others via its neighbors in $R$. As shown in Alg. 1 line 2 , we first tag each raw edge $e$ with three additional fields: 1) `dir` $\in \{\ell, r\}$ denotes its direction, 2) `st` $\in \{$`unknown`, `free`, `alternat`, `alternat_bk`$\}$ denotes its state, and 3) `ctr` denotes the number of times it has been included in the match. Note that an edge directed to the left indicates that it is or tends to be included in the match. We

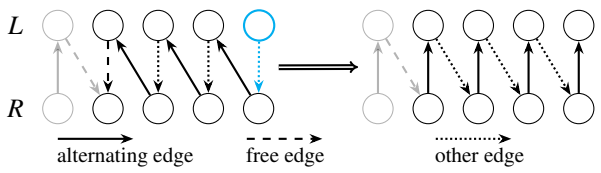



Figure 4: An example of bipartite matching: the alternating/augmenting path is highlighted by the bold edges, all of which are reversed after inserting the blue vertex.

**Algorithm 1** Oblivious bipartite matching omatch

**Input:** A bipartite graph $G = (L \cup R, E)$, a loop upper bound $\tau$;
**Output:** A matching $M \subseteq E$.

1: **for** $e = (u,v) \in E$ **do**
2: $\quad e \leftarrow \{(u,v),\ \texttt{dir} := r,\ \texttt{st} := \texttt{unknown},\ \texttt{ctr} := 0\}$
3: **for** $t \leftarrow 0 \ldots, \tau$ **do**
4: $\quad$ obliviously sort $E$ by the rules in descending priority:
$\quad\quad$ 1. $e.u$ ▷ *group edges incident to the same vertex in L*
$\quad\quad$ 2. $e.dir = \ell$ ▷ *edges toward left first*
$\quad\quad$ 3. $e.\texttt{st} = \texttt{free}$ ▷ *free edges first*
$\quad\quad$ 4. $e.\texttt{ctr}$ ▷ *edges with smaller counters first*
5: $\quad$ initialize the matching result $M \leftarrow \emptyset$, $e_0 \leftarrow \emptyset$
$\quad$ ▷*all if-else clauses are implemented obliviously as introduced in §2*
6: $\quad$ **for** $e = (u,v) \in E$ **do**
7: $\quad\quad$ **if** $e$ is the first edge in its group $e.u$ **then**
8: $\quad\quad\quad$ **if** $e_0.\texttt{st} = \texttt{alternat\_bk}$ **then**
9: $\quad\quad\quad\quad$ $e_0.\texttt{st} \leftarrow \texttt{alternat}$
10: $\quad\quad\quad$ increment $e.\texttt{ctr} \leftarrow e.\texttt{ctr} + 1$ if $e.\texttt{dir} = r$
11: $\quad\quad\quad$ set $e.\texttt{dir} \leftarrow \ell, M[e_0.u] \leftarrow e.v$, $e_0 \leftarrow e$
12: $\quad\quad$ **else if** $e.\texttt{st} = \texttt{free}$ and $e_0.\texttt{st} = \texttt{alternat}$ **then**
13: $\quad\quad\quad$ set $e_0.\texttt{dir} \leftarrow r, e.\texttt{dir} \leftarrow \ell, M[e.u] \leftarrow e.v$
14: $\quad\quad\quad$ increment $e.\texttt{ctr} \leftarrow e.\texttt{ctr} + 1$
15: $\quad\quad\quad$ update $e_0.\texttt{st} \leftarrow \texttt{unknown}, e.\texttt{st} \leftarrow \texttt{unknown}$
16: $\quad$ obliviously sort $E$ by the rules in descending priority:
$\quad\quad$ 1. $e.v$ ▷ *group edges incident to the same vertex in R*
$\quad\quad$ 2. $e.dir = \ell$ ▷ *edges toward left first*
$\quad\quad$ 3. $e.\texttt{ctr}$ ▷ *edges with greater counters first*
17: $\quad$ **for** $e = (u,v) \in E$ **do**
18: $\quad\quad$ **if** $e$ is the first edge in its group $e.v$ **then**
19: $\quad\quad\quad$ set $\texttt{st} \leftarrow \texttt{free}$ if $e.\texttt{dir} = r$ else $\texttt{unkown}$
20: $\quad\quad\quad$ set $e.\texttt{st} \leftarrow \texttt{st}, e_0 \leftarrow e$
21: $\quad\quad$ **else**
22: $\quad\quad\quad$ set $e.\texttt{st} \leftarrow \texttt{st}$
23: $\quad\quad\quad$ **if** $e.\texttt{dir} = \ell$ **then** $e_0.\texttt{st} \leftarrow \texttt{altert\_bk}$, $e.\texttt{dir} = r$
24: **return** $M$

then iterate for a predefined and *data-independent* times $\tau$. In each iteration, we obliviously sort the edges according to the rules shown in line 4. It groups edges by their left vertices, and then edges toward the left first, followed by free edges toward the right. A counter is used to ensure that the edges previously tied in the back are moved to the front in each iteration. The remaining ties are broken arbitrarily and consistently, *e.g.*, using the right vertex. As shown in line 6, we then try to include the first edge of each group in the match $M$. If the first edge tends to be alternated (*i.e.*, reversed) and there do exist free edges, we will replace the old match by a new one (*i.e.*, augmenting), and also update the state of both edges as `unknown`.

We then proceed to propagate edge states via vertices in $R$, which need to obliviously re-group edges based on their associated vertices in $R$ as shown in line 16. Priority is given to edges that are oriented leftward and reversed more frequently within each group. Then a linear scan shown in line 17 will mark the edges as free if $v \in R$ is not in the match, or preserve only one match for $v$ and correct the others. If there is more than one edge oriented leftward within a group, *i.e.*, several left vertices are racing for $v$, we will mark the first winning one as `alternat_bk`. Let $m$ be the size of the edge set and $d$ be the maximum possible length of augmenting paths, which are public to adversaries, we state the following theorem.

**Theorem 1.** *Assuming the existence of left-perfect matching and setting the number of iterations* $\tau := 3d + 1$*, Alg. 1 finds a left-perfect matching in* $O(\tau m \log^2 m)$ *time.*

*Proof.* Alg. 1 runs in $O(\tau m \log^2 m)$ time as it involves $\tau$ iterations, each dominated by two oblivious sortings of $O(m \log^2 m)$ time. Meanwhile, the obliviousness holds as two inner for-loops are implemented obliviously, and $\tau$ is independent of the input data. The remaining challenge remains to prove its correctness.

In the first iteration, all left vertices race for some right vertices and only a subset wins; the winning edges are then marked as `alternat_bk`, and some are labeled as `free`. In subsequent iterations, all edges marked as `alternat_bk` remain in the match and their states are changed to `alternat`; edges already marked as `alternat` will "augment" if there are free edges in the same group, while vertices lost in the previous race attempt to find free edges or race for another round. The strategy of state transitions from `alternat_bk` to `alternat` effectively prevents vertices within the match from racing for free edges against vertices outside the match. A vertex in the match will safely release a right vertex only if both of its new and old mates are not engaged in racing. And a vertex outside the match can possibly match the newly released vertex in the next iteration. Note that in the non-oblivious algorithm shown in Fig. 4, we recursively augment from the newly inserted vertex (i.e., racing vertices) to a free right vertex. While in our algorithm, we progressively *release* vertices or *alternate* edges from the other end of the alternating path. As it costs three iterations per alternation and alternations are performed concurrently, it takes at most $3d$ iterations to alternate all edges in the longest alternating path in addition to the first iteration. Therefore, $3d + 1$ iterations suffice to find a matching that is assumed to exist. □

For general bipartite graphs with $n := |L|$, we have a simple collary Alg. 1 finds a max-matching in $O(nm \log^2 m)$ time as the maximum possible length of alternating paths is $n$.

### 3.2.2 Oblivious Stashless Cuckoo Hash Table

Equipped with the oblivious bipartite matching algorithm, we are now prepared to present the construction of the oblivious stashless Cuckoo hash as shown in Alg. 2. To build the table from an input of $n$ data blocks and $k$ PRFs, we first establish the edges based on the keys of the data blocks, as shown in line 5. We then run Alg. 1 to solve the allocation of input data, and subsequently place them obliviously (empty

entries are filled with dummies). For lookups, we scan $k$ table entries based on the PRF values for a real lookup, or $k$ randomly selected entries for a dummy lookup. Data extraction is performed obliviously by compacting real items to the front. Thm. 2 establishes its security and efficiency.

**Algorithm 2** Oblivious stashless cuckoo hash CHT

build$(A,k)$ : ▷ *$A$ contains $n$ input items $(k_i, v_i)$ with distinct keys*
▷ *$k$ denotes the number of hash functions to be used*
1: compute bucket size $b \leftarrow \lfloor 2n/k \rfloor$
2: sample a PRF with range $\{1,\dots,b\}$
3: uniformly samples $k$ PRF keys $\mathtt{sk}_i$ at random
4: initialize $L \leftarrow \{1,\dots,n\}, R \leftarrow \{1,\dots,2n\}, E \leftarrow \emptyset$
5: **for** $i \leftarrow 1,\dots,n$ **do**
6: **for** $j \leftarrow 1,\dots,k$ **do**
7: compute $j$-th candidate entry $v \leftarrow \mathsf{PRF}_{\mathtt{sk}_j}(A[i].k) + j \cdot b$
8: establish an edge $e \leftarrow (i,v)$ and $E \leftarrow E \cup \{e\}$
9: set $\tau \leftarrow \max(3\log n + 1, 30)$
10: find a matching $M \leftarrow \mathsf{omatcher}((L \cup R, E), \tau)$
11: parse $M$ as $(i,j)$, *i.e.*, a bin placement scheme $P \leftarrow (A[i], j)$
12: obliviously place items in $A$ into $2n$ table entries $T$ w.r.t. $P$
13: initialize a counter for dummy lookups $\mathtt{ctr} \leftarrow 0$

lookup$(k')$ : ▷ *assume that real keys are all possitive*
1: set $\mathtt{ret} \leftarrow \perp$
2: set $k' \leftarrow -(\mathtt{ctr}+1)$ and increment $\mathtt{ctr}$ if $k' = \perp$
3: compute entry indices $\mathtt{id}_i \leftarrow \mathsf{PRF}_{\mathtt{sk}_i}(k'), \forall i \in \{1,\dots,k\}$
4: scan $T[\mathtt{id}_i], \forall i \in [k]$ and obliviously select the entry if $T[\mathtt{id}_i].k = k'$, and mark the matched one as dummy
5: **return** $\mathtt{ret}$

extract() :
1: obliviously compact the table $T$ and truncate to its half
2: obliviously shuffle $T$ uniformly at random
3: **return** $T$

The key point in obliviously building a stashless Cuckoo hash table lies in bounding the iteration times $\tau$ (line 9) for omatcher, *i.e.*, the maximum possible length of alternating paths. We adapt the proof technique in [31], establishing a bound of $\log n$ for somewhat large $n$ as shown in Lems. 3 and 4. We also impose an explicit lower bound of 30 for $\tau$ to handle scenarios involving small $n$.

**Lemma 3** (derived from [31])**.** *Given the bipartite graph $G = (L \cup R, E)$ setup as in Alg. 2, for $\mu \in [1,2)$ and any $Y \subset R$ s.t. $|Y| = \mu n$, the following inequality holds with probability at least $1 - e^{-\Omega(kn)}$:*

$$|\Gamma(Y)| \geq \frac{\mu}{2} n,$$

*where $\Gamma(Y)$ denotes neighbors of $Y$.*

*Proof.* Let $n$ denote the number of left vertices. For $\mu \in [1,2)$ and any $X \subset L$ with $|X| \geq (1-\mu/2)n$, we first prove that $|\Gamma(X)| \geq (2-\mu)n$ with all but negligible probability $e^{-\Omega(kn)}$ using same proof techniques in [106]. We then prove the lemma by contradiction. Denote $X \leftarrow \Gamma(Y)|$. If $|X| < \frac{\mu}{2}n$, we will have $|X'| \geq (1-\mu/2)n$ with $X' \leftarrow L \setminus X$. Consequently, the number of neighbors of $X'$ must exceed $(2-\mu)n$, which implies that some vertices in $X'$ have neighbors in $Y$, contradicting to the definition of $X'$. □

**Lemma 4.** *The maximum length of alternating paths in the random bipartite graph setup in Alg. 2 is bounded by $\log n$ with probability at least $1 - e^{-\Omega(kn)}$.*

*Proof.* We first assume the existence of a left-perfect matching $M$. Let $R_0$ be the set of free vertices in $R$, we have $|R_0| = n$ as $|R| = 2n$ and $|M| = n$. We expand $R_\lambda$ for $\lambda \geq 0$ as follows. Let $L_{\lambda+1} \leftarrow \Gamma(Y_\lambda)$ be the neighbors of $Y_\lambda$ in $L$, and $R_{\lambda+1} \leftarrow \Gamma(L_{\lambda+1}) \cup R_0$ be the neighbors of $L_{\lambda+1}$ together with free vertices, we claim that $|R_\lambda| \geq (2 - 2^{-\lambda})n$, which can be derived from Lem. 3 as $|R_{\lambda+1}| \geq n + |L_{\lambda+1}| \geq n + |R_\lambda|/2$ and $|R_0| = n$. Namely, we have $|L_{\lambda+1}| \geq (1 - 2^{-\lambda-1})n$. Taking $\lambda = \log n$ implies $|L_{\lambda+1}| \geq n - 1$. In brief, we can cover all vertices in $L$ in $\log n + 1$ rounds of expansion, implying that the length of alternating paths is at most $\log n$. □

**Theorem 2.** *Let $\beta$ be the size of the input data blocks. Alg. 2 obliviously implements $\mathcal{F}_{\mathsf{HT}}$ (formally defined in Func. A.2) for non-recurrent lookups with $O(nk\log^3 n + \beta n \log^2 n)$ building time, $O(k)$ lookup time, and $O(n\log^2 n)$ extraction time. In specific, the build process fails with a negligible probability $n^{-\Theta(k^2)} + e^{-O(kn)}$.*

*Proof.* By Lem. 2, build fails with $n^{-\Theta(k^2)}$ probability due to the absence of a left-perfect matching. Lem. 4 shows that we can find a left-perfect matching with all but $1 - e^{-O(kn)}$ probability. We hence obtained the claimed failure probability by combining them. Besides, as oblivious bipartite matching takes $O(nk\log^3 n)$ time and the oblivious bin placement runs in $O(\beta n \log^2 n)$ time, we then obtain the claimed building time complexity. We build a simulator for the build process as follows. Given $n := |A|$ and $k$ as inputs, the simulator samples $n$ key-value pairs with distinct keys as $A'$, and simulates omatcher and oblivious bin placement using their respective simulators. The remaining steps are identical to the build process. The parameter for the omatcher simulator, $\tau$, depends only on the number of input items $n$, which is public to adversaries. We can hence obliviously simulate the build process.

The lookup process is oblivious provided that the lookups are non-recurrent, and it runs in $O(k)$ time as it linearly scans $k$ entries. The running time of extract is dominated by an $O(n\log^2 n)$ oshuffling. It is correct as at most half real data will be preserved by compaction. We build a simulator for the extract process that obliviously simulates its functionality by substituting ocompaction and oshuffling with their respective simulators, while acknowledging that "truncating-to-half" operation is publicly known to adversaries. Note that substituting ocompaction with its simulator introduces a negligible failure probability [8]. □

**Remarks on efficiency.** We observe that the edges introduced by the construction of Cuckoo hash tables can be safely divided into $k$ groups, each containing $n$ edges, as it is public to adversaries. We can hence further optimize the oblivious matching algorithm by obliviously sorting each group of $n$ edges individually rather than collectively. This optimization slightly reduces the complexity of the omatcher algorithm from $O(nk\log^2 nk)$ to $O(kn\log^2 n)$.

Moreover, our proposed scheme not only enhances the lookup time compared to existing oblivious Cuckoo hash tables with a stash, but also reduces the building time by minimizing the rounds of required oblivious sorting in practice. Another interesting property is that the running time of omatcher, one of the most time-consuming components in building the table, is independent of the size of the input data blocks. This independence offers a notable performance advantage when processing large data blocks.

### 3.3 Oblivious Two-Tier Hash

We follow the same framework as the ones in OptORAMa [6] and FutORAMa [8]. We omit its full formal description and focus only on our modifications as they are quite modular. As shown in Fig. 2c and discussed in §1, the key intuition behind is that we are safe to place data blocks into hash table buckets in a non-oblivious (*i.e.*, efficient) way, as long as 1) they are randomly shuffled that can be efficiently accomplished by oblivious interspersion, and 2) a secret proportion, $\sim\varepsilon$, of each major hash bucket/table is obliviously relocated to a secondary hash table, referred to as the overflow pile.

Our first modification is to implement both the major hash tables and the secondary hash table using our tailored hash tables, whereas the original designs employ oblivious Cuckoo hash tables with stashes. The specific hash scheme to be used in our two-tier hash tables will be determined, again, by our hash scheme planner as detailed in §4. Generally, the major hash tables use bucket hash, while the secondary hash table employs oblivious stashless Cuckoo hash. We emphasize that our Cuckoo hash scheme offers remarkable performance advantages for the overflow pile, as the number of accesses, $n$, to the overflow pile is several times greater than the amount of data, $\varepsilon\cdot n$, it contains, and our Cuckoo hash scheme excels in lookup efficiency. Concretely, it scans only several table entries, whereas a bucket hash needs to access dozens to hundreds of entries.

The second optimization is to adaptively select the "overflow rate" $\varepsilon$ instead of using a fixed value. Although this does not yield better asymptotic complexity, it does enhance the actual runtime efficiency. Note that we must keep $\exp(-\varepsilon^2 Z/16)$ below a concretely negligible threshold [8], where $Z$ denotes the capacity of a major hash table. We then have $Z = C\cdot\varepsilon^{-2}$ where $C$ is a constant (*e.g.*, 1024) to meet this condition. As a meaningful $Z$ should be less than $n$, the possible values for $\varepsilon$ are tightly constrained within $\left[\sqrt{C/n}, 1\right)$, particularly if we limit $\varepsilon$ to powers of two. Thus, a brute-force search is sufficient to find $\varepsilon$ enjoying optimal running time.

**Time complexity.** We assume that major hash tables adopt bucket hash and the overflow pile employs oblivious stashless Cuckoo hash. The building time for the two tiers are $O\left(n\log^2\left(\varepsilon^{-2}C\right)\right)$ and $O\left(\varepsilon n\cdot\log^3(\varepsilon n)\right)$, respectively.Note that two terms exhibit opposite monotonicities, *i.e.*, as $\varepsilon$ decreases, it takes more time to build the major hash tables while less time for the overflow pile, and vice versa. The optimal value of $\varepsilon$ depends on the specific block size and computing environments. For the sake of brevity, we assume $\varepsilon = \log^{-3} n$, resulting in a total build time of $O(n\log^2\log n)$, which is asymptotically equivalent to that of the bucket hash, but offers better concrete efficiency for large $n$. The lookup process, dominated by a major hash table's lookup, runs in $O(\ell)$ time, where $\ell$ is the bucket size of major hash tables.

We conclude this section with Tab. 1 that provides a clearer illustration of the hash tables used in $\mathtt{H_2O_2RAM}$.

Table 1: Summary of hash schemes used in $\mathtt{H_2O_2RAM}$. †

| hash scheme | build & extraction time ‡ | lookup time | suitable scenarios |
|---|---|---|---|
| linear scan | $O(n)$ | | very small $n$ |
| bucket hash §3.1 | $O(n\log^2 n)$ | $O(\ell)$ | small to moderate $n$ |
| stashless cuckoo hash §3.2 | $O(n\log^3 n)$ | $O(1)$ | $n < t$ |
| two-tier hash §3.3 | $O(n\log^2\log n)$ | $O(\ell)$ | large $n$ |

† $n$: the size of a hash table, $\ell$: bucket size, $t$: the number of lookups performed during its lifetime.
‡ Adopting an $O(n\log n)$ osort algorithm [86] saves a $\log n$ factor for this column.

## 4 $\mathtt{H_2O_2RAM}$

We are now prepared to present $\mathtt{H_2O_2RAM}$, a high-performance hierarchical doubly oblivious RAM. As outlined in §1, $\mathtt{H_2O_2RAM}$ builds on the hierarchical ORAM with its framework introduced in §2.2. Due to space constraints, we defer the detailed description of $\mathtt{H_2O_2RAM}$ and the proof of Thm. 3, which formalizes the security of $\mathtt{H_2O_2RAM}$, to Appendix B.

**Theorem 3.** $\mathtt{H_2O_2RAM}$*, formally described in Alg. 5, obliviously implements ORAM functionality* $\mathcal{F}_{\mathsf{ORAM}}$ *(defined in Func. A.1) with a negligible error probability.*

**Comparison with FutORAMa [8].** Hierarchical ORAM (HORAM), despite its long-standing history, has garnered more theoretical interest than practical adoption compared to the tree-based framework. FutORAMa [8] is the first to make the hierarchical framework practically viable. Though

our work builds upon the HORAM framework, it differs from FutORAMa in the following aspects.

The key difference lies in the threat model. FutORAMa operates in a classic client-server setting, where a client outsources its confidential data to an untrusted server and aims to hide data access patterns. This approach demands intensive network communications, making its performance highly dependent on network conditions. To make HORAM practical, FutORAMa introduces a key relaxation, enabling the client to manage a sublinear-sized private memory instead of being restricted to a constant-sized one. This allows the client to download a batch of data from the server, process it locally, and then send the processed data back, thereby reducing frequent network interactions. In contrast, our work aims to address the limitation of TEEs in exposing memory access patterns. In TEEs, data access is not limited by the network conditions. However, most TEEs lack inherently oblivious memory, leaving us still constrained by the assumption of "constant-sized client memory" (*i.e.*, registers within the CPU). FutORAMa is hence incompatible with our setting.

Besides, FutORAMa and our work target different optimization aspects, driven by the distinct threat models outlined above. The optimizations in FutORAMa focus on efficiently utilizing sublinear-sized private memory while maintaining its reasonable size. Within this memory, *non-oblivious* plain operations are performed. Our work, on the other hand, concentrates on optimizing the small- and moderate-sized oblivious hash tables due to the lack of privileged memory in FutORAMa. Note that while the hash tables we focus on are not large in size, their frequently accessed nature has a critical impact on the overall system performance, making it essential to optimize their efficiency. To this end, we propose three hash schemes adapted and refined from existing works and develop a planner to determine the appropriate hash scheme for each level. $\mathtt{H_2O_2RAM}$ thus delivers substantial concrete performance improvements, despite having an asymptotic running time on par with that of the state-of-the-art approaches.

**Hash scheme planner.** As discussed in §1 and §3, it is impractical for us to calculate exact thresholds to identify the optimal hash scheme across various scenarios. We hence develop a hash scheme planner to help achieve this goal. First, it relies on the parameter selectors for each hash scheme, as outlined in §3, to optimize the performance of these schemes. It then selects the best one. Of course, we could leverage the insights shown in Tab. 1 to minimize unnecessary comparisons. Namely, stashless Cuckoo hash will be considered a candidate only when used in the overflow pile of a two-tier hash table. When a two-tier hash table outperforms a bucket hash table, subsequent levels will exclude the bucket hash scheme from consideration.

**Amortized access time.** Denote the capacity of $\mathtt{H_2O_2RAM}$ as $N$, w.l.o.g., we assume the bucket size $\ell = O(\log N)$, *i.e.*, a lookup runs in $O(\log N)$ time. For $t$ accesses, we perform $O(t \log N)$ lookups, and rebuild the $i$-th level $\lceil t/2^i \rceil$ times. Therefore, $t$ accesses require $O\left(t \log^2 N\right) + \sum_{i=1}^{\log N} \lceil \frac{t}{2^i} \rceil O\left(2^i \log^2 \log 2^i\right)$ time. The amortized access time is hence $\tilde{O}\left(\log^2 N\right)$. We thus achieve a complexity comparable to tree-based $O_2$RAM designs [18, 70, 96], fulfilling the premise of our insight discussed in §1. We also list an asymptotic comparison in Tab. 2.

Table 2: Asymptotic comparison of existing $O_2$RAM schemes, $N$ denotes the capacity.

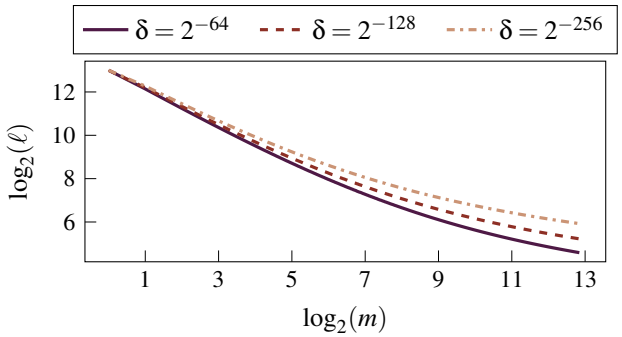
| Scheme | Access time |
|---|---|
| Oblix [70] & GraphOS [18] | $O\left(N \log^3 N\right)$ |
| ENIGMAP [96] | $\tilde{O}(N \log^2 N)$ |
| $\mathtt{H_2O_2RAM}$ | $\tilde{O}(N \log^2 N)$ |

**Extension for map support.** As noted in §2, a RAM of capacity $N$ is indexed by the logical address space $[N]$. However, in typical key-value map applications, keys are not confined to the range $[1,N]$. A naïve solution to support oblivious map is to implement a binary search tree with its backend data managed by ORAM with appropriate padding. While in our design, it is fortunate that all levels are implemented as (oblivious) hash tables, making it irrelevant whether the keys fall within the range $[1,N]$ for correctness or security. The only requirement is that the keys are hashed in a cryptographically secure and oblivious manner, and correct implementation of such hash functions is considered orthogonal to our work.

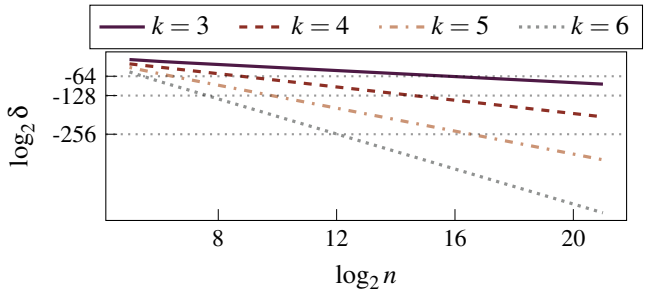
**Implementation.** We implement $\mathtt{H_2O_2RAM}$ in C++ on https://doi.org/10.5281/zenodo.14648338, and to the best of our knowledge, it is the first practical hierarchical-based $O_2$RAM implementation. Note that FutORAMa [8] is implemented in Python and serves more as a simulation. Similar to FutORAMa, the initial level in $\mathtt{H_2O_2RAM}$ does not begin with a capacity of two but a small linear scan level of capacity $256 \sim 1024$, depending on the actual block sizes. The rationale is based on the fact that multiple linear scan levels are less efficient than a single combined one.

## 5 Experimental Evaluation

**Experimental setup.** We implement $\mathtt{H_2O_2RAM}$ with C++20, and measure the execution time using Google Benchmark [37]. The experimental setup involves a physical server powered by a 96-core Intel Xeon(R) Platinum 8457C processor and 2TB of RAM. Each core operates at a frequency of 2.6 GHz. This server utilizes Debian GNU/Linux 10 as its operating system, with the kernel upgraded to version 5.15.120+. Using software packages provided by Intel Trusted Domain Extension (TDX) [44], a confidential virtual machine is configured with 64 cores, 512 GB RAM, and ubuntu-20.04, which has the same version kernel as the host. We use both OpenMP [10] and Intel oneTBB [45] libraries to paral-




(a) Bucket size $\ell$ versus the number of buckets $m$ required to achieve an overflow probability $\delta$ in bucket hash with 8192 data blocks.




(b) Concrete failure probabilities of stashless Cuckoo hash, $\delta$, for various hash table sizes $n$ and numbers of hash functions, $k$.

Figure 5: More precise overflow/failure probabilities.

lelize $H_2O_2$RAM. All TEE experiments discussed throughout our work are conducted within this trusted VM. To better measure the amortized access time of $H_2O_2$RAM, we access both $O_2$RAM and $O_2$MAP for $n$ times and report the average access time.

**Compared approaches.** We compared $O_2$MAP access times with EnigMap [70], and doubly oblivious single-source shortest path computation with GraphOS [18]. Both baseline approaches stand for state-of-the-art tree-based doubly oblivious solutions. Since TDX provides ample protected memory, we removed the manual encryption and decryption components from the EnigMap implementation [1], *i.e.*, relying solely on TDX's encryption mechanisms. As a result, our experiments are expected to perform faster than those on SGX (note that our CPU is more powerful). However, our reproduced results for EnigMap are approximately ten times slower than those reported in [96]. Despite this discrepancy, we confirm that their results still significantly surpass other state-of-the-art methods and do not undermine their conclusions presented in their abstract. Regarding GraphOS [18], we adopt the results reported in the "Gr-V0.13E" line of Figure 6c in their paper rather than re-running their implementation in our environment.

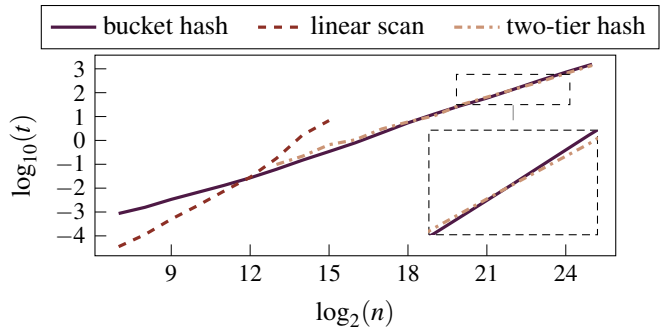
## 5.1 Experiment Stages

**Concrete overflow/failure probability.** As discussed in §3, we leverage more precise formulas together with numeric methods and high-precision math libraries to calculate concrete error probabilities, allowing a selection of more optimal parameters. Fig. 5a shows tight bucket sizes to ensure a (concretely) negligible overflow probability. We note that our method yields notably smaller bucket sizes compared to general yet loose bounds, such as $e \log_2 n$ [21] and 267 [8]. As shown in Fig. 5a, adding just one more hash function to the basic Cuckoo hash scheme is sufficient to achieve a relatively low failure probability $2^{-64}$ for hash tables larger than $2^{15}$. For smaller hash tables and stronger security (*i.e.*, lower failure probabilities), 4 to 6 hash functions suffice.

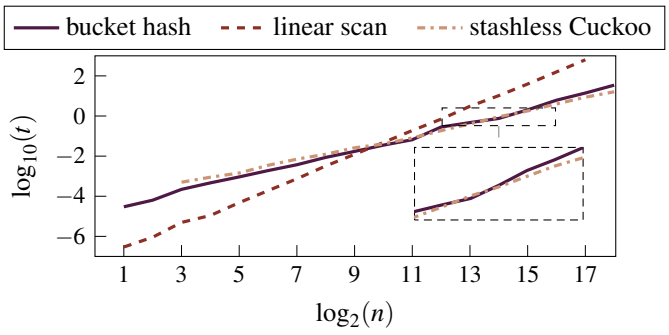
**Performance of tailored hash schemes and their optimal use cases.** As shown in Fig. 6, linear scan performs best when dealing with dozens to hundreds of input data in all cases. Bucket hash tables outperform the other methods for medium-sized input data. While for handling large input data, two-tier hash tables surpass bucket hash tables by 10% $\sim$ 30% if the number of lookups matches the input data size, as shown in Fig. 6a. In contrast, stashless Cuckoo hash tables provide over a 110% speedup compared to bucket hash tables when the number of lookups is 128 times the input data size, as shown in Fig. 6b. Note that some lines in the figure appear close to each other because the y-axis is on a logarithmic scale. The above results are consistent with our analysis in Tab. 1.

**Doubly oblivious RAM.** We then evaluate the performance of $H_2O_2$RAM in different scenarios. We generate $n$ data blocks with keys from 0 to $n-1$ and values of random bits. These blocks are then randomly shuffled to permute the data, upon which $H_2O_2$RAM is built. We also amortize the building time across $n$ data accesses. As shown in Fig. 7a, the amortized access time for $H_2O_2$RAM increases polylogarithmically with its capacity $n$, and increases approximately linearly with the data block size. Fig. 7b shows that parallelization effectively improves the access efficiency of $H_2O_2$RAM with performance gains observed up to 16 threads. Note that our implementation may achieve only suboptimal parallelization due to our limited expertise in this area, suggesting potential for further performance improvements.

**Doubly oblivious tasks.** We also evaluate the performance of $H_2O_2$RAM when applied in the computations of single-source shortest paths and the key-value map data structure, and draw a comparison with GraphOS [18] and EnigMap [96], resp. As shown in Fig. 8a, our solution reduces the Get operation time of a map by a factor of 100× to 997× compared to EnigMap [96]. In the single-source shortest path (SSSP) task, where we use the results for GraphOS directly from the "Gr-V0.13E" line in Figure 6c of their paper [18], $H_2O_2$RAM achieves substantial computation time improvements, ranging from 76.6× to 142.3×. For instance, on a graph with $|G| = 2^{18}$, $H_2O_2$RAM reduces the total processing time from

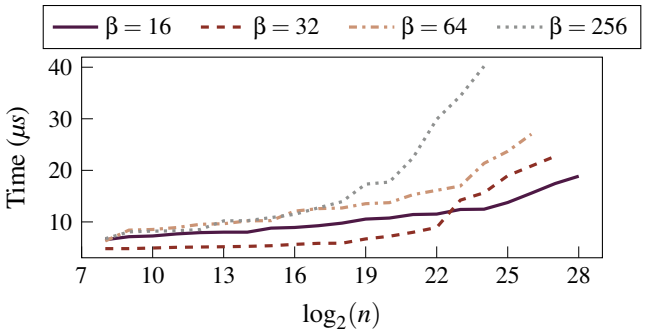



(a) Each hash table is accessed $n$ times. Linear scan terminates at $2^{15}$ due to its quadratic growth in running time. Two-tier hash begins at $2^{13}$ as it requires a minimum input size.

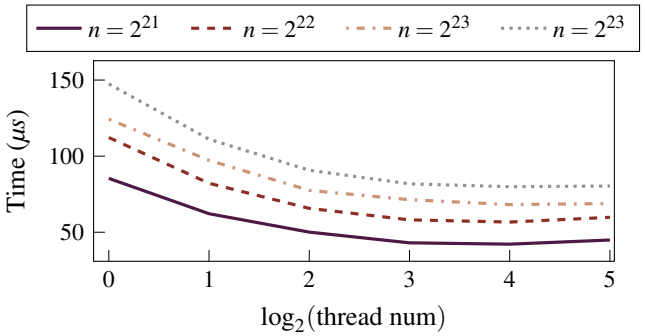



(b) Each hash table is accessed $128n$ times.

Figure 6: Overall running time $t$ (in seconds) of different hashing schemes. The block size is 256B.




(a) Performance of $H_2O_2$RAM under various input data sizes and data block sizes $\beta$.




(b) Performance of $H_2O_2$RAM under different degrees of parallelization and input data sizes.

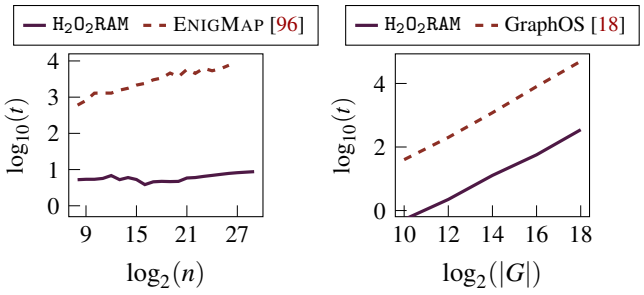
Figure 7: Amortized access time of $H_2O_2$RAM across different problem sizes.

approximately 13 hours (as reported in [18]) to less than 6 minutes (349 seconds). While the performance gap is less pronounced than that shown in Fig. 8a, this is primarily due to the relatively modest graph sizes used in our experiments. For larger graphs, GraphOS would require several days or even weeks to complete the computation, further underscoring the scalability advantages of $H_2O_2$RAM.

In addition to better running time, $H_2O_2$RAM also enjoys better memory space overhead. For instance, with $2^{22}/2^{23}$ blocks of 16 data blocks (a total of 64/128 MB of data), $H_2O_2$RAM requires 0.82/1.63 GB of memory, whereas EnigMap consumes 4.75/72.3 GB memory (*i.e.*, $H_2O_2$RAM reduces memory overhead by a factor of 5.79 to 44.36). Note that EnigMap experiences such an "exploded" memory consumption issue due to its packing strategies, which are designed for better locality, an inherent advantage enjoyed by $H_2O_2$RAM.

## 6 Related Work

**Oblivious RAM.** Since the seminal work in [33, 34], ORAM and its variants have attracted widespread interest in many applications, including but not limited to cloud computing [9, 18, 29, 32, 75, 93, 94, 102], multi-party computation [62, 64, 74, 99, 107], and secure processor designs [28, 61, 65, 83]. From a theoretical perspective, an $\Omega(\log n)$ lower bound shown in the very first study has been extended to many different settings [11, 16, 48, 51, 53, 54, 54, 100]. While it took decades to develop solutions [6, 7] that meet this lower bound, most of the efforts [19, 52, 77, 78, 95, 98] over this span were based on the hierarchical framework. However, a solution that is both asymptotically optimal and practically efficient, without relying on additional assumptions, remains unknown. On the other hand, Path ORAM [95], a simple yet powerful tree-based design, has opened up numerous practical applications for ORAM. Afterwards, a variety of optimized variants [9, 18, 24, 70, 74] have been developed for different application scenarios. The most relevant to us among these is ORAM with secure hardware. Shi [88] proposed a novel doubly oblivious heap and other data structures. ZeroTrace [84] offers constructions based on Path ORAM and Circuit ORAM [98], which is soon outperformed by Oblix [70]. A very recent work, GraphOS [18], further optimizes this design and offers several doubly oblivious graph algorithms. ENIGMAP [96] uses external-memory algorithms to op-




(a) Get operation time $t$ (in microseconds) of a map, where $n$ denotes its capacity.

(b) Single-source shortest path computation time $t$ (in seconds). The graph size $G$ is the sum of the vertices count $|V|$ and the edge count $|E|$ with $|E| = 4|V|$.

Figure 8: Comparison of $H_2O_2$RAM and EnigMap [96] / GraphOS [18]. The results for GraphOS are taken from the "Gr-V0.13E" line in Figure 6c of their paper.

timize oblivious accesses. Obliviate [2], MOSE [42], POSUP [41], Shroud [63], and Snoopy [24] adopt different optimizations for oblivious storage systems. Phantom [65], GhostRider [61], and Tiny ORAM [30] use FPGA as the backend trusted hardware.

**Oblivious primitives.** Our work also heavily relies on various oblivious operations and primitives. T-SGX [89] focuses on eradicating controlled-channel attacks. ObliVM [62], though relying on multiple noncolluding servers, proposed several operators for general oblivious execution. Obfuscuro [3], Klotski [108], and Obelix [101] stand for the most advanced approaches to oblivious execution of arbitrary code. Sinha et al. [91] implement an efficient compiler to enforce page-access obliviousness for a type and memory-safe languages. ObliCheck [92] can efficiently verify whether an algorithm is indeed oblivious. In addition, there have been significant recent advancements in oblivious sorting, shuffling, and compaction algorithms [85, 86]. In particular, Bitonic sorting/shuffling has long been considered the most concretely efficient algorithm, but it has been consistently outperformed by [85] and its subsequent follow-up [86]. However, such designs rely on an expensive offline preprocessing stage that generates some pseudorandomness, which makes it inappropriate for our design, as $H_2O_2$RAM frequently invokes osort/oshuffling.

**Relaxed oblivious designs.** Designs that trade full obliviousness for better performance are also prevalent. Examples include straightforward relaxations such as page-level obliviousness [81, 90, 91, 97] and the existence of an inherently oblivious private cache [27, 72]. Grubbs et al. [40] and Maiyya et al. [66] introduce relaxed notions of obliviousness in the context of key-value stores. In addition, differentially oblivious designs [20, 38, 79, 80, 103, 109], which bring the ideology of differential privacy, provide well-structured security notions.

## 7 Conclusion

In this work, we take a step forward in the practical application of hierarchical ORAM with Trusted Execution Environments, further underscoring the potential of the hierarchical ORAM framework. Our design, $H_2O_2$RAM, offers a general and efficient approach for executing algorithms that require the protection of access patterns. Moreover, the components we introduced, such as the oblivious bipartite matching and the oblivious stashless Cuckoo hash table, are of independent interest. We have also implemented and open-sourced $H_2O_2$RAM, and conducted empirical evaluations in various scenarios to show its concrete efficiency. The results show that $H_2O_2$RAM surpasses state-of-the-art designs by up to $\sim 10^3\times$ in running time and by $44\times$ in space consumption.

## Acknowledgements

We appreciate the detailed and valuable feedback provided by our anonymous reviewers. We thank Qingling Feng and Dian Chen for their discussions and for their spare time in running experiments. This work was supported in part by the Research Grants Council of Hong Kong under Grants CityU 11218322, 11219524, R6021-20F, R1012-21, RFS2122-1S04, C2004-21G, C1029-22G, C6015-23G, and N_CityU139/21 and in part by the Innovation and Technology Commission of Hong Kong (ITC) under Mainland-Hong Kong Joint Funding Scheme (MHKJFS) under Grant MHP/135/23. This work was also supported by the InnoHK initiative, the Government of the HKSAR, and the Laboratory for AI-Powered Financial Technologies (AIFT).

## Ethics Considerations

Our research adheres to stringent ethical standards to ensure the integrity and societal impact of the work presented. The core ethical considerations in this study include the protection of data privacy in cloud computing, and the broader implications of the technology we develop. We conducted our evaluation experiments using synthetic (randomly generated) data. Indeed, our objective is to transform programs into forms where their behavior is (pseudo-)independent of the input data, which may contain highly sensitive private information. Consequently, our work contributes to safeguarding user privacy in a broad spectrum of applications, including but not limited to searchable encryption, private contact discovery in end-to-end messaging, key transparency, and anonymous broadcasting/subscription platforms. However, we emphasize the following two major ethical issues that must be considered when applying our techniques:

1. Our approach relies heavily on redundant computations to safeguard data privacy, potentially resulting in the waste use of computational resources and, more critically, increased energy consumption. For instance, we observed that our server was operating at nearly full power during the experiments, while non-privacy-preserving solutions were merely accessing random addresses. Therefore, it is essential to evaluate whether such a high level of protection is necessary when designing a privacy-preserving application. Blindly applying such techniques will definitely lead to significant energy/resource waste and environmental pollution.

2. One important application of our work is to ensure anonymity. Maintaining online anonymity is a fundamental right for everyone. However, anonymity can occasionally be associated with hate speech or criminal activities. Therefore, those using our technology should also consider how to prevent inappropriate dissemination of offensive content or information.

In short, our work primarily focuses on privacy-preserving applications in cloud computing; however, special attention must still be given to the two aforementioned issues when applying this technology.

## Open Science Compliance

In compliance with the Open Science Policy adopted by USENIX and other leading research communities, we commit to sharing our research artifacts in a way that promotes transparency, reproducibility, and accessibility. We share our code on https://doi.org/10.5281/zenodo.14648338 with clear instructions on reproducing the experiments.

## References


[1] Tinoco Afonso, Gao Sixiang, and Shi Elaine. EnigMap: Oblivious data structure library. URL https://github.com/odslib/EnigMap.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for Intel SGX. In *NDSS*, 2018.

[3] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on Intel SGX. In *NDSS*, 2019.

[4] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. URL https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[5] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*, 2020.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. *JACM*, 70(1):1–70, 2022.

[7] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel RAM. In *SODA*, 2022.

[8] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. FutORAMa: A concretely efficient hierarchical oblivious RAM. In *CCS*, 2023.

[9] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*, 2015.

[10] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. URL https://www.openmp.org/.

[11] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[13] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 2017.

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[16] David Cash, Andrew Drucker, and Alexander Hoover. A lower bound for one-round oblivious RAM. In *TCC*, 2020.

[17] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *NDSS*, 2019.

[18] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. GraphOS: Towards oblivious graph processing. In *VLDB*, 2023.

[19] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *ASIACRYPT*, 2017.

[20] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467. SIAM, 2019.

[21] Shuchi Chawla. Lecture 7: Randomized load balancing and hashing, 2009.

[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. *IEEE S&P*, 18(3):28–37, 2020.

[23] Alibaba Cloud. TEE-based confidential computing. URL https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/tee-based-confidential-computing.

[24] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[25] Samuel Dittmer and Rafail Ostrovsky. Oblivious tight compaction in O(n) time with smaller constant. In *SCN*, 2020.

[26] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *CCS*, 2017.

[27] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious query processing for secure databases. *VLDB*, 13 (2):169–183, 2019.

[28] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[29] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Freecursive ORAM: Nearly free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, 2015.

[30] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious RAM controller. In *FCCM*, 2015.

[31] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *TOCS*, 38(2):229–248, 2005.

[32] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In *ACNS*, 2015.

[33] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43 (3):431–473, 1996.

[35] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[36] Google. Confidential computing, . URL https://cloud.google.com/security/products/confidential-computing?hl=en.

[37] Google. Google Benchmark, . URL https://github.com/google/benchmark.

[38] S. Dov Gordon, Jonathan Katz, Mingyu Liang, and Jiayu Xu. Spreading the privacy blanket: Differentially oblivious shuffling for differential privacy. In *ACNS*, 2022.

[39] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, 2016.

[40] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX*, 2020.

[41] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PETS*, 2019(1), 2019.

[42] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. Mose: Practical multi-user oblivious storage via secure enclaves. In *CODASPY*, 2020.

[43] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SICOMP*, 2(4):225–231, 1973.

[44] Intel. MKTME side channel impact on Intel TDX, . URL https://www.intel.com/content/www/us

/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html.

[45] Intel. Intel oneAPI threading building blocks (oneTBB), . URL https://github.com/oneapi-src/oneTBB.

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

[47] Anati Ittai, Gueron Shay, Johnson Simon, and Scarlata Vincent. Innovative technology for cpu based attestation and sealing. Online at https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html.

[48] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *SODA*, 2019.

[49] Jack Kiefer. Sequential minimax search for a maximum. *AMS*, 4(3):502–506, 1953.

[50] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SICOMP*, 39(4):1543–1561, 2010.

[51] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious RAM (for all parameters). In *CRYPTO*, 2021.

[52] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[53] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, 2018.

[54] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. In *SODA*, 2020.

[55] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security*, 2020.

[56] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 2017.

[57] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent Cuckoo hashing. In *EuroSys*, 2014.

[58] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *SODA*, 2019.

[59] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: software-based power side-channel attacks on x86. In *S&P*, 2021.

[60] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[61] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[62] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S&P*, 2015.

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

[64] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, 2013.

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

[66] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. *PACMMOD*, 1(4):266:1–266:25, 2023.

[67] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. In *USENIX Security*, 2017.

[68] John P Mechalas and Benjamin J Odom. Intel® Software Guard extensions tutorial series: Part 1, Intel SGX. Online at https://www.intel.com/content/www/us/en/developer/articles/training/intel-software-guard-extensions-tutorial-part-1-foundation.html.

[69] Microsoft. Azure confidential computing. URL https://learn.microsoft.com/en-us/azure/confidential-computing/.

[70] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *S&P*, 2018.

[71] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *S&P*, 2020.

[72] Chandrasekhar Nagarajan, Ali Shafiee, Rajeev Balasubramonian, and Mohit Tiwari. ρ: Relaxed hierarchical ORAM. In *ASPLOS*, 2019.

[73] Daniel Noble. Explicit, closed-form, general bounds for Cuckoo hashing with a stash. *Cryptology ePrint Archive*, 2021.

[74] Daniel Noble, Brett Hemenway, and Rafail Ostrovsky. MetaDORAM: Breaking the log-overhead information theoretic barrier. In *ECCC*, 2024.

[75] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[76] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[77] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, 2018.

[78] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[79] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. Adore: Differentially oblivious relational database operators. In *VLDB*, 2022.

[80] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. Doquet: Differentially oblivious range and join queries with private data structures. In *VLDB*, 2023.

[81] Maan Haj Rachid, Ryan D. Riley, and Qutaibah M. Malluhi. Enclave-based oblivious RAM using Intel's SGX. *Computers & Security*, 91:101711, 2020.

[82] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[83] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, 2013.

[84] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS*, 2018.

[85] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *CCS*, 2022.

[86] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-On/Waks-Off: Fast oblivious offline/online shuffling and sorting with Waksman networks. In *CCS*, 2023.

[87] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-shield: Enabling address space layout randomization for SGX programs. In *NDSS*, 2017.

[88] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *S&P*, 2020.

[89] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[90] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS*, 2016.

[91] Rohit Sinha, Sriram Rajamani, and Sanjit A Seshia. A compiler and verifier for page access oblivious computation. In *FSE*, 2017.

[92] Jeongseok Son, Griffin Prechter, Rishabh Poddar, Raluca Ada Popa, and Koushik Sen. ObliCheck: Efficient verification of oblivious algorithms with unobservable state. In *USENIX Security*, 2021.

[93] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious distributed cloud data store. In *NDSS*, 2013.

[94] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[95] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. *JACM*, 65(4):1–26, 2018.

[96] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap: External-memory oblivious map for secure enclaves. In *USENIX Security*, 2023.

[97] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques. In *DIMVA*, 2017.

[98] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, 2015.

[99] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In *CCS*, 2014.

[100] Mor Weiss and Daniel Wichs. Is there an oblivious RAM lower bound for online reads? *Journal of Cryptology*, 34(3):18, 2021.

[101] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *S&P*, 2024.

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

[103] Pengfei Wu, Jianting Ning, Xinyi Huang, and Joseph K. Liu. Differentially oblivious two-party pattern matching with sublinear round complexity. *TDSC*, 20(5): 4101–4117, 2023.

[104] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. Leakage-abuse attacks against forward and backward private searchable symmetric encryption. In *CCS*, 2023.

[105] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, 2015.

[106] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO*, 2023.

[107] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *S&P*, 2016.

[108] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. Klotski: Efficient obfuscated execution against controlled-channel attacks. In *ASPLOS*, 2020.

[109] Leqian Zheng, Lei Xu, Cong Wang, Sheng Wang, Yuke Hu, Zhan Qin, Feifei Li, and Kui Ren. SWAT: A system-wide approach to tunable leakage mitigation in encrypted data stores. In *VLDB*, 2024.

[110] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

[111] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGOPS Operating Systems Review*, 38(5), 2004.

# A Preliminaries (Cont.)

All functionalities that we formalize here describe only the input-output behavior of the primitives rather than their implementation details. The ideal functionality of an ORAM, as shown in Func. A.1, implements logical memory.

**Functionality A.1** Oblivious RAM $\mathcal{F}_{\mathsf{ORAM}}$:

$\mathcal{F}_{\mathsf{ORAM}}$ reactively holds $n$ $w$-bit memory words $A[1,\ldots,n]$, in which $A[\mathtt{addr}]$ is initialized as $0^w, \forall \mathtt{addr} \in [n]$.

- $\mathcal{F}_{\mathsf{ORAM}}.\mathsf{access}(\mathtt{op}, \mathtt{addr}, v)$: ▷ $\mathtt{op} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathtt{addr} \in [n]$, $v \in \{0,1\}^w$
  1: **if** $\mathtt{op} = \mathtt{read}$ **then** $\mathtt{res} \leftarrow A[\mathtt{addr}]$
  2: **else** $A[\mathtt{addr}] \leftarrow v, \mathtt{res} \leftarrow v$ ▷ $\mathtt{op} = \mathtt{write}$
  3: **return** $\mathtt{res}$

The ideal functionality shown in Func. A.2 implements a hash table (also known as a dictionary) that reactively supports table build, key lookup, and data extraction operations.

**Functionality A.2** Oblivious Hash Table $\mathcal{F}_{\mathsf{HT}}$:

Denote a key pair as $(k,v) \in \{0,1\}^{\ell_k} \times \{0,1\}^{\ell_v}$. W.l.o.g., we assume that $\ell_k = O(w)$ and $\ell_v = O(w)$, *i.e.*, both the key and the value can be stored in $O(w)$ memory words.

- $\mathcal{F}_{\mathsf{HT}}.\mathsf{build}(A)$: ▷ *$A$ contains $n$ possibly dummy (i.e., $\perp$) key-value pairs with distinct keys*
  1: initialize its internal state $(A,T)$ with $T \leftarrow \emptyset$
  2: output nothing
- $\mathcal{F}_{\mathsf{HT}}.\mathsf{lookup}(k)$: ▷ $k \in \{0,1\}^{\ell_k} \cup \{\perp\}$
  1: **if** $k \neq \perp \wedge k \in T$ **then return** fail ▷ *$k$ is a recurrent lookup*
  2: **if** $k = \perp \vee k \notin A$ **then** $v' \leftarrow \perp$
  3: **else**
  4: $\quad v' \leftarrow v$, where $(k,v) \in A$
  5: $\quad T \leftarrow T \cup \{(k,v')\}$
  6: **return** $v'$
- $\mathcal{F}_{\mathsf{HT}}.\mathsf{extract}()$:
  1: **for** $(k,v) \in A$ **do**
  2: $\quad$ **if** $k \in T$ **then** $k \leftarrow \perp, v \leftarrow \perp$
  3: shuffle $A$ uniformly at random
  4: **return** $A$

We then formally define oblivious simulation of a (reactive) functionality $\mathcal{F}_{\mathsf{F}}$ in Def. 1. W.l.o.g., we assume that $\mathcal{F}_{\mathsf{F}}$ takes commands and input data of the form $(\mathtt{cmd}, \mathtt{inp})$, and produces an output $\mathtt{out}$, while probably maintaining some internal (secret) state. For a RAM machine $M_{\mathsf{F}}$ implementing $\mathcal{F}_{\mathsf{F}}$, the execution of $(\mathtt{cmd}, \mathtt{inp})$ will produce side-channel information $\mathtt{addrs}$ that indicates the memory addresses accessed during the process. We occasionally refer to certain parameters as "public", indicating that the simulator additionally takes these parameters as input and that it is acceptable for adversaries to have knowledge of them.

**Definition 1** (Oblivious Machine Implementing a Functionality $\mathcal{F}_{\mathsf{F}}$)**.** *A RAM machine $M_{\mathsf{F}}$* obliviously implements *the reactive functionality $\mathcal{F}_{\mathsf{F}}$ if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator* Sim*, such that the view of the adversary $\mathcal{A}$ in the following experiments $\mathsf{Expt}^{real,M_{\mathsf{F}}}_{\mathcal{A}}(1^\lambda)$ and $\mathsf{Expt}^{ideal,\mathcal{F}_{\mathsf{F}}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$ is computationally indistinguishable.*

$\mathsf{Expt}^{\text{real},M_{\mathsf{F}}}_{\mathcal{A}}(1^\lambda)$:

1: $(\mathtt{cmd}_1,\mathtt{inp}_1)\leftarrow\mathcal{A}(1^\lambda)$, $i\leftarrow 1$
2: **while** $\mathtt{cmd}_i\neq\perp$ **do**
3: $\quad(\mathtt{out}_i,\mathtt{addrs}_i)\leftarrow M_{\mathsf{F}}(1^\lambda,\mathtt{cmd}_i,\mathtt{inp}_i)$
4: $\quad(\mathtt{cmd}_{i+1},\mathtt{inp}_{i+1})\leftarrow\mathcal{A}(1^\lambda,\mathtt{out}_i,\mathtt{addrs}_i)$
5: $\quad i\leftarrow i+1$

$\mathsf{Expt}^{\text{ideal},\mathcal{F}_{\mathsf{F}}}_{\mathcal{A},\mathsf{Sim}}(1^\lambda)$:

1: $(\mathtt{cmd}_1,\mathtt{inp}_1)\leftarrow\mathcal{A}(1^\lambda), i\leftarrow 1$
2: **while** $\mathtt{cmd}_i\neq\perp$ **do**
3: $\quad\mathtt{out}_i\leftarrow\mathcal{F}_{\mathsf{F}}(\mathtt{cmd}_i,\mathtt{inp}_i)$, $\mathtt{addrs}_i\leftarrow\mathsf{Sim}(\mathtt{cmd}_i,1^\lambda)$
4: $\quad(\mathtt{cmd}_{i+1},\mathtt{inp}_{i+1})\leftarrow\mathcal{A}(1^\lambda,\mathtt{out}_i,\mathtt{addrs}_i)$
5: $\quad i\leftarrow i+1$

## B Details on $\mathtt{H_2O_2RAM}$

We provide the details of $\mathtt{H_2O_2RAM}$'s implementation and the proof of Thm. 3 here. Recall that $\mathtt{H_2O_2RAM}$ consists of $O(n)$ levels. Let $L:=\lceil\log n\rceil$ and $\ell$ be the threshold representing the highest level at which a linear scan performs optimally among all the hash schemes presented in Tab. 1. Each level $i\in\{\ell,\ldots,L\}$ is a tailored hash table $T_i$ with a capacity of $2^i$, where $T_\ell$ is specifically implemented as a plain array.

**Algorithm 5** $\mathtt{H_2O_2RAM}.\mathsf{access}(\mathtt{op},\mathtt{addr},v)$:

**Input:** $\mathtt{op}\in\{\mathtt{read},\mathtt{write}\}$, $\mathtt{addr}\in[n]$, $v\in\{0,1\}^w$
1: initialize $\mathtt{res}\leftarrow\perp$
2: **for** $i\in\{\ell,\ldots,L\}$ **do**
3: $\quad$**if** $T_i$ is empty **then** continue
4: $\quad$**if** $\mathtt{res}=\perp$ **then** $\mathtt{res}\leftarrow T_i.\mathsf{lookup}(\mathtt{addr})$
5: $\quad$**else** $T_i.\mathsf{lookup}(\perp)$
6: **if** $\mathtt{op}=\mathtt{write}$ **then** $\mathtt{res}\leftarrow v$
7: append $(\mathtt{addr},\mathtt{res})$ to $T_\ell$ ▷ *$T_\ell$ is a plain array*
8: **if** $T_\ell$ is full **then**
9: $\quad$let $i^*$ be the first empty level or $i^*=L$ if all levels are non-empty
10: $\quad$let $A\leftarrow T_\ell.\mathsf{extract}()\|\ldots\|T_{i^*-1}.\mathsf{extract}()$
11: $\quad$**if** $i^*=L$ **then** obliviously compact $A$ to its half
12: $\quad$obliviously intersperse (*i.e.*, shuffle) $A$
13: $\quad T_{i^*}.\mathsf{build}(A)$
14: **return** $\mathtt{res}$

For brevity, we omit precise negligible probabilities of certain oblivious building blocks, instead denoting them as $\delta_{\mathsf{oht}}$, $\delta_{\mathsf{oshuffle}}$, $\delta_{\mathsf{ocompact}}$, and $\delta_{\mathsf{ointersperse}}$ corresponding to the oblivious hash table, oblivious shuffle, oblivious compact, and oblivious intersperse operations, respectively. The mathematical forms of these values can be derived by examining their specific instantiations. For instaince, the ocompact implementation [8] yields $\delta_{\mathsf{ocompact}}=O\left(\frac{n}{\mathsf{polylog}\,n}e^{-\mathsf{polylog}\,n}\right)$. Thm. 2 shows that our oblivious stashless cuckoo hash has $\delta_{\mathsf{oht\_cuckoo}}=n^{-\mathsf{poly}(k^2)}+e^{-O(kn)}$ with $k=\omega(1)$.

Let $\mathsf{Sim}_{\mathsf{F}}$ denote the oblivious simulator of a functionality $\mathcal{F}_{\mathsf{F}}$, $n$, $L$, and $\ell$ be defined as the ones in $\mathtt{H_2O_2RAM}$. We build the oblivious simulator of $\mathtt{H_2O_2RAM}$ as shown in Sim. B.1.

To prove Thm. 3, the remaining task is to construct hybrid constructions between Alg. 5 (denoted as Construction 1) and Sim. B.1. Construction 2 is the same as Construction 1 except that lines 4 $\sim$ 5 are replaced by the line 4 in $\mathsf{Sim}_{\mathtt{H_2O_2RAM}}$, yielding an adversarial advantage of $\sum_{i\in\{\ell,\ldots,L\}}\delta_{\mathsf{HT}_i(\mathsf{lookup})}$. Construction 3 is the same as Construction 2 except that lines 6 $\sim$ 7 are replaced by the line 7 in $\mathsf{Sim}_{\mathtt{H_2O_2RAM}}$. Construction 4 is the same as Construction 3 except that line 10 is replaced by the line 8 in $\mathsf{Sim}_{\mathtt{H_2O_2RAM}}$, yielding an adversarial advantage of $\sum_{i\in\{\ell,\ldots,L\}}\delta_{\mathsf{HT}_i(\mathsf{extract})}$. Construction 5 is the same as Construction 4 except that lines 11 $\sim$ 12 are replaced by lines 9 $\sim$ 13 in $\mathsf{Sim}_{\mathtt{H_2O_2RAM}}$, yielding an adversarial advantage of $\delta_{\mathsf{ocompact}}+\delta_{\mathsf{ointersperse}}$. Finally, Sim. B.1 is the same as Construction 5 except that the line 13 in Alg. 5 is replaced by its line 13, yielding an adversarial advantage of $\sum_{i\in\{\ell,\ldots,L\}}\delta_{\mathsf{HT}_i(\mathsf{build})}$. In short, the adversarial advantage of $\mathtt{H_2O_2RAM}$ is upper-bounded by $\sum_{i\in\{\ell,\ldots,L\}}\delta_{\mathsf{HT}_i}+\delta_{\mathsf{ocompact}}+\delta_{\mathsf{ointersperse}}$. As all the above values are negligible, the advantage of any PPT adversary against $\mathtt{H_2O_2RAM}$ is also negligible. It concludes the proof of Thm. 3.

**Oblivious Simulator B.1** $\mathsf{Sim}_{\mathtt{H_2O_2RAM}}(\mathsf{access},1^\lambda)$ :

Mark all levels but the bottom as empty.

**Output:** memory access pattern $\mathtt{addrs}$
1: initialize $\mathtt{addrs}\leftarrow\emptyset$
2: **for** $i\in\{\ell,\ldots,L\}$ **do**
3: $\quad$**if** level $i$ is empty **then** continue ▷ *public information*
4: $\quad\mathtt{addrs}\leftarrow\mathtt{addrs}\cup\{\mathsf{Sim}_{\mathsf{HT}_i}(\mathsf{lookup},1^\lambda)\}$
5: add the addresses of $\mathtt{op},\mathtt{res},v$, and the tail of $T_\ell$ to $\mathtt{addrs}$
6: **if** level $\ell$ is full **then** ▷ *public information*
7: $\quad$let $i^*$ be the first empty level or $i^*=L$ if all levels are non-empty ▷ *public information*
8: $\quad\mathtt{addrs}\leftarrow\mathtt{addrs}\cup\{\mathsf{Sim}_{\mathsf{HT}_\ell}(\mathsf{extract},1^\lambda)\,\|\,\ldots\,\|\,\mathsf{Sim}_{\mathsf{HT}_{i^*-1}}(\mathsf{extract},1^\lambda)\}$
9: $\quad$let $\tilde{n}$ be the length of $A$ in Alg. 5 line 11
10: $\quad$**if** $i^*=L$ **then**
11: $\quad\quad\mathtt{addrs}\leftarrow\mathtt{addrs}\cup\{\mathsf{Sim}_{\mathsf{ocompact}}(1^\lambda,\tilde{n})\}$
12: $\quad\mathtt{addrs}\leftarrow\mathtt{addrs}\cup\{\mathsf{Sim}_{\mathsf{ointersperse}}(1^\lambda,\tilde{n})\}$
13: $\quad\mathtt{addrs}\leftarrow\mathtt{addrs}\cup\{\mathsf{Sim}_{\mathsf{HT}_{i^*}}(\mathsf{build},1^\lambda,\tilde{n})\}$
14: $\quad$mark the levels from $\ell$ to $i^*$ as empty and $i^*$ as non-empty
15: **return** $\mathtt{addrs}$