Alibi: A Flaw in Cuckoo-Hashing based Hierarchical ORAM Schemes and a Solution

Brett Hemenway Falk[1], Daniel Noble[2], and Rafail Ostrovsky[3]

[1] University of Pennsylvania, fbrett@cis.upenn.edu
[2] University of Pennsylvania, dgnoble@cis.upenn.edu
[3] UCLA, rafail@cs.ucla.edu

Abstract.
There once was a table of hashes
That held extra items in stashes
It all seemed like bliss
But things went amiss
When the stashes were stored in the caches

The first Oblivious RAM protocols introduced the "hierarchical solution," (STOC '90) where the server stores a series of hash tables of geometrically increasing capacities. Each ORAM query would read a small number of locations from each level of the hierarchy, and each level of the hierarchy would be reshuffled and rebuilt at geometrically increasing intervals to ensure that no single query was ever repeated twice at the same level. This yielded an ORAM protocol with polylogarithmic overhead.

Future works extended and improved the hierarchical solution, replacing traditional hashing with cuckoo hashing (ICALP '11) and cuckoo hashing with a combined stash (Goodrich et al. SODA '12). In this work, we identify a subtle flaw in the protocol of Goodrich et al. (SODA '12) that uses cuckoo hashing with a stash in the hierarchical ORAM solution.

We give a concrete distinguishing attack against this type of hierarchical ORAM that uses cuckoo hashing with a *combined* stash. This security flaw has propagated to at least 5 subsequent hierarchical ORAM protocols, including the recent optimal ORAM scheme, OptORAMa (Eurocrypt '20).

In addition to our attack, we identify a simple fix that does not increase the asymptotic complexity.

We note, however, that our attack only affects more recent *hierarchical ORAMs*, but does not affect the early protocols that predate the use of cuckoo hashing, or other types of ORAM solutions (e.g. Path ORAM or Circuit ORAM).[4]

1 Introduction

In this work, we describe an attack on a wide variety of *hierarchical Oblivious RAM (ORAM)* protocols in the literature. Oblivious RAM is a cryptographic primitive designed to allow a client to securely execute RAM programs using an *untrusted* memory. ORAM provides a method for simulating a virtual memory array, such that for any two equal-length sequences of reads and writes into the virtual array, the sequences of accesses to the underlying *physical memory* are indistinguishable.

Typically, encryption protects the data *content*, however, even when the underlying data are encrypted simply observing the *data access pattern* can leak significant information.

ORAM is applicable in several different types of scenarios, including:

1. **Outsourced storage:** If a client makes use of an outsourced (cloud) storage provider, even if the content is encrypted, the storage provider can observe the client's access pattern. This may leak sensitive information. ORAM allows all sequences of accesses (of equal length) to be indistinguishable to the server. (Note that if the *amount* of data that the user accesses is sensitive, then ORAM cannot hide this.)
2. **Secure hardware:** If a small, trusted hardware component makes use of a (cheaper) untrusted memory, observing the memory access pattern can compromise the security of the processes running within the trusted component. This was the original proposed application [Ost90] and is a concern where memory side-channel attacks exist. A secure enclave, such as Intel SGX, is a recent real-world computing environment in which computation is performed on secure hardware, but the application needs the memory resources of an untrusted operating system. A series of works have shown that revealing memory access patterns is indeed a problem for SGX [BMD+17, GESM17, MIE17, JHOvD17], and this leakage can be mitigated using ORAM and other oblivious data structures to allow enclaves to use untrusted memory without leaking access patterns [SGF17].
3. **Secure multiparty computation (MPC):** ORAM is also useful in secure multiparty computation (MPC), where a group of parties engage in a distributed protocol to compute a joint function of their private data. Most MPC protocols use cryptographic secret sharing to protect the *content* of the data, and execute computations in the *circuit model* to ensure that the computation's control flow remains independent of the private data. Efficient ORAM protocols have the potential for allowing efficient, secure multiparty computation in the *RAM model* [OS97, LHS+14, Ds17, WHC+14].

The first ORAM construction [Ost90], introduced the *hierarchical solution*, and many subsequent works have expanded and built on this paradigm [Ost92, GO96, GMOT12, KLO12, LO13, PPRY18, AKL+20]. We review the hierarchical solution in Section 2.5.

The original Hierarchical ORAM builds a hierarchy of $\mathcal{O}(\log(n))$ levels, each containing a hash table with buckets of size $\mathcal{O}(\log(n))$, leading to lookups (ignoring the costs of rebuilds[5]) having a cost of $\mathcal{O}(\log^2(n))$.

To reduce the cost of each lookup, the traditional hash tables at each level can be replaced with cuckoo hashing, which reduces the cost of accessing each hash table to $\mathcal{O}(1)$ per virtual access. The initial solution [PR10] allowed cuckoo hashing to fail with some non-negligible probability, and in the case that it did, the hash table would be reconstructed. The failure (and rebuilding) of a cuckoo hash led to security problems, however, since the ORAM protocol would rebuild the hash table until there were no collisions, an adversary who observed a collision in the *physical access pattern*, would learn that the client had made queries for elements not stored in that level [GM11].

This problem was resolved by reducing the cuckoo hash failure probability by including a stash [PR04]. If each Cuckoo Hash Table in the hierarchy includes a $\mathcal{O}(\log(n))$-sized stash, the probability of a build failure becomes negligible, and no rehashing is needed [GM11].[6] At query time, every element of the stash at each level needed to be accessed, so although this eliminated the security problem created by cuckoo hashing failures, it did not improve the asymptotic overhead, which remained $\mathcal{O}\left(\log^2(n)\right)$.

Scanning separate cuckoo stashes at every level of the hierarchy significantly adds to the query complexity, and Goodrich et al. [GMOT12] then observed that even though the size of the stash for *each* level needs to be $\mathcal{O}(\log(n))$ in order to ensure a negligible probability of failure, the same failure probability could be maintained by combining the stashes at all levels into a single $\mathcal{O}\left(\log(n)\right)$-sized stash. Similarly, Kushilevitz et al. [KLO12] proposed that elements that would otherwise be placed in a cuckoo stash could instead be re-inserted directly into the ORAM data structure. Both these techniques improved the asymptotic complexity of accesses in the hierarchical solution to $\mathcal{O}(\log(n))$ physical accesses per virtual access.

In this work, we show that the techniques of combining cuckoo stashes across different levels of the hierarchy (introduced by Goodrich et al. and Kushilevitz et al.) creates a subtle security flaw which gives an adversary non-negligible advantage in distinguishing access patterns. The problem is similar to the problem in [PR10], where rehashing in the event of a build failure leaked information about the elements being stored at that level. Removing the elements from the stash on each level, like performing a rehashing, causes the elements that would have been in the stash to no longer be in that level. Therefore, if these elements are

[5] Rebuilds require constructing oblivious hash tables, which is relatively costly, so the amortized cost of lookups is usually dominated by the rebuild cost. Much of the progress in the literature has been towards reducing this cost, but to simplify the narrative, we focus here only on the costs of lookups without rebuilds.

[6] Even though a logarithmic-sized stash provides a negligible failure probability, for the smaller levels, a failure probability that is negligible in the size of the *level* may be non-negligible in the overall size of the ORAM. To avoid this problem, [GM11] suggested using traditional hash tables (rather than cuckoo hashing) for the smaller levels of the hierarchy, *i.e.,* until the level size reached $\mathcal{O}\left(\log^7(n)\right)$.

searched for they will be found before this level is reached, so instead of accessing the locations for the stashed elements at that level, random locations will be accessed instead. This means that, if all elements that were placed in a given level are searched for (including the items that were stashed), the access pattern of that level is less likely to contain any collisions in the *physical access pattern*. In contrast, if no elements from that level are accessed, all accessed locations will be random. The expected number of collisions will therefore be higher in the second case, and we will show that this difference is non-negligible.

This flaw affects a large number of papers [GMOT12, KLO12, LO13, PPRY18, KM19, AKL+20] which combine stashes in order to eliminate super-constant sized stashes at each level. This does not affect earlier hierarchical solutions that did not combine the stash e.g. [Ost90, Ost92, GO96] or non-hierarchical ORAMs such as PathORAM [SvDS+13] or Circuit ORAM [WCS15]. In addition to finding this flaw, we present a simple solution. Our solution applies to all schemes which suffer from the flaw without affecting their asymptotic complexity.

In Section 2.3, we review cuckoo hashing, and in Section 2.5 we review the basic hierarchical ORAM construction. In Section 3, we present our concrete attack that allows an adversary to distinguish two different access patterns with non-negligible probability in hierarchical ORAM solutions that use Cuckoo Hashing with a *combined stash*. This attack has a nice intuitive interpretation. However, this attack does not apply directly to PanORAMa and OptORAMa, so in Section 4 we present a generic version of our attack which does apply to these protocols. The generic attack is also shorter and simpler. In Section 5 we present our solution and prove that it is correct. Finally, we present the protocols that have been affected by this flaw in Section 6.

2 Preliminaries

2.1 Notation and Model

For any positive integer x, $[x] \stackrel{\text{def}}{=} \{1, \ldots, x\}$. For a set Z, $z \xleftarrow{\$} Z$ denotes that z is chosen uniformly at random from Z and 2^Z denotes the powerset of Z.

We denote a sequence using parenthesis as follows: $v = (v_1, \ldots, v_t)$. Sequences can alternatively be thought of as vectors or tuples, and we use the standard subscript v_i to denote the element at location i of sequence v. Set notation (e.g. $\in, \cup$) is often applied to sequences, in which case the sequence is implicitly first mapped to the set of elements it contains.

We think of an ORAM as an oblivious implementation of a RAM. Therefore, the index space, which we denote $\mathcal{V}$, is simply $[N]$, where N is the size of the ORAM.[7] We assume that the payloads are chosen from a space $\mathcal{W}$. For all $w \in \mathcal{W}$ we assume that $|w|$, the length of the bit-representation of w, is the same, so that items cannot be distinguished by volumetric attacks.

[7] With some additional work, an ORAM scheme can be made to be an oblivious implementation of a *dictionary*, *i.e.,* that have keys chosen from a space different than $[N]$, but we avoid this version for simplicity.

As is standard practice, we model the hash functions as truly random functions (see [Mit09] for a discussion of this assumption). Assuming that the hash functions are truly random implies that the adversary (who only learns outputs of the hash function) cannot gain any additional information about the hash function. Our protocols are secure against computationally *unbounded* adversaries in this model.[8] We consider protocols secure (information-theoretically secure in the random hash function model) if the distributions of adversary views do not change much based on sensitive data. Formally, let $D(x)$, $D(x')$ be two distributions of views of the adversary on differing sensitive data x and x'. Let $\Delta(D(x), D(x'))$ denote the statistical distance between two distributions. Protocols are secure if $\Delta(D(x), D(x'))$ is negligible (in N) for all pairs x, x'.

2.2 Oblivious Hash Tables

The hierarchical ORAM scheme builds on Oblivious Hash Tables which we formalize and abstract in Definition 1. We view a hash table as a method for storing (v, w) pairs, where $v \in \mathcal{V} = [N]$ is a (virtual) index, and $w \in \mathcal{W}$ is a payload. Let $\mathcal{X} = \mathcal{V} \times \mathcal{W}$.

Definition 1 (Oblivious Hash Tables). *An Oblivious Hash Table*

$$\mathsf{T} = (\mathsf{Gen}, \mathsf{Build}, \mathsf{Lookup}, \mathsf{Delete}, \mathsf{Extract})$$

is a tuple of polynomial-time algorithms

- ***Setup:*** $\mathsf{k} \leftarrow \mathsf{Gen}(N, m)$ *generates a key for a hash table of capacity m, storing (virtual) indices from $[N]$. In most cases, the key is simply the description of the hash functions.*
- ***Building:*** *The function $T \leftarrow \mathsf{Build}(\mathsf{k}, X)$ takes a set, $X \subset \mathcal{X}$, $|X| \leq m$ and builds a table, T, containing the elements in X. For any X, the probability that $\mathsf{Build}(\mathsf{k}, X)$ fails is negligible in N, i.e., is bounded by $N^{-\omega(1)}$.*
- ***Lookup:*** *The deterministic function $Q \leftarrow \mathsf{Lookup}(\mathsf{k}, v)$ takes a (virtual) index $v \in \mathcal{V}$, and returns a set of query locations $Q \subset [|T|]$.*
- ***Delete:*** *The deterministic function $\mathsf{Delete}(\mathsf{k}, v, T)$ removes items (v, w) if they exist in any location $T[i]$ where $i \in Q \leftarrow \mathsf{Lookup}(\mathsf{k}, v)$. Delete accesses exactly the indexes of T in Q and does not access any other memory.*
- ***Extract:*** *The function $\bar{X} \leftarrow \mathsf{Extract}(\mathsf{k}, T)$, takes a key k and a table T and returns a set of elements $\bar{X}$.*

These algorithms satisfy the following correctness properties. Suppose $\mathsf{k} \leftarrow \mathsf{Gen}(N, m)$ and $X \subset \mathcal{X}$ with $|X| \leq m$.

- ***Building:*** *If $T \leftarrow \mathsf{Build}(\mathsf{k}, X)$, then $T \in \mathcal{X}^{|T|}$. For every $(v, w) \in X$, we say that the payload w was stored in virtual location v and that (v, w) is stored in T.*

[8] In practice, implementations must use hash functions that are not truly random, but seem sufficiently random to a computationally bounded adversary.

– ***Lookup:*** *If* $T \leftarrow \mathsf{Build}(\mathsf{k}, X)$*, then for any* $(v, w) \in X$*, if* v *has not been deleted from* T*, the lookup* $Q \leftarrow \mathsf{Lookup}(\mathsf{k}, v)$ *produces a set of indices,* $Q \subset [|T|]$ *such that* $(v, w) \in T[i]$ *for some* $i \in Q$ *with probability at least* $1 - N^{-\omega(1)}$*.*
– ***Extraction:*** *If* k, T *are constructed as above, and* D *is the set of items deleted from table* T*, and* $\bar{X} \leftarrow \mathsf{Extract}(\mathsf{k}, T)$ *then* $x = (v, w) \in \bar{X}$ *iff* $(v, w) \in X$*,* $v \notin D$*,*

Additionally, these algorithms will need to allow the above functions to be executed obliviously. We define two notions of obliviousness: access-obliviousness and full obliviousness. Full obliviousness includes access-obliviousness. In our attack, we show that "combined-stash" cuckoo hashing schemes are not access-oblivious*, and hence cannot be fully oblivious. Since the techniques used to obliviously perform builds and extractions are complex and varied, focusing on access-obliviousness will simplify exposition.*

In brief, a protocol is access-oblivious if equal-length non-repeating sequences of indexes have indistinguishable outputs from Lookup*. This is the best that can be achieved. Since* Lookup *is deterministic, repeated indexes in the input to* Lookup *will result in repeated outputs, so if one sequence contains repeats and another doesn't the outputs of* Lookup *will be easily distinguishable. ORAM can be viewed as a way of modifying an Oblivious Hash Table to allow repeated queries of the same index.*

Definition 2. *A sequence* $v = (v_1, \ldots, v_t)$ *is said to be* non-repeating *if for all* $1 \leq i < j \leq t$*,* $v_i \neq v_j$*.*

– ***Obliviousness:***
  • ***Access-obliviousness:*** *For any two sets* $X, X' \subset \mathcal{X}$ *with* $|X|, |X'| \leq m$ *and any* non-repeating *sequences of virtual indices* $v, v' \in \mathcal{V}^t$ *the sequence of outputs of* $\mathsf{Lookup}(\mathsf{k}, \cdot)$ *on* v *and* v' *have negligible statistical distance (in* N*). In other words*

$$\Delta\left((Q_1, \ldots, Q_t), (Q'_1, \ldots, Q'_t)\right) < N^{\omega(1)}$$

where the sequence of queries $(Q_1, \ldots, Q_t)$ *and* $(Q'_1, \ldots, Q'_t)$ *are generated according to the following experiments:*

$$\left\{ (Q_1, \ldots, Q_t) \;\middle|\; \begin{array}{l} T \leftarrow \mathsf{Build}(\mathsf{k}, X) \\ Q_1 \leftarrow \mathsf{Lookup}(\mathsf{k}, v_1) \\ \quad \vdots \\ Q_t \leftarrow \mathsf{Lookup}(\mathsf{k}, v_t) \end{array} \right\}$$

and

$$\left\{ (Q'_1, \ldots, Q'_t) \;\middle|\; \begin{array}{l} T' \leftarrow \mathsf{Build}(\mathsf{k}', X') \\ Q'_1 \leftarrow \mathsf{Lookup}(\mathsf{k}', v'_1) \\ \quad \vdots \\ Q'_t \leftarrow \mathsf{Lookup}(\mathsf{k}', v'_t) \end{array} \right\}.$$

- ***Full obliviousness:*** *The complete sequence of accesses from building, lookups, deletions and extractions are oblivious, provided that the lookup and deletion sequences are the same, and the sequences are non-repeating. Note that deletions access the same locations as the results of lookups, so the access pattern of deletions do not provide additional information and can be ignored. Concretely, for any two sets $X, X' \subset \mathcal{X}$ with $|X|, |X'| \leq m$ and any non-repeating sequences $v, v' \in \mathcal{V}^t$ and*

$$A \stackrel{def}{=} \left\{ \mathsf{Acc} \begin{pmatrix} T \leftarrow \mathsf{Build}(\mathsf{k}, X) \\ \bar{X} \leftarrow \mathsf{Extract}(\mathsf{k}, T) \end{pmatrix} \;\middle|\; \mathsf{k} \leftarrow \mathsf{Gen}(N, m) \right\}$$

and

$$A' \stackrel{def}{=} \left\{ \mathsf{Acc} \begin{pmatrix} T' \leftarrow \mathsf{Build}(\mathsf{k}', X') \\ \bar{X}' \leftarrow \mathsf{Extract}(\mathsf{k}', T') \end{pmatrix} \;\middle|\; \mathsf{k}' \leftarrow \mathsf{Gen}(N, m) \right\}$$

where $\mathsf{Acc}(f(\cdot))$ are the set of physical memory accesses when executing function f and Q_i, Q_i' are defined as above using the same $v, v', X, X', \mathsf{k}, \mathsf{k}'$, then

$$\Delta((A, Q_1, \ldots, Q_t), (A', Q_1', \ldots, Q_t')) < N^{\omega(1)}$$

Remark 1 (Full Obliviousness). In a single-party ORAM setting, the hash-table must provide full obliviousness. It is possible in a multi-party setting to have the construction, accessing and extraction of the hash table be performed by different parties (*e.g.,* [LO13]). In this case, the set of functions executed by each individual party must be oblivious, but the combined set of all functions need not be.

Remark 2 (Insertions). Although most hash tables support *insertion*, the hierarchical ORAM construction does not require this feature – instead, elements are inserted into the ORAM only during rebuilds. Thus we do not include insertion as a necessary functionality in our formal definition of a hash table.

Remark 3 (Deletions and Extraction). Some ORAM schemes do not delete items as they are accessed, but rather extract data from all levels and then perform deduplication. However, the definition presented here simplifies proofs.

2.3 Cuckoo hashing

Cuckoo hashing was introduced in [PR04] as a method of multiple-choice hashing with expected constant-time lookups. Since its introduction, many variants of cuckoo hashing have been proposed and analyzed (see [Mit09] for a review). In this section, we review a basic common form of cuckoo hashing, but we emphasize that our attack works for almost all types of hashing with a stash.

We view a Cuckoo Hash Table as an array, T, with $cn + s$ locations, each having capacity one. Each element, x, can be placed in one of d locations given by $h_i(x)$ for $i = 1, \ldots, d$ where $h_i(x) \in [cn]$. If an element cannot be placed in

one of its d locations, it is placed in a logarithmic-sized "stash," *i.e.,* a location in $cn + 1, \ldots, cn + s$.

With appropriate choices of constants c and d, and a stash of size $s = \log(n)$, cuckoo hashing will succeed except with probability negligible in n (Theorem 2 of [ADW14]).

- **Key generation:** Generate $d \geq 2$ hash functions $h_i : \mathcal{V} \to [cn]$ for $i \in [d]$.
- **Building:** The build algorithm must place each element $(v, w) \in X$ in either $T[h_i(v)]$ for some $1 \leq i \leq d$ or in $T[cn + j]$ for some $1 \leq j \leq s$. If there is no allocation of elements that satisfies this condition, the build fails. Building can be accomplished by repeated insertions, or an "offline" algorithm. We do not specify how the build is accomplished obliviously as this varies significantly between protocols.
- **Lookups:** Return $Q = (h_1(v), \ldots, h_d(v), cn + 1, \ldots, cn + s)$. To read an element from a virtual index, v, read $T[h_i(v)]$ for $i = 1, \ldots, d$, and check if any of the elements retrieved are of the form (v, x) for some x.
- **Deletions:** Find $Q \leftarrow \mathsf{Lookup}(\mathsf{k}, v)$ and for any $i \in Q$ if $T[i] = (v, x)$ set $T[i] = (\perp, \perp)$.
- **Extractions:** Again, the method for performing extractions obliviously varies significantly between protocols, so we do not outline it here.

Fig. 1. Cuckoo Hash Table (1-table version) [PR04]

Lemma 1. *Cuckoo Hash Tables, as presented in Figure 1 are* access oblivious.

Proof. Since each hash function is truly random, the first time an item is queried to a hash function, the result is chosen uniformly at random and independent of all previous choices. Therefore, within the scope of the access obliviousness experiment, the values Q_i and Q'_i will all be chosen uniformly at random and independently, since each access sequence is distinct, and the keys are different in the two experiments. Therefore, $(Q_1, \ldots, Q_t)$ and $(Q'_1, \ldots Q'_t)$ will actually be chosen from the same random distribution, and the statistical distance between them is 0.

Remark 4 (Set Membership in Table). The *access pattern* of a Cuckoo Hash Table does not reveal whether the queried elements were present in the table or not. This follows because non-stash locations accessed are always chosen uniformly at random from $[cm]^d$ (and the stash locations are always accesssed).

Unlike some other constructions, Cuckoo Hash Tables hide set membership *without* the insertion of dummy elements, *i.e.,* pre-inserted elements that should be searched for in the case the item is not in the table.

If Cuckoo hashing is combined with an appropriate Build and Extract construction, it can be fully oblivious. Note that this not only requires that the Build

and Extract functions are oblivious in themselves, but that when Build, Lookup and Extract are all performed by a single entity, that the *combined* sequence of accesses is still oblivious.

Remark 5 (1-table vs d-table cuckoo hashing). We describe a single-table cuckoo hashing scheme, where all d hash functions hash into the same table. Alternatively, some cuckoo hashing constructions use d tables, and hash function i hashes into table i. Setting d to 2 is a common choice, resulting in 2-table cuckoo hashing. Using 1- vs d-table cuckoo hashing does not change the asymptotic performance of the hashing scheme, although it does change some details in the analysis.

A single-table Cuckoo Hash Table corresponds naturally to bipartite multigraph with n left-hand nodes (corresponding to $[n]$) and cn right-hand nodes corresponding to the hash buckets (*i.e.,* the first cn locations in the array T). Then a left hand node, v, is connected to d right hand neighbors given by $\{h_i(v)\}_{i=1}^{d}$. It is straightforward to see that the build procedure can succeed if there is a bipartite matching that includes $|X| - s$ left-hand vertices. The matched elements can be placed in their right-hand neighbors (given by the matching) and the remaining s elements can be placed in the stash.

This also shows that the build procedure can be implemented by building this bipartite multigraph and calculating a maximum matching. We assume that whatever build procedure is used does find such a maximum matching. In practice, analyses of build processes generally assume that a maximum matching is found, even if they use an algorithm which is not known to provide a maximum matching. For instance, in [KMW09] builds use a bounded-time insertion which is not guaranteed to find an optimal allocation, but is heuristically found to be nearly optimal.

To be an Oblivious Hash Table, the functions Build and Lookup need to fail with probability $N^{-\omega(1)}$. If a Cuckoo Hash Table is successfully built, the locations returned by Lookup will always include the location of the queried item if it is stored in the table, so the probability of failure is 0. Build, however, can fail. If the stash is chosen by finding a maximum matching, the probability of failure is $\mathcal{O}(n^{-s})$ for any constant s [KMW09]. A similar result holds for $s = \mathcal{O}(\log n)$, for which the probability of failure is $\mathcal{O}(n^{-\frac{s}{2}})$ [ADW14]. Therefore, if $s = \Theta(\log(n))$ the failure probability is $\mathcal{O}(n^{-\Theta(\log n)})$, which is negligible in n. Note that for ORAMs, the failure probability needs to be negligible not in the capacity of the Cuckoo Hash Table, n, but in the capacity of the ORAM, N. If N is polynomial in n this will hold. Goodrich and Mitzenmacher show that if the stash size is $\Theta(\log(N))$ and $n = \Omega(\log^7(N))$ the failure probability is still negligible in N and propose using another type of oblivious hash table for $n = o(\log^7(N))$ [GM11]. We similalry assume that for $n = o(\log^7(N))$ some alternative Oblivious Hash Table is used so that the failure probability of each hash table is indeed negligible in N, rather than n.

We have shown here that the Cuckoo Hash Table presented here, with appropriate Build and Extract functions, is an example of an oblivious hash table

(with failure negligible in n). We next show how oblivious hash tables can be used to construct a hierarchical ORAM. This is secure, but we will later show that if the stashes are combined this breaks obliviousness.

2.4 ORAM

An Oblivious RAM (ORAM) provides access to a virtual memory such that all equal-length sequences of virtual memory accesses have indistinguishable physical access sequences. We define an ORAM formally below.

Definition 3 (ORAM). *An ORAM* $\mathsf{O} = (\mathsf{Init}, \mathsf{Query})$ *is a tuple of polynomial-time algoirthms:*

- ***Init:*** $O \leftarrow \mathsf{Init}(A, N)$*, where* N *is an integer, and* A *is an array of length* N *of elements from some space* $\mathcal{W}$*. This initializes the value of index* $i \in \mathcal{V} = [N]$ *to* $A[i] \in \mathcal{W}$*.*
- ***Query:*** $w' \leftarrow \mathsf{Query}(O, v, w)$ *where* O *is an ORAM object,* $v \in \mathcal{V}$ *is an index and* $w \in \mathcal{W} \cup \{\perp\}$*. If* $w = \perp$ *this is a read query and it returns the value at index* v*. If* $w \neq \perp$ *this is a write query and it returns* $\perp$ *and sets the value at index* v *to* w*.*

  The ORAM must satisfy the following correctness guarantee.

- ***Consistency:*** *When a read is performed on index* v*, the result equals the value that was last written to index* v*, or if a write has never been performed on index* v*, it returns the initial value of index* v*,* $A[v]$*.*

  The ORAM must additionally satisfy the following security property.

- ***Obliviousness:***
  Regardless of the data, or the sequence of queries, the access pattern to the physical memory is indistinguishable. Formally, for any initial arrays A*,* A' *of length* N *and any sequence of queries* $(v_1, w_1), \ldots, (v_t, w_t)$*,* $(v'_1, w'_1), \ldots, (v'_t, w'_t)$*, where* $v_i, v'_i \in \mathcal{V}$*,* $w_i, w'_i \in \mathcal{W} \cup \{\perp\}$*, given*

$$C \stackrel{def}{=} \left\{ \mathsf{Acc} \begin{pmatrix} O \leftarrow \mathsf{Init}(A, N) \\ \mathsf{Query}(O, v_1, w_1) \\ \cdots \\ \mathsf{Query}(O, v_t, w_t) \end{pmatrix} \right\}$$

  and

$$C' \stackrel{def}{=} \left\{ \mathsf{Acc} \begin{pmatrix} O' \leftarrow \mathsf{Init}(A', N) \\ \mathsf{Query}(O', v'_1, w'_1) \\ \cdots \\ \mathsf{Query}(O', v'_t, w'_t) \end{pmatrix} \right\}$$

  then

$$\Delta(C, C') < N^{\omega(1)}$$

Note that the basic ORAM security definition only gives the adversary the ability to see the *access pattern*, but not the underlying data itself. To hide the data, each record can be encrypted under the client's key using a symmetric-key cryptosystem, or, in multi-server ORAMs, each record can be secret-shared among the servers (e.g. [KM19]).

2.5 Hierarchical ORAM

The hierarchical ORAM construction was originally put forward in [Ost90, Ost92, GO96] and has since been used as a basis for many future ORAM protocols including [GMOT12, LO13, PPRY18, AKL+20]. In this section, we lay out a generic version of hierarchical ORAM and show it to be secure. In Section 3, we show how modifications to this basic scheme caused a subtle security problem that caused future schemes (using this modification) to be insecure.

A hierarchical ORAM consists of $\ell + 1$ levels. Of these, there are ℓ levels each consisting of an Oblivious Hash Table of increasing capacities. Additionally, there is one level L_0, also called the cache, which is an oblivious object similar to an Oblivious Hash Table but that additionally supports insertions and repeated queries in the access sequence. The cache only ever contains at most $\mathfrak{c}$ elements (where typically $\mathfrak{c} = \Theta(\log(N))$). We choose ℓ such that $\mathfrak{c}2^\ell \geq N$. Since $\mathfrak{c}$ is small, the cache can be implemented easily by performing a linear scan of its contents on each access.

We present the Hierarchical ORAM formally in Figure 2. We will now show why such ORAMs are secure, provided that the hash tables are fully oblivious. First, we need the following lemma.

Lemma 2. *The ORAM of Figure 2 satisfies an invariant that all possible indexes $v \in \mathcal{V}$ are stored in exactly one level in the ORAM. This invariant holds after initialization, after each cache insertion and after each rebuild, though need not hold between these points.*

Proof. By induction. The ORAM is initialized to store all indexes $v \in \mathcal{V}$ in level L_ℓ. Each query is to some $v \in \mathcal{V}$. When a lookup to some index v is made, by induction this index will exist at some level. Since each level is searched, this index will be found and deleted from this level. It will then be placed in the cache. Therefore, once the item has been inserted into the cache, each index $v \in \mathcal{V}$ will be stored in exactly one location. If a rebuild occurs, certain levels will be emptied and merged into a larger level. However, this merge preserves the set of indexes in the ORAM, since all indexes from levels $i = 0, \ldots, i^*$ are extracted and placed in level i^*.

Lemma 3. *An index is queried at most once at each (non-cache) level between rebuilds of that level, or equivalently, an index is queried at most once to any Oblivious Hash Table.*

Proof. If an index, $v \in \mathcal{V}$ is queried at a level L_i, it will be found at some level, (since by Lemma 2 it must exist at *some* level). It will then be placed

– **Input:** A virtual memory size N. An array of initial values A.
– **Init:** Set $t = 0$
Set $X = (v, A[v])$ for all $1 \leq v \leq N$.
For $i = 0, \ldots, \ell - 1$, set $\mathsf{k}_i \leftarrow \mathsf{Gen}(N, \mathfrak{c}2^i)$, $T_i \leftarrow \mathsf{Build}(\mathsf{k}_i, \emptyset)$.
Set $\mathsf{k}_\ell \leftarrow \mathsf{Gen}(N, \mathfrak{c}2^\ell)$, $T_\ell \leftarrow \mathsf{Build}(\mathsf{k}_\ell, X)$.

Hierarchical ORAM Initialization

– **Input:** A virtual memory address, v. A payload, x. (For read queries $x = \perp$.)
– **State:** A counter, t. Hash tables $\{T_i\}_{i\in[\ell]}$. Hash keys $\{\mathsf{k}_i\}_{i\in[\ell]}$. Local memory, m.
– **Scan the cache:** Initialize $\mathsf{found} = \mathsf{false}$. Read every element in the cache L_0. If a pair (v, w) is found, set $m = w$, $\mathsf{found} = \mathsf{true}$, and delete the old item from the cache.
– **Search each level:** For i in $1, \ldots, \ell$
  • If $\mathsf{found} = \mathsf{false}$ set $Q_i \leftarrow \mathsf{Lookup}(\mathsf{k}_i, v)$, otherwise set $Q_i \leftarrow \mathsf{Lookup}(\mathsf{k}_i, \mathsf{dummy}\circ t)$ where $\circ$ denotes concatenation, ensuring $\mathsf{dummy}\circ t \notin \mathcal{V}$.
  • Access $T_i[j]$ for all $j \in Q_i$. If there is a $j \in Q_i$, and a w such that $T_i[j] = (v, w)$, then set $m = w$ and $\mathsf{found} = \mathsf{true}$.
  • $\mathsf{Delete}(\mathsf{k}_i, v, T_i)$
– **Insert into the cache:** If $x \neq \perp$ (*i.e.,* it was a write query), insert (v, x) into the cache, otherwise insert (v, m) into the cache.
– **Rebuilding:** Increment t. Let $\tau = 2\mathfrak{c}$ be the rebuild period. If t is a multiple of τ initiate a rebuild (as described below).
– **Output:** Output m. If it was a read query, m will contain the read value.

Hierarchical ORAM queries.

– **State:** A counter, t. Hash tables $\{T_i\}_{i\in[\ell]}$. Hash keys $\{\mathsf{k}_i\}_{i\in[\ell]}$.
– **Identify level:** Let $\bar{i}$ be the largest value such that $\frac{t}{\tau} = 0 \bmod 2^{\bar{i}}$. Let $i^* = \min(\bar{i} + 1, \ell)$. We will merge levels $0, \ldots, i^*$ into level i^*.
– **Merge levels:** Initialize $X = \emptyset$. For $i = 0, \ldots, i^*$, and obliviously evaluate $X = X \cup \mathsf{Extract}(\mathsf{k}_i, T_i)$. Set $\mathsf{k}_{i^*} \leftarrow \mathsf{Gen}(N, \mathfrak{c}2^{i^*})$, and $T_{i^*} = \mathsf{Build}(\mathsf{k}_{i^*}, X)$.
– **Clear lower levels:** For $i = 0, \ldots, i^* - 1$, set $\mathsf{k}_i \leftarrow \mathsf{Gen}(N, \mathfrak{c}2^i)$, $T_i \leftarrow \mathsf{Build}(\mathsf{k}_i, \emptyset)$.

Hierarchical ORAM rebuilds.

Fig. 2. Hierarchical ORAM

in the cache. Until L_i is rebuilt, it will not exist in L_i, since the tables only support deletions, not insertions. Since the sizes of the tables are exponentially increasing, if L_j is rebuilt for some $j > i$, L_i will also be rebuilt (possibly to an empty table) so conversely, if L_i has *not* been rebuilt, L_j will also have not been rebuild for all $j > i$. Therefore, the index will not be stored at L_j for any $j > i$. Therefore, since the index must be stored somewhere, it is stored at some level L_k, where $k < i$. Since the Hierarchical ORAM searches levels sequentially, it will find the item before L_i is reached, will set $found = true$ and will therefore search for $dummy \circ t$. Therefore each $v \in \mathcal{V}$ will only be searched for once in L_i between rebuilds of L_i. The values of t increment with each ORAM query, so each query of form $dummy \circ t$ will also be queried at most once at any level.

We now show that the oblivious property of the ORAM follows easily from this lemma and the properties of Oblivious Hash Tables:

Theorem 1. *The Hierarchical ORAM protocol in Figure 2, when using an Oblivious Hash Table at each level, is oblivious as per Definition 3.*

Proof. The security of the ORAM protocol rests on two key facts: (1) *No repeated accesses:* An index is queried once in each level between rebuilds, or equivalently, the sequence of queries to each hash table is non-repeating. This was demonstrated in Lemma 3. (2) *Oblivious accesses:* Our definition of an Oblivious Hash Table (Definition 1) produces indistinguishable physical access patterns provided that the two sequences of virtual indices are *non-repeating*. This is satisfied as per fact (1), so the combined access patterns of builds, lookups, deletions and extractions at each level have distributions separated by negligible statistical distances. Accesses to the cache are always the same, so these do not increase the statistical distance between access distributions. Furthermore the access patterns of builds, lookups, deletions and extractions of each Oblivious Hash Table are independent of each other Oblivious Hash Table, since different keys are used each time. Therefore the combined access pattern of the entire data structure also has distributions separated by negligible statistical distances so is secure by Definition 3.

Remark 6 (Efficiency). While rebuilding the hash tables is expensive,[9] these rebuilds occur at a frequency proportional to the capacity of the table, thus the *amortized* cost can remain low. The exact communication cost depends on how the hash tables are implemented, and how the oblivious functions Build and Extract are implemented. We do not focus on these details here, as they do not bear directly on our attack.

3 The Attack

In this section, we describe a novel attack on hierarchical ORAM protocols that use cuckoo hashing with a combined stash. This attack applies directly to

[9] In the client-server setting expense is measured by communication between the client and the server. In the MPC setting, expense is measured as the communication between the parties in the computation.

[GMOT12, KLO12, LO13] and Instantiation 2 of [KM19]. The recent works of PanORAMa [PPRY18] and OptORAMa [AKL+20] use a modified hierarchical solution with multiple cuckoo tables at each level. Since the attack presented here assumes that the adversary can know which indexes are stored in the Cuckoo Hash Table, it does not apply directly to PanORAMa and OptORAMa. In Section 4 we present a more general attack that also applies to PanORAMa and OptORAMa. The general attack is also simpler, but this attack has the advantage of having an intuitive interpretation.

3.1 Simplified attack

First, we describe this attack in a simplified setting, which we later show is equivalent to the ORAM setting.

Imagine the following construction of a hash table. A Cuckoo Hash Table, as defined in Figure 1, is modified in the following way. When querying some item $v \in \mathcal{V}$, the stash will be searched first. If the item is found in the stash, then some new unique index $v' \notin \mathcal{V}$ will be searched for in the remainder of the table, *i.e.,* $h_i(v')$ will be accessed for $1 \leq i \leq d$. This construction is presented in Figure 3. We will show that this object is no longer an Oblivious Hash Table.

Build, Delete and Extract are the same as in Cuckoo Hash Tables (Figure 1)

- **Lookups:** Lookup takes the key k, an index v and the table object T, and returns a set of indexes, Q. If v is not in the stash, (*i.e.,* $T[j] \neq (v, w)$ for any $cm + 1 \leq j \leq cm + s$) return $Q = (cm + 1, \ldots, cm + s, h_1(v), \ldots, h_d(v))$. However, if v *is* in the stash pick a new $v' \notin \mathcal{V}$, using an internal counter to ensure that the same v' is never selected twice, and return $Q = (cm + 1, \ldots, cm + s, h_1(v'), \ldots, h_d(v'))$.

Fig. 3. Stash-Resampling Cuckoo Hash Table

Observe that previously, Lookup only took k and v as parameters, whereas in this definition, its behavior depends on an additional parameter T. Specifically, Lookup now depends on which items were placed in T's stash. The fact that the access pattern changes depending on how the table is constructed breaks the abstraction of an Oblivious Hash Table. We will next show that this break leads to a concrete vulnerability.

Remark 7. We describe our attack in terms of cuckoo hashing, but essentially the same argument goes through with other hashing schemes that use a stash.

Let T be a Stash-Resampling Cuckoo Hash Table containing indices $v = (v_1, \ldots, v_t)$ and using hash functions $h = (h_1, \ldots, h_d)$. Imagine computing $\mathsf{Lookup}(k, v_i, T)$

for $1 \leq i \leq t$. Let v' be the sequence of inputs to the hash functions. If v_i was not stashed, $v'_i = v_i$, but if v_i was stashed, v'_i will be some other unique value.

Now imagine that a Cuckoo Hash Table is constructed using hash functions h, but with indices v'. All items that were already stored in the table can continue to be stored in the table. However, it is likely that if v_i was stashed, v'_i will not need to be stashed, since it is hashed to new locations, one of which is probably empty. Therefore the stash size of this Cuckoo Hash Table is smaller than usual. Now, an adversary does not know h or v', but it *does* learn $h_j(v'_i)$ since these are returned by Lookups. Therefore, it can learn what the stash size *would have been* in a table that used hash functions h and indexes v'.

In contrast, let v'' be a sequence of t accesses, none of which are in T. Since none are in the stash, v'' are also the inputs to the hash functions and the adversary can learn from the access pattern the size the stash would have been if the table stored v''. The values of $h_j(v''_i)$ will be chosen uniformly at random, so this stash would be chosen from the usual stash size distribution. Hence, if the adversary calculates what the stash size *would have been* if a table was constructed from the hash function inputs, the distribution of stash sizes will be *smaller* if v is queried than if v'' is queried.

We now prove formally that a Stash-Resampling Cuckoo Hash Table is not access-oblivious. We formalize the intuition above by representing the accesses as a bipartite graph, with m left-vertices corresponding to the m inputs to the hash functions, with cm right-vertices corresponding to the non-stash locations in the table and edges from a left-vertex to a right-vertex if one of the hash functions maps the left-vertex to the right-vertex. A maximum matching in the graph therefore corresponds to a possible assignment of elements to locations in the hypothetical hash table constructed by the adversary. The number of unmatched elements then will correspond to the stash size. Below, we formalize the correspondance from access sequences to graphs and show that the distribution of the number of unmatched elements in the graphs indeed differs non-negligibly.

Definition 4 (Graph Representation of an Access-Pattern). *The Graph Representation of an Access Pattern, $B(m, c, Q)$ is a function that takes as inputs integers m and c and a sequence of access sets, $Q = Q_1, \ldots, Q_m$, and returns a bipartite multigraph with left vertices $a_1, \ldots, a_m$, right vertices $b_1, \ldots, b_{cm}$ and edges (a_i, b_j) for $j \in Q_i \cap [cm]$.*

Definition 5 (Left-regular bipartite multigraph). *We define a left-regular bipartite multigraph to be a graph $G = (L \cup R, E)$ with the following properties.*

- *It is bipartite, with vertex sets L and R, and each edge being directed from L to R, i.e., $\forall (u, v) \in E, u \in L, v \in R$.*
- *Every vertex in L has a constant number of edges, denoted d.*
- *E is a multiset, i.e., the edge (u, v) may occur multiple times.*

Definition 6 (Random left-regular bipartite multigraph). *We define $H_0(m, c, d)$ to be a function that produces a random left-regular bipartite multigraph, where $|L| = m$, $|R| = c \cdot m$, $d \geq 1$ is the degree of each vertex in L and where each*

outgoing edge from a vertex $u \in L$ has an end-point, $v \in R$, that is chosen uniformly at random from R (and independent of all other choices).

If $Q = (Q_1, \ldots, Q_m)$ is the result of outputs of Lookup to a sequence of queries to a (Stash-Resampling) Cuckoo Hash Table with capacity m and degree d, then $G \leftarrow B(m, c, Q)$ will be a left-regular bipartite multigraph, since every Q_i will contain d vertices in $[cm]$. We will soon show that for a Stash-Resampling Cuckoo Hash Table, if none of the queried elements are in the table, G will be sampled as a *random* left-regular bipartite multigraph, but if the table contents are queried, the left-regular bipartite multigraph will be sampled from a *different* distribution of graphs which will have fewer unmatched elements.

Definition 7 (Matching of a bipartite multigraph). *For a bipartite multigraph $G = (L \cup R, E)$, a matching is a set of edges $E' \subseteq E$ such that*

$$(u, v), (u', v') \in E' \Rightarrow u \neq u', v \neq v'.$$

A maximum matching is a matching of maximum size. There may be multiple such matchings, but they will all be the same size; we use $M(G)$ to denote some such matching and $|M(G)|$ to be this size, which is independent of which matching is chosen. $S(G) \stackrel{def}{=} m - |M(G)|$ is the number of unmatched elements on the left-hand side.

Note that for any G, $1 \leq |M(G)| \leq m$, so $0 \leq S(G) \leq m - 1$.

Lemma 4 (Lower bound on unmatched elements). *For all $0 \leq s \leq m-1$ and $G \leftarrow H_0(m, c, d)$, where d, c are constants,*

$$\Pr[S(G) \geq s] \geq \left(\frac{1}{cm}\right)^{ds+d-1}$$

which is non-negligible in m.

Proof. Pick $s+1$ elements of L. The probability that *all* $d \cdot (s+1)$ edges of these elements will have the same endpoint $v \in R$ is $\left(\frac{1}{cm}\right)^{d(s+1)-1} = \left(\frac{1}{cm}\right)^{ds+d-1}$. If this occurs, any matching can contain at most 1 of these elements, which means that at least s of these elements will be unmatched. Thus $S(G) \geq s$. Note that for any constant d and s, this probability is non-negligible.

Next, we describe two distributions on the integers $[m-1]$.

Definition 8. *Fix constants $d, m \in \mathbb{N}$, and $c > 1$. Let $M(\cdot)$ be an algorithm that takes a bipartite multigraph G, and returns a maximum matching $M(G)$.*

- ***Distribution 0:** Let s_0 be the random variable denoting the number of unmatched elements in a random bipartite multigraph. $s_0 \stackrel{def}{=} S(H_0(m, c, d))$.*

– ***Distribution 1:*** *Define a distribution of graphs according to the following process. First construct a graph* $G' \leftarrow H_0(m, c, d)$*. Let* $G' = (L \cup R, E')$*. Let* $M(G')$ *be a maximum matching in* G'*. Initialize* $E = E'$*. For every* $u \in L$ *s.t.* $\nexists (u, v) \in M(G')$*, remove every edge* $(u, v) \in E'$*, and replace it with a new edge* (u, v') *where* v' *is chosen uniformly at random from* R*. Let* $G = (L_1 \cup R_1, E)$ *be the modified graph. Let* $H_1(m, c, d, M(\cdot))$ *denote the function that samples a graph from this distribution. Define* s_1 *to be the number of unmatched elements in this experiment, i.e.,* $s_1 \stackrel{def}{=} S(H_1(m, c, d, M(\cdot)))$*.*

Although the distributions s_0 *and* s_1 *depend on parameters, we generally suppress these dependencies for notational convenience.*

Intuitively, the expected value of s_1 should be smaller than the expected value of s_0, since the vertices which were not matched get another chance to be matched when new end-points are chosen for them. In Lemma 5 we show that this is indeed the case, and that the distributions of s_0 and s_1 are statistically different (*i.e.,* non-negligibly different).

Lemma 5. *If* s_0 *and* s_1 *are the random variables described above, then the statistical distance between* s_0 *and* s_1 *is at least* $\frac{1}{m}\left(1 - \left(\frac{1}{c}\right)^d\right)\left(\frac{1}{cm}\right)^{2d-1}$ *which is non-negligible in* m*.*

Proof. Consider the graph $G' = (L \cup R, E') \leftarrow H_0(m, c, d)$ generated as the first step in generating distribution s_1, where $|R| = c \cdot m$. Let $M = M(G')$. Let $S \subset L$ be the unmatched vertices in L. We know $|S|$ is distributed by s_0. When G is constructed (as the second step of distribution s_1), each $u \in S$ will receive d new random neigbors. For $v \in L/S$ we can use the existing matching M for G and for $u \in S$ we can match it to a neighbor directly if this neighbor is not already matched.[10] Since at most m elements of R will ever be matched, the probability that a new random neighbor is already matched is at most $\frac{1}{c}$. There is then at most a $\left(\frac{1}{c}\right)^d$ probability that *all* d right-hand neighbors of u are already matched. Let e'_i be the event that v_i is unmatched in G', and e_i the event that v_i is unmatched in G. This shows:

$$Pr[e_i] \leq \left(\frac{1}{c}\right)^d Pr[e'_i]$$

Thus by linearity of expectation

$$E[s_1] = \sum_{1 \leq i \leq m} Pr[e_i] \leq \sum_{1 \leq i \leq m} \left(\frac{1}{c}\right)^d Pr[e'_i] = \left(\frac{1}{c}\right)^d E[s_0].$$

[10] This greedy matching assignment not give an optimal matching for G, but it will provide an upper bound for s_1 in terms of s_0.

By Lemma 4, $Pr(s_0 \geq s) \geq \left(\frac{1}{cm}\right)^{ds+d-1}$. Since s_0 is a non-negative distribution, $E[s_0] \geq Pr(s_0 \geq 1) \geq \left(\frac{1}{cm}\right)^{2d-1}$ so

$$|E[s_0] - E[s_1]| \geq \left(1 - \left(\frac{1}{c}\right)^d\right)\left(\frac{1}{cm}\right)^{2d-1}.$$

In particular, this means that the expected values, $E[s_0]$ and $E[s_1]$ are non-negligibly different. Finally, notice that $0 \leq s_0, s_1 \leq m$, so

$$\Delta(s_0, s_1) \geq \frac{1}{m}|E[s_0] - E[s_1]| \geq \frac{1}{m}\left(1 - \left(\frac{1}{c}\right)^d\right)\left(\frac{1}{cm}\right)^{2d-1}$$

which means that $\Delta(s_0, s_1)$ is also non-negligible.

Now we show that the Stash-Resampling Cuckoo Hash Table is not access oblivious.

Theorem 2. *The Stash-Resampling Cuckoo table presented in Figure 3 is not access-oblivious.*

Proof. Let $X = X' = \{1, \ldots, m\}$ for some $m \leq \frac{N}{2}$. Let $v_i = i + m$ and let $v'_i = i$ for $1 \leq i \leq m$. The adversary will generate a table with the input data, lookup the sequence of virtual indices and construct a bipartite graph based on these lookup results.

Let there be two experiments:

$$\left\{\begin{array}{l} k \leftarrow \mathsf{Gen}(N, m) \\ T \leftarrow \mathsf{Build}(k, X) \\ \{Q_i \leftarrow \mathsf{Lookup}(k, v_i, T)\}_{i\in[m]} \\ G \leftarrow B(m, c, Q) \\ s = S(G) \end{array}\right\} \text{and} \left\{\begin{array}{l} k' \leftarrow \mathsf{Gen}(N, m) \\ T' \leftarrow \mathsf{Build}(k, X') \\ \{Q'_i \leftarrow \mathsf{Lookup}(k', v'_i, T')\}_{i\in[m]} \\ G' \leftarrow B(m, c, Q') \\ s' = S(G') \end{array}\right\}$$

In the first experiment, none of the queries are in X, therefore none will be in the stash. Therefore $Q_i = (cm + 1, \ldots, cm + s, h_1(v_i), \ldots, h_d(v_i))$. Since the v_i are distinct from each other and the elements stored in the table, $h_j(v_i)$ will be chosen uniformly at random from $[cm]$ and independently of all previous variables. Therefore, each left-vertex in G will have d neighbors, chosen uniformly at random from b_j. Therefore G is chosen exactly according to H_0.

In the second experiment, all of the queries are in X'. If we were to search according to the oblivious Cuckoo Hash Table of Figure 1 then the corresponding graph would be distributed according to $H_0(m, c, d)$. However, for any element that was not in the maximum matching, (*i.e.,* the elements in the stash) the Stash-Resampling Cuckoo Hash Table will instead pick new indices to query, $\bar{v}'_j$ and return locations $h_i(\bar{v}'_j)$ which will not have been queried before so will be new random locations. Therefore, for these elements that were not in the maximum matching, the corresponding edges will be re-chosen uniformly at random. The

graph from the second experiment will therefore be constructed according to distribution $H_1(m, c, d, M(\cdot))$, assuming the stash was chosen by some maximum matching algorithm $M(\cdot)$.

We have already shown that distributions $H_0(m, c, d)$ and $H_1(m, c, d, M(\cdot))$ are distinguishable. Therefore an adversary can distinguish the two experiments, so Stash-Resampling Cuckoo Hash Tables are not access-oblivious.

Remark 8. Note that the attack described above is immediately applicable in cases where the stash is accessed *before* the associated Cuckoo Hash Table, and if the target is found in the stash, the protocol searches for dummy elements in the table. For instance, our attack would apply to a hierarchical ORAM that stored a stash at the same level, but accessed the stash *first*, and searches for a dummy in the rest of the table if the element is found in the stash.

3.2 Hierarchical ORAM with a combined stash

We now present how hierarchical ORAMs were constructed using a combined stash. We will show that this breaks the abstraction of an Oblivious Hash Table, and results in access patterns identical to those of the Stash-Resampling Cuckoo Hash Table, which breaks obliviousness.

Beginning with the protocol of Goodrich et al. [GMOT12], a number of hierarchical ORAM schemes stored stashed items from a table construction in a shared stash or re-inserted them into the cache. Since most schemes re-insert stash items into the cache, we will present this version. Figure 4 presents the changes between the stash-reinserting hierarchical ORAM and the original hierarchical ORAM protocol from Section 2.5. All other parts of the protocol remain the same.

A Stash-Reinserting ORAM is an ORAM equivalent to that of Figure 2 with the following modifications:

- **Rebuild:** Rather than table T_{i*} storing all elements in X, at most $\mathfrak{c}$ of these elements can be stored in a stash. The stash is not stored at this level, but is is padded to size $\mathfrak{c}$ and inserted into the cache.
- **Rebuild frequency:** Since the cache is of size $\mathfrak{c}$ after a rebuild, the rebuild period is now $\tau = \mathfrak{c}$.

Fig. 4. Stash-Reinserting Hierachical ORAM

Theorem 3. *The Stash-Reinserting ORAM of Figure 4 is insecure; i.e., it does not satisfy the oblivious property in Definition 3.*

Proof. Let $A = A' = 0^N$. Let the hierarchical ORAM be such that there will be some level L_i of capacity $m \leq \frac{N}{2}$ that is implemented using a Cuckoo Hash Table.[11]

Let $U = ((1,0), \ldots, (2m,0))$ and $U' = ((1,0), \ldots, (m,0), (1,0), \ldots, (m,0))$ be two sequences of ORAM queries.

After m queries, L_i will be constructed.[12] In both experiments L_i will be constructed using the elements $(1,0), \ldots, (m,0)$. A Cuckoo Hash Table will be constructed in both cases, with these contents.[13]

The stash will be re-inserted in both cases. We have from Lemma 2 that each of these stashed elements will exist at a single location at the start of each access. Since levels L_j for all $j \geq i$ will only be rebuilt when L_i is also rebuilt, we know that these elements must remain in some level L_k with $k < i$ until L_i is rebuilt. This means that, until this point in time, they will always be found *before* L_i is accessed. Thus, by the ORAM query algorithm, a dummy query will be performed in L_i.

Therefore, the access pattern in the Cuckoo Hash Table at L_i will be the same as that of the Stash-Resampling Cuckoo Hash Table in Figure 3, where elements were searched in the stash first, and if found in the stash a dummy was searched in the remainder of the table. The only difference is that in the Stash-Resampling Cuckoo Hash Table, the algorithm also accessed a pre-assigned stash, but this is not an issue since the attack to the stash-resampling algorithm does not use the access pattern to the stash (as this access pattern is always the same). Observe that, exactly like in the attack of Theorem 2, one sequence of accesses (U) will only access elements that were *not* in the data table, and the other sequence (U') will only access elements that *were* in the data table (including the stash). Therefore, by the same argument as Theorem 2 the statistical distance between ORAM access pattern distributions is non-negligible. Therefore, the ORAM protocol is insecure.

[11] Some schemes use a mixture of hash table types at different levels. We do not require that all levels use a Cuckoo Hash Table, only that there is at least one such level of size $\leq \frac{N}{2}$ that has its stash re-inserted into the ORAM data structure.

[12] This is not quite true. We would like to construct L_i such that it contains indices $1, \ldots, m$ (although some may of these may be stashed). However, due to reinsersions of the stash this will actually need to occur in a level with capacity roughly $2m$. If additional accesses are needed to trigger the rebuild, then the same element, *e.g.,* $(1,0)$ can be looked up multiple times. The exact details of what sequence of accesses is needed in order to cause elements $1, \ldots, m$ to be inserted into a particular level also varies depending on how exactly the ORAM is constructed. More generally, the sequence $(1,0), \ldots, (m,0)$ at the beginning of both U and U' should be replaced with whatever sequence in the given ORAM is needed in order to instantiate a level to contain exactly the indices $1, \ldots, m$.

[13] It is possible that when the ORAM is initialized, elements from L_ℓ are stashed and stored in the cache. These elements would inadvertently also be stored in L_i. The effect of this on the Cuckoo Hash Table is small.

4 The Generic Attack

The attack in Section 3 assumes that an adversary knows all m elements that were placed in the Cuckoo Hash Table. However, in PanORAMa [PPRY18] and OptORAMa [AKL+20] each level contains multiple Cuckoo Hash Tables and only some of the elements are placed in any given table. We therefore now construct a more general attack that assumes only that the adversary knows a *superset* of the elements that were placed in the table. More formally, we can weaken the definition of Access-Obliviousness in Definition 1 such that the contents of the data in the two experiments are the same, and the access patterns cannot depend directly on the table contents, but are functions of any superset of the contents.

Definition 9. *A hash-table is* access-oblivious in the knowledge of a content superset *if for all datasets $X \subset \mathcal{X}$ with indices $V \subset \mathcal{V}$, and all PPT algorithms $f, f' : 2^{\mathcal{V}} \to \mathcal{V}^t$, there exists Y, with $V \subset Y \subset \mathcal{V}$, $|Y| \leq |\mathcal{V}| - 3$ such that the distribution of outputs of $\mathsf{Lookup}(k, f(Y), T)$ has negligible (in $|Y|$) statistical distance from the distribution of outputs of $\mathsf{Lookup}(k', f'(Y), T')$ where $T \leftarrow \mathsf{Build}(k, X)$, $T' \leftarrow \mathsf{Build}(k', T')$.*

We show that a Stash-Resampling Cuckoo Hash Table (Figure 3) does not satisfy this weaker security guarantee. We will then show that an adversary can then use this to differentiate sequences in PanORAMa and OptORAMa with non-negligible probability. For simplicity, our proof assumes $d = 2$ hash functions, which is the choice used by PanORAMa and OptORAMa, but can easily be extended to any constant number of hash functions.

4.1 Generic Stash-Resampling Cuckoo Hash Table Attack

Theorem 4. *Stash-Resampling Cuckoo Hash Tables with $d = 2$ are not access-oblivious in the knowledge of a content superset.*

Proof. Let $|X| = |V| = n' \geq 3$ and $|Y| = m$. The adversary algorithms are as follows: f chooses distinct $v_1, v_2, v_3 \xleftarrow{\$} Y$ and outputs $A = (v_1, v_2, v_3)$. f' chooses distinct $v'_1, v'_2, v'_3 \xleftarrow{\$} \mathcal{V}/Y$ and outputs $A' = (v'_1, v'_2, v'_3)$.

Let the set $S \subset V$ denote the indexes stored in the stash constructed by the table T, that is built in the first experiment, and let $|S| = s$. Define $B = V/S$ to be the indexes that were successfully stored in their hashed locations in T and define $C = S \cup Y/V = Y/B$ to denote indexes in Y that are not (either because they weren't in V to begin with, or because they were stashed). As previously assumed, there are 2 hash functions. By the definition of a Stash Resampling Cuckoo Hash Table, if $v_1, v_2, v_3 \in B$ are distinct elements, it is impossible for $(h_1(v_i), h_2(v_i))$ to be equal for all $i \in \{1, 2, 3\}$. Let r be the size of the set of outputs of $(h_1(v), h_2(v))$. For the Cuckoo Hash Table of Figure 1 $r = (cn')^2$. Let $Q_i = \mathsf{Lookup}(k, T, v_i)$ and $Q'_i = \mathsf{Lookup}(k', T', v'_i)$ be the results of Lookups in the first and second experiments respectively, but ignoring the

stash locations (since these are chosen deterministically). We now show that $\Delta((Q_1, Q_2, Q_3), (Q'_1, Q'_2, Q'_3)) \geq m^{-\Theta(1)}$, *i.e.,* that the accesses to A and A' are statistically different.

Let us first look at the distribution of (Q_1, Q_2) and (Q'_1, Q'_2). Since $v'_1, v'_2 \notin Y \supset V$, Q'_1 and Q'_2 contain random locations in the table. Therefore, the probability that $Q'_1 = Q'_2$ is exactly $\frac{1}{r}$.

Now let us look at the distribution of (Q_1, Q_2). If both $v_1, v_2 \in C$, Q_1 and Q_2 will both be chosen uniformly at random, and the probability that $Q_1 = Q_2$ would be $\frac{1}{r}$. Even if only one of v_1 or v_2 is in C, the locations returned by Lookup for this element will be chosen uniformly at random and independent from all previous choices, so the probability that $Q_1 = Q_2$ would be $\frac{1}{r}$ in this case also.

Now let us examine the probability that $Q_1 = Q_2$ for a randomly selected $v_1, v_2 \in B$. Let this probability be denoted by p for a given Cuckoo Hash Table implementation. For a random $v_1, v_2 \in V$, the probability that $(h_1(v_1), h_2(v_1)) = (h_1(v_2), h_2(v_2))$ is $\frac{1}{r}$. However, the build algorithm has some choice in which items it places in the stash. It is possible that elements that cause collisions are either more, or less, likely to be placed in S. Therefore p could be different from $\frac{1}{r}$, but we show it cannot be much different without making the output distributions non-negligibly statistically distant.

$$Pr(Q_1 = Q_2) = \frac{\binom{|B|}{2}}{\binom{|Y|}{2}} p + \left(1 - \frac{\binom{|B|}{2}}{\binom{|Y|}{2}} \frac{1}{r}\right)$$

$$Pr(Q_1 = Q_2) - Pr(Q'_1 = Q'_2) = \frac{\binom{|B|}{2}}{\binom{|Y|}{2}} \left(p - \frac{1}{r}\right)$$

There are two cases. In the first case $|p - \frac{1}{r}| \geq m^{\Theta(1)}$ then $|Pr(Q_1 = Q_2) - Pr(Q'_1 = Q'_2)|$ is non-negligible in m, (Q_1, Q_2) and (Q'_1, Q'_2) will be statistically different, and the proof is done. In the second case $|p - \frac{1}{r}| = m^{\omega(1)}$ and we will proceed to show that then (Q_1, Q_2, Q_3) and (Q'_1, Q'_2, Q'_3) are statistically different.

Let us examine the probability that $Q_1 = Q_2 = Q_3$. If $v_1, v_2, v_3 \in C$, then this probability is $\frac{1}{r^2}$, since we can imagine one v_i being pre-set, and each other v_j is chosen uniformly at random and independently from a space of size $\frac{1}{r}$. If at least two of v_1, v_2, v_3 are in C, then the one that is not can *be* pre-set, and by the same argument as above the probability that $Q_1 = Q_2 = Q_3$ is $\frac{1}{r^2}$.

Now if $v_i, v_j \in B$, $v_k \in C$ for some distinct $i, j, k \in \{1, 2, 3\}$, the probability that $Q_i = Q_j$ is exactly p, by our definition of p. Q_k is chosen uniformly at random and independently from a space of size $\frac{1}{r}$, so the probability that $Q_i = Q_j = Q_k$ is $p\frac{1}{r}$.

Finally, let us examine the case where $v_1, v_2, v_3 \in B$. Since these items were successfully stored in the table, they *cannot* all have been hashed to the same 2 locations. Therefore in this case $Pr(Q_1 = Q_2 = Q_3) = 0$.

We therefore have:

$$Pr(Q_1 = Q_2 = Q_3) - Pr(Q'_1 = Q'_2 = Q'_3) = \frac{\binom{|C|}{1}\binom{|B|}{2}}{\binom{|Y|}{3}}\left(p - \frac{1}{r}\right) - \frac{\binom{|B|}{3}}{\binom{|Y|}{3}}\frac{1}{r}$$

But we are looking at the case that $p - \frac{1}{r}$ is negligible. On the other hand $\frac{\binom{|B|}{3}}{\binom{|Y|}{3}}\frac{1}{r} = \frac{n'(n'-1)(n'-2)}{m(m-1)(m-2)r}$ which, since $r = \mathcal{O}(m^2)$, is non-negigible in m.

Therefore $|Pr(Q_1 = Q_1 = Q_3) - Pr(Q'_1 = Q'_2 = Q'_3)|$ and subsequently $\Delta((Q_1, Q_2, Q_3), (Q'_1, Q'_2, Q'_3))$ are also non-negligible in m.

4.2 Attack against PanORAMa and OptORAMa

In PanORAMa and OptORAMa, rather than each ORAM level containing a single Cuckoo Hash Table, each level has a number of equal-size bins, an Overflow Table and a (level-specific) Combined Stash. The bins, the Overflow Table and the Combined Stash are all implemented as Cuckoo Hash Tables. The Combined Stash Table contains the combined stashes of all bins on that level. The Overflow Table and the Combined Stash additionally have their own stashes. These stashes are removed from the level and reinserted into the ORAM.

Provided that items found in the Combined Stash are still searched for at each bin, the fact that the stashes of all bins in a given level are combined is not an issue.[14] However, the fact that the stashes of the Overflow Table and of the (level-specific) Combined Stash are removed from the level and re-inserted into the ORAM makes the protocols vulnerable to the attack described in this paper.

Like in the regular ORAM attack, let $u_1, \ldots, u_m$ be a sequence of distinct accesses of length $m \leq N - 3$ such that following this sequence of accesses, a level L_i is built with the the set $Y = \{u_1, \ldots, u_m\}$ as input.

Let T be the Overflow Hash Table,[15] and X be the set of items input the the Build function. X is unknown to the adversary, but it is guaranteed that $X \subseteq Y$. Let S be the set of stashed elements in the Overflow Hash Table.

Observe that if an index $x \in S$ is queried, PanORAMa and OptORAMa will find x before reaching L_i and will query a nonce in T instead. Therefore, the access sequence to the Overflow Hash Table in the ORAM is the same as that of a Stash-Resampling Cuckoo Hash Table.

[14] OptORAMa seaches in the Combined Stash *after* searching in the bins, so the access pattern in the bins will be the same for items that are later found in the Combined Stash. However, in PanORAMa, the Combined Stash is accessed *before* the bins are accessed and a random bin is chosen in the case that the data is found in the Combined Stash. Therefore, the access patterns in the individual bins are also vulnerable to a distinguishing attack based on the fact that stashed elements will not be searched for. This can simply be solved by searching the bins before searching the Combined Stash.

[15] The proof would work out the same if T was the Combined Stash Hash Table.

Since the Overflow Hash Table is not access-oblivious, to an adversary that knows $Y \supseteq X$, by Theorem 4, the ORAM protocols are not access-oblivious either. In particular, let the adversary choose distinct v_1, v_2, v_3 uniformly at random from Y. Let $A = (u_1, \ldots, u_m, v_1, v_2, v_3)$. Let $v'_1, v'_2, v'_3 \notin Y$ be distinct elements and $A' = (u_1, \ldots, u_m, v'_1, v'_2, v'_3)$. The access sequences of the ORAM on A and A' will have non-neglible statistical distance in m (and N).

5 Alibi: Secure Hierarchical ORAM with Reinserted Stashes

The basic problem arises when a stashed element is found *before* the appropriate level of the ORAM hierarchy is searched. As a successful criminal needs not only to be hidden in the location where they committed a crime, but also needs an alibi who claims to have seen them enacting their everyday life, likewise the stashed elements need not only hide their presence in the levels to which they are reinserted, but also need to hide their absence from the levels from which they came. To fix this problem, we need to ensure that even when an element cannot be *stored* at a certain level of the ORAM hierarchy (*i.e.,* because it falls in the cuckoo stash), it must still be *searched for* at this level. This way, the set of physical accesses at a level will always be chosen uniformly at random and be fully independent. Each element therefore needs to store a record of the locations where it would have been, and needs to be searched for in these locations if accessed.

There are some small subtleties here. First, an element needs to store the fact that it was ejected from a level not only when it is in the cache, but at least until this level is rebuilt or the item is searched for, since if it is looked up at *any* point before this level is rebuilt it needs to be searched for in this level. Second, it is entirely possible that the same element that had been stashed at some level L_i could be stashed again at some level L_k with $k < i$, *before* L_i is rebuilt or the element queried. Therefore each element needs to store the location of all levels from which it was ejected due to having fallen in the stash. Since there are $\ell \leq \log N$ levels in the hierarchical ORAM, it is possible to store which levels the item was ejected from using $\log N$ bits.

The flaw can be fixed using the following simple modification. For each element (v, x) the algorithm will additionally store a bit array $\mathfrak{e}$ of length ℓ, which records at which levels the item was "stashed."

Our solution modifies the generic hierarchical ORAM protocol of Figure 2; these modifications[16] are presented in Figure 5.

Lemma 6. *In the Alibi protocol presented in Figure 5 there is an invariant that given a tuple $(v, w, \mathfrak{e})$ stored at some level, $\mathfrak{e}[i] = 1$ if and only if v was stashed at level L_i during the last rebuild and v has not been queried by the ORAM*

[16] This protocol uses a slightly definition of Oblivious Hash Tables. Rather than returning a single array, Build returns a tuple (T_i, S_i), where T_i is the main table and S_i is the stash. Lookup only contains the non-stash locations.

- **Initializing records:** When initializing the ORAM, for each input tuple (v, x) store the tuple $(v, x, 0^\ell)$ in the ORAM.
- **On rebuilds:** When a hash table, T_i, is constructed at level i, suppose $(T_i, S_i) \leftarrow \mathsf{Build}(\mathsf{k}_i, X)$.
  - **Stashed records:** For each record $(v, x, \mathfrak{e}) \in S_i$, set $\mathfrak{e}[i] = 1$, and $\mathfrak{e}[j] = 0$ for $j = 1, \ldots, i-1$. Finally, insert $(v, x, \mathfrak{e})$ into the cache (or combined stash) as usual.
  - **Regular records:** For each record $(v, x, \mathfrak{e}) \in T_i$, set $\mathfrak{e}[i] = 0$, and $\mathfrak{e}[j] = 0$ for $j = 1, \ldots, i-1$.
- **On queries:** On input (v, x), initialize $\mathsf{found} = \mathsf{false}$, $\mathfrak{f} = 0^\ell$.
  - **Scan the cache:** If a record $(v, w, \mathfrak{e})$ is found in the cache, set $m = w$, $\mathsf{found} = \mathsf{true}$ and $\mathfrak{f} = \mathfrak{e}$ and delete the item from the cache.
  - **Search each level:** For i in $1, \ldots, \ell$
    - If $\mathsf{found} = \mathsf{true}$ and $\mathfrak{f}[i] = 0$ then set $Q_i \leftarrow \mathsf{Lookup}(\mathsf{k}_i, \mathsf{dummy} \circ t)$. Otherwise set $Q_i \leftarrow \mathsf{Lookup}(\mathsf{k}_i, v)$,
    - Probe locations $T_i[j]$ for $j \in Q_i$.
    - If there is a $j \in Q_i$, such that $T_i[j] = (v, w, \mathfrak{e})$, then set $m = w$, $\mathsf{found} = \mathsf{true}$ and $\mathfrak{f} = \mathfrak{e}$.
    - Execute $\mathsf{Delete}(\mathsf{k}_i, v, T_i)$
  - **Rewrite the cache:** If $x \neq \perp$ (*i.e.,* it was a write query), insert $(v, x, 0^\ell)$ into the cache. Otherwise insert $(v, m, 0^\ell)$ into the cache.

Fig. 5. Alibi Hierarchical ORAM protocol (delta to standard protocol of Figure 2)

since this rebuild. This invariant holds initially, after each query and after each rebuild.

Proof. By induction. This is initially true, as no items have been stashed and $\mathfrak{e}[i] = 0$ for all items.

If a level L_i is rebuilt, all levels L_j for $j < i$ will be rebuilt as empty levels. Therefore following the rebuild $\mathfrak{e}[j] = 0$ for all such levels, satisfying the invariant for these levels. For level L_i, some elements may be stashed after the rebuild, $\mathfrak{e}[i] = 1$ for exactly these elements, so the invariant is satisfied for level i. For any level L_j with $j > i$, the level has not been rebuilt and $\mathfrak{e}[j]$ is not modified, so the invariant will hold if it held before.

After a query $\mathfrak{e}[j]$ is set to 0 for all j, so $\mathfrak{e}[i]$ will only be 1 if there has not been a query since the last rebuild.

Therefore, by induction, this invariant always holds.

Theorem 5. *Let $(v_1, \cdots, v_m)$ be the sequence of indices that are looked up at level L_i with $i > 0$, between two subsequent rebuilds of that level. Then the Alibi protocol satisfies the following property. If $v_k = v'_k$ for some $k' < k$, then $Q_k = \mathsf{Lookup}(\mathsf{k}_i, \mathsf{dummy} \circ t)$ else $Q_k = \mathsf{Lookup}(\mathsf{k}_i, v_k)$.*

Proof. Immediately after L_i is rebuilt, all levels L_j for $0 < j < i$ are empty. Furthermore, the cache L_0 contains only elements from S_i, the stash of level i. Therefore, by Lemma 6 every element $(v, w, \mathfrak{e})$ either exists in level L_j for some $j \geq i$ or has $\mathfrak{e}[i] = 1$. This will remain true until L_i is rebuilt or v is queried. If $v_k = v$ is queried and v is stored at some level $j \geq i$, then it will be looked up at level i, *i.e.,* $Q_k = \mathsf{Lookup}(\mathsf{k}_i, v_k)$. Similarly, if $v_k = v$ is queried, and $\mathfrak{e}[i] = 1$ then when v is found $\mathfrak{f}[i]$ will be set to $\mathfrak{e}[i]$ so v will still be looked up at level L_i, *i.e.,* $Q_k = \mathsf{Lookup}(\mathsf{k}_i, v_k)$. In both cases $\mathfrak{e}[i]$ will be set to 0 (if it wasn't already) and it will be moved to a level $j < i$ (if it wasn't already). Only a rebuild on level L_i could change either of these facts. Therefore, until L_i is rebuilt, for any subsequent queries $v'_k = v$, a dummy item will be looked for in L_i, *i.e.,* $Q'_k = \mathsf{Lookup}(\mathsf{k}_i, \mathsf{dummy} \circ t)$.

Theorem 6. *The Alibi Stash-Reinserting ORAM protocol, when instantiated with an Oblivious Hash Table with a stash, is secure, i.e., it satisfies the security property of Definition 3.*

Proof. This follows similarly to the proof of Theorem 1. The ORAM satisfies two properties: (1) *No repeated accesses:* Each query is queried at most once to each level between rebuilds. This follows directly from Theorem 5. Any query in the form $v \in \mathcal{V}$ is queried at most once, since the theorem implies that any future accesses will be to dummy items. Any query in the form $\mathsf{dummy} \circ t$ is queried at most once, because t is incremented after each query. Therefore the lookups to the Oblivious Hash Table are *non-repeating.* (2) *Oblivious accesses:* The Oblivious Hash Table satisfies a property that the results of Lookup have distributions with negligible distances for all non-repeating access patterns. We know from (1) that in an ORAM the accesses to each Oblivious Hash Table

are indeed non-repeating. The ORAM only accesses the non-stash part of the Oblivious Hash Table, and since it is only accessing a subset, the distribution of access patterns of the ORAM on each level still differ negligibly. Furthermore, for an Oblivious Hash Table satisfying Full Obliviousness, the distribution of all memory accesses by an ORAM on each level differ negligibly.

Since each Oblivious Hash Table is built independently, the distance between distributions of their combined accesses will be at most the sum of the distances between distributions of accesses at each Oblivious Hash Table. Also, the distribution of accesses to the cache is the same each time, since the entire cache is scanned initially and a single item is written at the end, so this does not increase the distance between the access patterns of the ORAM. The distance between distributions of the entire sequences of t_{max} queries to an ORAM system with ℓ levels, will therefore be at most ℓt_{max} times that of any individual Oblivious Hash Table. Since the latter is negligible in N, and ℓt_{max} is polynomial in N, then the the total distance between distributions of ORAM access will also be negligible in N. Therefore the ORAM is oblivious.

Remark 9. It may initially seem that the proof of security above would apply to the flawed schemes as well. However, because the schemes *resample* the queries based on whether they were stored in the stash, the access pattern of the remaining table changes, and changes specifically in a way that depends on the structure of the table. We showed that in the case of Cuckoo Hashing this change causes a change in the combined set of accesses that is distinguishable.

Complexity: Since each element only needs to store one bit for each level, and there are at most $\log N$ levels, then the additional size of each element is increased by $\log N$ bits. Since the index is at least $\log N$ bits and the payload is $\Omega(\log n)$ the items still have the same asymptotic sizes so this does not change the asymptotic communication complexity. All of the modifications above only involve modifying or reading $\mathfrak{e}$ when v and/or x would also be read or modified. We have that $|\mathfrak{e}| = \mathcal{O}(|v|)$, $|\mathfrak{e}| = \mathcal{O}(|x|)$. Therefore the modification only increases communication costs by up to constant factors.

Correctness: The modifications do not change the output of the program, only the access patterns. The only operation that does not involve only modifications of $\mathfrak{e}$ is that during an access, if the item has already been found, the real item may be searched for in subsequent levels rather than a random item. This does not change the output of the program, since the value that was already found is the one that will be used.

Note that this fix also applies to PanORAMa and OptORAMa. Even though these protocols contain multiple Cuckoo Hash Tables at each level, it is possible to view the entire level as a single Oblivious Hash Table with a stash. (The stash of the level would be the union of the stashes of the Overflow Table and the level-specific Combined Stash Table).

6 Summary of Affected Papers

Goodrich et al. [GMOT12] introduced the idea of using Cuckoo tables with combined stashes for Hierarchical ORAM. This introduced the flaw described in this paper. Kushilevitz et al. [KLO12] introduced the alternative approach of reinserting elements from the stash into the ORAM ("cache the stash"). While there are differences between these approaches, in either case an element that was stashed will be found prior to the the level from which it was ejected and random locations accessed at this level instead. Therefore both approaches are vulnerable to our attack.

Lu and Ostrovsky [LO13] then used the stash-reinsertion of [KLO12] in their 2-server ORAM protocol, inheriting this vulnerability. Similarly Kushilevitz and Mour [KM19] created a 3-server ORAM that also uses cuckoo hashing (Instantiation 2) based on [KLO12], but using a shared stash [GMOT12] rather than reinserting the stash. This ORAM protocol is therefore vulnerable to the attack from this paper. Kushilevitz and Mour also present other multi-party ORAM protocols based on other techniques which are not subject to this attack.

Two alternative Hierarchical ORAM protocols were also published that avoided the flaw described in this paper. The Hierarchical ORAM protocol [MZ14] of Mitchell and Zimmerman uses a different model where the client can keep track of which level each item should be stored at. Knowing before-hand that an element does not exist at a certain level allows the algorithm to search for *pre-inserted dummy elements* at these levels. The data-structure therefore no longer needs to hide where data is stored, but only whether an element is real or a dummy, so any standard hash tables can be used instead of Cuckoo hashing. The two-tiered Hierarchical ORAM protocol of Chan et al. [CGLS17] then presented an alternative to cuckoo hashing with a stash. Instead, two hash tables existed, each with bins of size $\log^{\epsilon}(\lambda)$ for some constant $\epsilon \in (0.5, 1)$ and security parameter λ. They presented an oblivious construction in which elements would be placed in the first hash table if possible and in the second if not. They showed that the probability that an element could not be placed was negligible. Since this protocol used two-tier hashing rather than Cuckoo hashing with a combined stash it is immune to the attack we have presented.[17]

However, the flaw resurfaced again in the recent asymptotic breakthroughs of PanORAMa [PPRY18] and OptORAMa [AKL+20].[18] These achieved efficiency by storing most of the data in small bins, which are small enough to be sorted without increasing the asymptotic performance, while remaining items are placed

[17] Chan et al. also presented a concrete instantiation of Goodrich and Mitzenmacher's ORAM protocol in an appendix of the full version of their paper. The protocol they present uses a Cuckoo Hash Table at each level and a shared stash, so is vulnerable to the attack described in this paper. However, they recommend, somewhat clairvoyantly, that since Cuckoo hashing is complex and hard to prove correct, that their two-tier hash-table protocol should be used rather than the Cuckoo-hashing protocol.

[18] In response to our preprint, Asharov et al. have updated the OptORAMa paper to include a fix.

in an overflow pile. Each of these bins is implemented as a cuckoo table and stashes are shared, but the combined stash for the bins is kept at the same level as the bins. Therefore it is possible to search the bins for the stashed elements and then to access the single-level combined stash, so the bin tables are not vulnerable to this attack. However, in both papers, the overflow and single-level combined stash cuckoo tables both have stashes that are re-inserted into the ORAM data structure. They are therefore vulnerable to the variant of our attack in Section 4.

Our attack *does not* affect the tree-based ORAM protocols, such as Binary Tree ORAM [SCSL11], Path ORAM [SvDS+13] and Circuit ORAM [WCS15], as these do not use cuckoo hashing.

In summary, this flaw existed in the the ORAM literature for almost a decade and has affected six significant protocols, including the most recent asymptotic breakthroughs. The fact that such a flaw could exist unnoticed for so long motivates the development of simpler protocols for oblivious data structures.

References

[ADW14] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica*, 70(3):428–456, 2014.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In *EUROCRYPT*, pages 403–432. Springer, 2020.

[BMD+17] 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.

[CGLS17] 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*, pages 660–690. Springer, 2017.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535, 2017.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[GM11] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587. Springer, 2011.

[GMOT12] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167. SIAM, 2012.

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

[JHOvD17] Tara Merin John, Syed Kamran Haider, Hamza Omar, and Marten van Dijk. Connecting the dots: Privacy leakage via write-access patterns to the main memory. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156. SIAM, 2012.

[KM19] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In *PKC*, pages 3–33. Springer, 2019.

[KMW09] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient RAM-model secure computation. In *S&P*, pages 623–638. IEEE, 2014.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396. Springer, 2013.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90. Springer, 2017.

[Mit09] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *ESA*, pages 1–10, 2009.

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *STACS*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, volume 97, pages 294–303. Citeseer, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

[Ost92] Rafail Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, 1992.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, pages 871–882. IEEE, 2018.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519. Springer, 2010.

[SCSL11] Elaine Shi, T-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214. Springer, 2011.

[SGF17] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from Intel SGX. *IACR Cryptol. ePrint Arch.*, 2017:549, 2017.

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

[WCS15] Xiao Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, pages 850–861, 2015.

[WHC+14] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, pages 191–202. ACM, 2014.

Supplementary Material

A Distinguishing distributions

In this section, we review a basic fact that if two distributions are statistically different, and supported on polynomial-sized sets, then they are polynomial-time distinguishable.

Lemma 7. *Let $\{X_n\}$, $\{Y_n\}$ denote two sequences of distributions supported on polynomial-sized sets, i.e., there is a constant c, such that $\max(|X_n|, |Y_n|) < n^c$. In addition, assume that X_n and Y_n are efficiently samplable.*

Then if $\Delta(X_n, Y_n)$ is non-negligible, the distributions $\{X_n\}$ and $\{Y_n\}$ are polynomial-time distinguishable.

Proof. Consider the following maximum likelihood distinguisher, D. Let $W = \operatorname{supp}(X_n) \cup \operatorname{supp}(Y_n)$, and $m = |W|$. Define

$$p_z \stackrel{\text{def}}{=} \Pr[X_n = z]$$
$$q_z \stackrel{\text{def}}{=} \Pr[Y_n = z]$$

Fix $t = \operatorname{poly}(n)$.

Recall that if $W = X_n \cup Y_n$,

$$\begin{aligned}\sum_{w \in W} \max(p_w, q_w) &= \frac{1}{2} \sum_{w \in W} [[\max(p_w, q_w) + \min(p_w, q_w)] + [\max(p_w, q_w) - \min(p_w, q_w)]] \\ &= \frac{1}{2}\left[2 + \sum_{w \in W} [\max(p_w, q_w) - \min(p_w, q_w)]\right] \\ &= \frac{1}{2}[2 + 2\Delta(X_n, Y_n)] \\ &= 1 + \Delta(X_n, Y_n)\end{aligned}$$

First, D will estimate the frequency of elements in both X_n and Y_n by sampling. First D will draw tm samples from X_n, let X_{sampled} denote the multiset corresponding to these samples. Similarly D will draw tm samples from Y_n. Let Y_{sampled} be the multiset corresponding to these samples.

Then D defines

$$\tilde{p}_w \stackrel{\text{def}}{=} \frac{\text{number of times } w \text{ occurred in } X_{\text{sampled}}}{tm}$$
$$\tilde{q}_w \stackrel{\text{def}}{=} \frac{\text{number of times } w \text{ occurred in } Y_{\text{sampled}}}{tm}$$

Finally, given a sample z from a distribution $Z \in \{X_n, Y_n\}$, the adversary will guess

$$A(z) = \begin{cases} X \text{ if } \tilde{p}_z \geq \tilde{q}_z \\ Y \text{ if } \tilde{p}_z < \tilde{q}_z. \end{cases}$$

A Hoeffding bound shows that

$$\Pr\left[|\tilde{p}_z - p_z| > \delta\right] < 2e^{-2mt\delta^2}$$

and similarly

$$\Pr\left[|\tilde{q}_z - q_z| > \delta\right] < 2e^{-2mt\delta^2}$$

Fix $\delta > 0$, and define

$$G \stackrel{\text{def}}{=} \{z \in W \mid |p_z - q_z| > 2\delta\}$$
$$B \stackrel{\text{def}}{=} \{z \in W \mid |p_z - q_z| \leq 2\delta\}$$

Now, notice that

$$\max(p_z, q_z) - 2\delta < \min(p_z, q_z) \qquad \text{for all } z \in B\ . \tag{1}$$

The Hoeffding bounds give

$$\Pr\left[\max(p_z, q_z) = \max\left(\tilde{p}_z, \tilde{q}_z\right)\right] > 1 - 2e^{-2mt\delta^2} \qquad \text{for } z \in G \tag{2}$$

Let $\epsilon = \max_z\left(\left|\Pr(X_n = z) - \Pr(Y_n = z)\right|\right)$. Thus $\epsilon \geq \frac{\Delta(X_n, Y_n)}{m}$, which is non-negligible.

$\Pr[\ A \text{ is correct }]$

$$\begin{aligned}
&= \frac{1}{2}\left[\sum_{z\in Z} \Pr\left[\max\left(\tilde{p}_z, \tilde{q}_z\right) = \max\left(p_z, q_z\right)\right] \max\left(p_z, q_z\right) + \sum_{z\in Z} \Pr\left[\max\left(\tilde{p}_z, \tilde{q}_z\right) \neq \max\left(p_z, q_z\right)\right] \min\left(p_z, q_z\right)\right] \\
&= \frac{1}{2}\left[\sum_{z\in Z} \Pr\left[\max\left(\tilde{p}_z, \tilde{q}_z\right) = \max\left(p_z, q_z\right)\right] \max\left(p_z, q_z\right) + \sum_{z\in B} \Pr\left[\max\left(\tilde{p}_z, \tilde{q}_z\right) \neq \max\left(p_z, q_z\right)\right] \min\left(p_z, q_z\right)\right] \\
&\geq \frac{1}{2}\left[\sum_{z\in G} \Pr\left[\max\left(\tilde{p}_z, \tilde{q}_z\right) = \max\left(p_z, q_z\right)\right] \max\left(p_z, q_z\right) + \sum_{z\in B} \left[\max\left(p_z, q_z\right) - 2\delta\right]\right] \\
&\geq \frac{1}{2}\left[\left(1 - 2e^{-2mt\delta^2}\right) \sum_{z\in Z} \max\left(p_z, q_z\right) - 2m\delta\right] \\
&= \frac{1}{2}\left[\left(1 - 2e^{-2mt\delta^2}\right)\left[1 + \Delta(X_n, Y_n)\right] - 2m\delta\right] \\
&= \left(1 - 2e^{-2mt\delta^2}\right)\left[\frac{1}{2} + \frac{1}{2}\Delta(X_n, Y_n)\right] - m\delta
\end{aligned}$$

Which is a non-negligible advantage for sufficiently large t and sufficiently small δ.

B Acknowledgements

This research was sponsored in part by ONR grant (N00014-15-1-2750) "SynCrypt: Automated Synthesis of Cryptographic Constructions". This research was supported in part by DARPA under Cooperative Agreement No: HR0011-20-2-0025, NSF-BSF Grant1619348, US-Israel BSF grant 2012366, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.