Weighted Oblivious RAM, with Applications to Searchable Symmetric Encryption

Léonard Assouline, Brice Minaud

Weighted Oblivious RAM, with Applications to Searchable Symmetric Encryption

Léonard Assouline and Brice Minaud

École Normale Supérieure, PSL University, CNRS, Inria, Paris, France

Abstract. Existing Oblivious RAM protocols do not support the storage of data items of variable size in a non-trivial way. While the study of ORAM for items of variable size is of interest in and of itself, it is also motivated by the need for more performant and more secure Searchable Symmetric Encryption (SSE) schemes.

In this article, we introduce the notion of *weighted* ORAM, which supports the storage of blocks of different sizes. In a standard ORAM scheme, each data block has a fixed size B. In weighted ORAM, the size (or *weight*) of a data block is an arbitrary integer $w_i \in [1, B]$. The parameters of the weighted ORAM are entirely determined by an upper bound B on the block size, and an upper bound N on the total weight $\sum w_i$ of all blocks—regardless of the distribution of individual weights w_i. During write queries, the client is allowed to arbitrarily change the size of the queried data block, as long as the previous upper bounds continue to hold.

We introduce a framework to build efficient weighted ORAM schemes, based on an underlying standard ORAM satisfying a certain suitability criterion. This criterion is fulfilled by various Tree ORAM schemes, including Simple ORAM and Path ORAM. We deduce several instantiations of weighted ORAM, with very little overhead compared to standard ORAM. As a direct application, we obtain efficient SSE constructions with attractive security properties.

1 Introduction

When sensitive data is stored in an untrusted environment, encryption is not enough. The pattern of memory accesses to encrypted data can reveal a great deal about its contents. In some settings, observing the pattern of memory accesses can allow a honest-but-curious host server to fully reconstruct the contents of an encrypted database [GLMP19]; in others, measuring cache misses can enable an attacker to recover secret key material [WHS12]. Untrusted environments where an adversary may be able to observe memory accesses, partially or completely, arise in many common scenarios. These include private information stored in an external cloud service, trusted enclaves running on an untrusted computer, or even public clouds where memory caches are shared across multiple tenants. In all these settings, security requires to hide not only the contents of each data item, but also which item is accessed.

Oblivious RAM (ORAM) protocols provide a powerful tool to fully hide memory access patterns. The notion of ORAM was introduced by Goldreich and Ostrovsky [GO96], motivated by a scenario where a processor accesses untrusted memory. The processor operates in a RAM model of computation: it wishes to access memory words at arbitrary addresses. Naturally, memory words have a fixed size. In line with its historical motivation, ORAM is normally viewed as storing items of fixed size.

However, in many potential applications of ORAM, it is natural to consider items of variable size. Suppose for instance that a client wishes to store private files on an external cloud storage service. Different files may have different sizes; it may also be the case that the size of a file varies with time. This motivates the idea of ORAM for variable-size items.

The Trivial Approach. Of course, it is possible to generically emulate the storage of variable-size files using a memory allocation scheme for fixed-size items. In our case, using an ORAM for items

of fixed size B, the most natural approach is to split each file into chunks of size B. Each chunk is then stored as a separate data item (called a block) within the ORAM, on the server side. To retrieve a file, the client simply queries all chunks corresponding to the desired file.

This simple variable-size-to-fixed-size reduction is not always satisfactory. A first issue relates to padding. If $B > 1$, before files can be split into chunks of size B, they must be padded to a multiple of the block size B. If many files are much smaller than the block size, padding becomes expensive. Both motivating applications given below show examples where padding can be prohibitive.

To reduce the cost of padding, it is tempting to reduce the block size B. However, this has several drawbacks, asymptotically and in practice. Let us start with theoretical ones. For a fixed total amount of data, reducing B increases the ORAM overhead (i.e. the ratio between the communication cost of the ORAM scheme, and the cost of an insecure exchange), since the overhead scales with the number of blocks. For example, an ORAM storing N blocks of size 1 typically has an overhead in polylog N; whereas with N/B blocks of size B, the overhead becomes polylog(N/B). In later applications such as length-hiding ORAM, or ZeroSSE (Section 6.2), B can be very large, which makes the difference significant. In theory, larger block sizes are also preferable: for instance, Path ORAM achieves optimal $\mathcal{O}(\log n)$ overhead if the block size is $\Omega(\lambda^2)$ bits, but its overhead is $\mathcal{O}(\log^2 n)$ if the block size is $\Theta(\lambda)$ bits (where λ is the security parameter).

In practice, setting the block size to be very small, say a single memory word of 128 bits, has a deeper impact that is easy to overlook, but much more impactful in practice that the asymptotic difference above. Modern computers can only fetch memory from disk at the granularity level of a page, typically 4kB. This is enforced at all levels: by the operating system, in caches, and at the physical disk layer (for both HDDs and SSDs). When fetching many 128-bit words at random locations in the ORAM scheme, the server actually fetches the entire page for each. In each of those pages, only a fraction 1/256 of the data in the page is actually useful (128 bits out of 4kB). This results in very poor I/O efficiency, which correlates directly with disk throughput [BBF+21]. The issue is easy to overlook because it is not reflected in the simple Random-Access Machine model of computation that is used to compute asymptotics, where all memory accesses have unit cost. But it has a very large impact in practice. This is well-known in SSE literature, where an entire branch of the area studies memory efficiency [ANSS16, MM17, BBF+21, MR22]. In ORAM literature, the PHANTOM implementation of Path ORAM uses blocks of size one page, likely for the same reason [MLS+13]. Reading many tiny items at random memory locations is extremely inefficient, losing a factor up to B in throughput (when the bottleneck in throughput does not come from communication bandwidth).

If one thinks of storing entire documents in the context of a private online storage service, having many documents much smaller than the page size is rather unlikely. But in other applications, it is quite realistic. A case in point is the use of ORAM for Searchable Symmetric Encryption (SSE).

Motivating Application 1: Searchable Symmetric Encryption (SSE). The goal of SSE is to enable a client to outsource the storage of an encrypted database to an untrusted server, while being able to securely search the data. At minimum, the client is able to issue search queries asking for all entries that match a given keyword. To realize this functionality, for efficiency reasons, virtually all modern SSE constructions rely on a *reverse index*. The reverse index records, for each keyword, the list of identifiers of entries that match the keyword.

The majority of SSE solutions accept to leak the *search pattern* and *access pattern* of the client: that is, they leak to the server the repetition of queries, and the identifiers of documents matched by each query. This allows those constructions to trade off privacy for efficiency and scalability. Nevertheless, revealing access patterns to the server can be quite damaging, and has led to a number of attacks [CGPR15, GLMP19]. Those attacks have in turn motivated SSE approaches that rely on ORAM [GMP16, KMO18].

The most natural way to avoid leaking the search pattern is to store the reverse index in an ORAM. In that scenario, the "files" to be stored on an ORAM are actually the list of matches for a

given keyword in an SSE scheme. For some databases, there may be many keywords that uniquely identify a file, or that match only a few files. In other words, there may be many lists much smaller than the block size B of the ORAM.

In practice, this is actually a major roadblock. As argued earlier, it is desirable to have a relatively large ORAM block size, at least one memory page. On the other hand, the identifier of an entry can be set to 64 bits, or even less. This means that a single ORAM block is 512 times larger than the minimal list size. If, say, many keywords match less than 10 entries in the database, padding those lists to the block size blows up their storage by a factor more than 50. More generally, if we set p to be the page size, measured in number of identifiers per page, then padding means that server storage grows at least in $\mathcal{O}(pN)$, where N is the size of the plaintext reverse index; whereas we would like to achieve linear storage $\mathcal{O}(N)$. Of course, with $p = 512$ as earlier, the practical difference is quite large.

Addressing that problem is not an easy task. In SSE literature, avoiding the cost of padding to the page size has been the focus of several recent works [BBF+21, MR22]. Those works have motivated the creation of *weighted* memory allocation schemes, that can accommodate items of variable size, including weighted cuckoo hashing [BBF+21], and weighted two-choice allocation [MR22]. However, there is no weighted ORAM. This means that in order to use ORAM with SSE, current options are either to choose a block size much smaller than the page size, or to suffer a prohibitive padding overhead for some data distributions—both of which are undesirable.

Motivating Application 2: Length-Hiding ORAM. Let us go back to the scenario where a client wishes to store private files of various sizes on a honest-but-curious cloud server. As noted earlier, the simplest way to hide access patterns is to store the files in an ORAM. Each file is split into chunks of size B, and each chunk is stored in a separate ORAM block. In order to fetch a file, the client queries all chunks of the file to the ORAM. When a file is queried, the only information leaked to the server is the number of chunks of the file.

In some settings, even that much information may be too much information. For instance, the number of chunks of a file might be enough information to uniquely identify the file [CGPR15]. In that case, repeated accesses to the file are leaked to the server. This reveals the access pattern of the client to the files, defeating the purpose of ORAM. More subtly, the length of answers to certain types of database queries can be enough to infer the entire contents of encrypted database [GLMP18]. Traffic analysis attacks are another example of using length information to infer sensitive data [DCRS12]. Attacks that exploit length information can be particularly insidious, because traditional encryption does not attempt to hide length.

If leaking the lengths of the files is judged to be too damaging, the client may wish to use additional mechanisms to protect their privacy. Going back to our running example about private file storage, the simplest and most secure protection is to mandate that, whenever a file is accessed, the client should query as many chunks as the size of the *largest* file. In that case, only the number of chunks of the largest file is leaked to the server—or an upper bound on that number.

Let N be the total size of the files to be stored on the remote server. Let B be the ORAM block size, and let U be an upper bound on the size of the largest file (all quantities are counted in number of memory words). The overhead of ORAM constructions typically scales in $\mathsf{Polylog}(n)$, where n is the number of blocks stored in the ORAM. Setting aside padding issues for a moment, with block size B, we have $n = N/B$. In order to minimize the overhead, it would be attractive to simply set $B = U$. But here again, we would run into padding issues: most files might be much smaller than the largest file. The optimal solution would be a *weighted* ORAM able to accommodate files of arbitrary size up to U, with an overhead $\mathsf{Polylog}(N/U)$, or optimally, $\log(N/U)$.

1.1 Our Contributions

The discussion so far leads to the following question: can we devise a *weighted* ORAM—that is, an ORAM that natively accommodates items of variable size? Beside the motivating applications given in the introduction, the existence of weighted ORAM may be viewed as a natural question: it fits within a long line of work on weighted allocation mechanisms, both within and outside cryptography, such as [TW07, BFHM08, ANSS16, BBF+21, MR22].

We answer the previous question in the affirmative, and build a weighted ORAM. Our construction naturally handles not only items of different sizes, but items whose size varies with time, without the need for padding. To state the result precisely, let us introduce some notation. In the remainder, an atomic item stored within the ORAM is called a *block*. Let B denote an upper bound on the block size. Unlike traditional ORAMs, blocks can take any size in $[1, B]$. We will sometimes call the size of a block its *weight*. Let $w_i \in [1, B]$ denote the weight of the i-th block. Let m be the total number of blocks. Let N be an upper bound on the *total weight* $\sum_{i \leq m} w_i$. We want to build and ORAM that can accommodate *any* vector $\mathbf{w} = (w_i)_{i \leq m}$ of weights, as long as the following two conditions are fulfilled.

Condition 1. Every block w_i has weight at most B;
Condition 2. The total weight $\sum w_i$ is at most N.

For ease of exposition, we will assume that the number of blocks m is fixed, but our constructions can be easily adapted to a variable number of blocks, so long as the previous two conditions continue to hold. The parameters of our ORAM constructions will depend *only* on B and N; crucially, they do not depend on the distribution of the weight vector $\mathbf{w}$.

The interface of our weighted ORAMs is identical to standard ORAM: to retrieve a block, the client queries an identifier of the block (e.g. a virtual memory address). When writing a block, the client also inputs new data for the block. This data need not be of the same size as the data originally associated to the block identifier. The client can freely change the size of a block with every access, as long as Conditions 1 and 2 remain true.

As our main contribution, we build a weighted ORAM in the sense given above. In fact, we show a significantly stronger result. Many standard Tree-based ORAM algorithms admit a natural extension to handle blocks of variable size: at setup, the ORAM is dimensioned as if to accommodate N/B blocks of size B, but instead receives an arbitrary number of blocks of variable size bounded by B, with total size N. These blocks are read and written through the ORAM in essentially the same way as in the original, fixed-block size Tree ORAM, except for minor alterations to reflect the fact that blocks do not have the same size.

The main obstacle with that approach is technical. While Path ORAM is one of the most attractive solutions for practical Tree ORAM [SvS+13], its correctness proof is notoriously difficult—prompting the introduction of Simple ORAM as a less efficient variant that allows for a simpler correctness proof [CP13]. Our main result is to show that the natural weighted extensions of several existing Tree ORAM schemes, including Path ORAM and Simple ORAM, are in fact correct. For that purpose, we introduce a general framework: we prove that as long as a Tree ORAM fulfills a certain structural property, its weighted extension preserves correctness. The centerpiece of the proof is a Schur-convexity argument, which ultimately reduces the correctness of the weighted extension to that of the original ORAM. (An overview of the proof argument is given in Section 4.3, before the formal proof.) Practical experiments show that our weighted ORAM construction behaves in line with the analysis.

As an application of weighted ORAM, we build two SSE schemes, ZeroSSE and BlockSSE. Unlike most of SSE literature, both constructions completely hide access patterns. To our knowledge, ZeroSSE is the only construction that leaks neither the access pattern nor the size of retrieved objects, with full correctness. (The only other construction that we are aware of, in [KMO18], pays the price of

having a non-negligible correctness failure probability.) BlockSSE hides access pattern, but not the size of retrieved objects. To our knowledge, it is the only ORAM-based SSE with worst-case server storage $\mathcal{O}(N)$, rather than $\mathcal{O}(BN)$, where B is the ORAM block size.

Our main result builds on Tree ORAMs, because of their higher practical efficiency compared to hierarchical ORAMs. This makes tree-based construction currently more attractive for applications such as SSE. Nevertheless, it is worth remarking that the position map of a weighted Tree-based ORAM, as we have built, has blocks of fixed size. Hence, it can be stored using any standard ORAM scheme, not necessarily tree-based. In particular, from a more theoretical perspective, the position map can be stored using an optimal ORAM with logarithmic overhead, following the groundbreaking result of Asharov *et al.* [AKL+20]. This results in a weighted ORAM with logarithmic overhead. The question of building weighted hierarchical ORAM schemes is briefly discussed in Appendix D.

As another direct application of our construction, setting the block size B of our weighted ORAM to be equal to an upper-bound bound U on the size of the largest item to be stored in the ORAM, we immediately deduce from our construction weighted Path ORAM a length-hiding ORAM with communication overhead $\mathcal{O}(\log^2(N/U))$. If we use an optimal (standard) ORAM for the position map, as indicated above, we obtain a length-hiding ORAM with communication overhead $\log(N/U)$. This overhead is optimal, since such a goal includes as a special case the setting where all blocks have size U, and is thus subject to known ORAM lower bounds for an ORAM storing N/U blocks [GO96, LN18].

1.2 Related work

While there is a rich literature on ORAM, surprisingly little of it deals with objects of variable size. To the best of our knowledge, only two articles mention this subdomain of ORAM.

In [RAC16], Roche *et al.* present the first ORAM that stores objects of variable size. Their goal is to build a remote data structure that satisfies the security requirements of ORAM, and in addition allows for secure deletion of items and history independence. In other words, in the case of a *total leakage of the structure* (such an event is referred to as a *catastrophic attack*):

- Items that have been deleted by the client can never be recovered through leaked data.
- The internal structure does not reveal information about which elements were last accessed.

The data structure is built on top of a weighted ORAM. However, their construction for such an ORAM is limited: obliviousness and correctness (i.e. the client-side stash overflows with negligible probability) can be proven only if the size of the blocks follow a geometric probability distribution. In comparison, although we assume that block sizes are bounded by B, we do not need to assume anything on the distribution of block sizes. In more detail, there are two limitations to the assumptions of [RAC16]. First, many common distributions are not upper-bounded by a geometric distribution, for instance Zipf distributions. Second and more fundamentally, the ORAM user has no reason in general to pick item sizes independently, or to pick them from the same distribution. The construction of [RAC16] was designed with a specific use case in mind; its applicability beyond that use case is limited.

Another construction of ORAM for objects of variable size may be found in [LHL+18]. Their construction is also based on Tree ORAMs. The idea is to allow block size to be equal to a multiple of some value s (padding up to a multiple if needed), and to store all "splinters" of size s of a block along the same path from root to leaf. This construction has the strong requirement of a trusted proxy that shuffles blocks during certain operations. Moreover, the construction is flawed: cf. Appendix C.

1.3 Organization of the paper

In Section 3 we recall the definitions of ORAM, SSE, and Schur convexity, a tool we will use in our proof. Section 4 is where we state our generic criterion for converting a standard ORAM into

one that supports objects of variable size, and prove our main result. Concrete examples of known ORAM schemes that we can turn into weighted ORAM are presented in Section 5. An alternative generic construction that converts any ORAM into a weighted ORAM at the cost of a small blowup is presented in Appendix B. We discuss applications to the field of SSE in Section 6.

2 General Preliminaries

Throughout this work, memory size will be counted as a number of *memory words*. It is assumed that a memory word is large enough to store any address in memory. In practical applications, one may think of 64-bit or 128-bit words. Algorithms will be considered in the RAM model, where accessing an arbitrary memory word costs $O(1)$ operations.

The security parameter is denoted by λ. A quantity is said to be *negligible*, denoted $\mathrm{negl}(\lambda)$, if it is $O(\lambda^{-c})$ for every constant c. A probability is said to be *overwhelming* if it is $1 - \mathrm{negl}(\lambda)$. It is always assumed that the number of blocks n stored in the ORAM satisfies $n \geq \lambda$, so that any quantity that is $\mathrm{negl}(n)$ is also $\mathrm{negl}(\lambda)$.

When an algorithm A with input x is probabilistic, we may sometimes explicitly write the random coins used by A as an input of A, separated by a semicolon, as in $A(x; r)$.

2.1 Majorization and Schur Convexity

Given a vector $\mathbf{v}$ in $\mathbb{R}^m$, we denote by $\mathbf{v}^{\downarrow} \in \mathbb{R}^m$ the vector with the same components, sorted in decreasing order.

Definition 1 (Majorization order). *Let $\mathbf{v}$, $\mathbf{w}$ be two vectors in $\mathbb{R}^m$ such that $\sum_{i=1}^m v_i = \sum_{i=1}^m w_i$. The vector $\mathbf{w}$ is said to* majorize *$\mathbf{v}$, written $\mathbf{v} \prec \mathbf{w}$, if:*

$$\forall k \in [1, m], \quad \sum_{i=1}^{k} v_i^{\downarrow} \leq \sum_{i=1}^{k} w_i^{\downarrow}.$$

Definition 2 (Schur convexity). *Let $f : \mathbb{R}^m \mapsto \mathbb{R}$. The map f is said to be* Schur-convex *if it is non-decreasing for the majorization order. That is, for any two vectors $\mathbf{v}$, $\mathbf{w}$ with $\sum_{i=1}^m v_i = \sum_{i=1}^m w_i$,*

$$\mathbf{v} \prec \mathbf{w} \Rightarrow f(\mathbf{v}) \leq f(\mathbf{w}).$$

Definition 3 (Convexity). *Let $f : \mathbb{R}^m \mapsto \mathbb{R}$. The map f is said to be* convex *if for any two vectors $\mathbf{v}$, $\mathbf{w}$ in $\mathbb{R}^m$, and any α in $[0, 1] \subset \mathbb{R}$, it holds that:*

$$f(\alpha\mathbf{v} + (1 - \alpha)\mathbf{w}) \leq \alpha f(\mathbf{v}) + (1 - \alpha) f(\mathbf{w}).$$

Definition 4 (Symmetry). *Let $f : \mathbb{R}^m \mapsto \mathbb{R}$. The map f is said to be* symmetric *if for any vector $\mathbf{v} \in \mathbb{R}^m$, and any permutation matrix P over m elements, $f(\mathbf{v}) = f(P\mathbf{v})$.*

The link between convexity and Schur convexity is visible in the next lemma.

Lemma 1. *Let $f : \mathbb{R}^m \mapsto \mathbb{R}$. If f is symmetric and convex, then it is Schur-convex.*

We refer the reader to [MOA79] for a detailed presentation of the theory of majorization, including a proof of Lemma 1.

3 ORAM Preliminaries

3.1 Weighted Oblivious RAM

A weighted ORAM, also written wORAM, is a pair of client-server protocols $(\texttt{Setup}, \texttt{Access})$, defined as follows.

- $\texttt{Setup}(N, B, \mathsf{D})$ takes as input an upper bound N on the total amount of data, a block size B, and a set D of pairs of the form $(\mathsf{a}_i, \mathsf{data}_i)$, where the a_i's are pairwise distinct *addresses*, and data_i is arbitrary data of size $|\mathsf{data}_i| \in [1, B]$. `Setup` outputs an initial client state and initial server state.
- $\texttt{Access}(\mathsf{op}, \mathsf{a}, \mathsf{data})$ takes as input an operation $\mathsf{op} \in \{\text{read}, \text{write}\}$, an address a, and some data data of size $|\mathsf{data}| \in [1, B]$. If $\mathsf{op} = \text{read}$, `Access` outputs to the client the data last written to address a. If $\mathsf{op} = \text{write}$, `Access` replaces the data written at address a by data. `Access` may also update the client and server states.

We say that $\texttt{Setup}(N, B, \mathsf{D})$ is *legal* if the total amount of data in D (*i.e.* the sum of the sizes of the data_i's) is at most N. Likewise, we say that $\texttt{Access}(\mathsf{op}, \mathsf{a}, \mathsf{data})$ is *legal* if address a was defined during setup, and in the case that $\mathsf{op} = \text{write}$, if $|\mathsf{data}| \in [1, B]$, and the total amount of data contained in the database after replacing the data at address a by data remains of size at most N. On the other hand, it is *not* required that the size of data matches the size of the data previously written at a.

Definition 5 (Correctness). *A wORAM scheme is said to be* correct *if, given a legal setup and any sequence of legal access operations, a read access at address* a *outputs the data last written to address* a*, except with negligible probability.*

Definition 6 (Security). *A wORAM scheme is said to be* secure *if, given any two legal sequences of operations of the same length* $(\texttt{Setup}(N, B, \mathsf{D}), \texttt{Access}(\mathsf{op}_1, \mathsf{a}_1, \mathsf{data}_1), \ldots, \texttt{Access}(\mathsf{op}_k, \mathsf{a}_k, \mathsf{data}_k))$ *and* $(\texttt{Setup}(N, B, \mathsf{D}'), \texttt{Access}(\mathsf{op}'_1, \mathsf{a}'_1, \mathsf{data}'_1), \ldots, \texttt{Access}(\mathsf{op}'_k, \mathsf{a}'_k, \mathsf{data}'_k))$*, the views of the server arising from each sequence are computationally indistinguishable.*

A few remarks are in order. First, although we have defined `Setup` and `Access` as general client-server protocols, it is common in ORAM to require that the server behave like a passive memory, allowing only read and write accesses. That is, the client only ever asks the server to read or write specific data at a specific address. The server performs no computation of its own. Although this is not required in the previous definition, the wORAM schemes in this work are in that model.

Second, it is assumed that the contents of all memory locations on the server are encrypted using IND-CCA encryption, with a key known only to the client. Whenever the client accesses a memory location, they can reencrypt the data at that location, so that the server cannot learn the contents of any memory location, or whether it was changed during the access. As a result, the only way the server can infer information is by observing which locations the client queries in server memory. That is why the security definition of wORAM (following that of ORAM) focuses only on memory locations.

Finally, note that standard ORAM is the special case of wORAM where all addresses store data of the same size B. If the total amount of data is N, then such a scheme contains $n = N/B$ blocks. Throughout this paper, the letter N is reserved for the total amount of data. In the case of standard ORAM, the letter n is used for the number of blocks, with $n = N/B$.

3.2 Tree ORAM

We build wORAM by altering standard ORAM schemes following the *Tree ORAM* paradigm. In this section, we provide a high-level algorithmic view of that paradigm. That view is purposefully

designed to accomodate several existing Tree ORAM schemes. It will also lay the groundwork for the construction of wORAM in the next section.

Existing Tree ORAM schemes are standard ORAMs, designed to store items of fixed size. In a Tree ORAM, to store n items of size B, the server creates a full binary tree with n leaves. (From now on, we assume n is a power of 2, increasing to the next power of 2 if necessary.) Throughout the article, the root of the tree is viewed as being at the top, and leaves as being at the bottom of the tree. Given a leaf l of the tree, the path from the root to the leaf l is denoted by $\mathcal{P}(l)$.

Each node of the tree, also called a *bucket*, can store up to Z data blocks of size B. Nodes are always padded to be of size ZB before being stored (encrypted) on the server.

In addition to the tree, the server may also store a *stash*, which may contain additional data blocks that could not fit in the tree. In the remainder, we view the stash as a special node directly above the root. This is relevant in two situations. First, there may be cases where a node is full (*i.e* it contains Z items), and where additional items need to be pushed to the parent node; if this happens at the root level, overflowing items are pushed to the stash. Second, whenever we consider the path $\mathcal{P}(l)$ from some leaf l to the root in the tree, we implicitly (and slightly abusively) also consider the stash to be part of the path. The stash is always padded to some upper bound R, before being stored (encrypted) on the server.

To each item with address a is associated a leaf of the tree $\texttt{pos}(a)$. The array mapping each address a to the corresponding leaf $\texttt{pos}(a)$ is called a *position map*. For now, we will assume the position map is stored by the client. By design, Tree ORAMs maintain the following invariant at all times: the item at address a is stored in one of the nodes on the path $\mathcal{P}(\texttt{pos}(a))$ from the root to the leaf $\texttt{pos}(a)$ (including the stash, as noted earlier).

During setup, each item with address a is stored in the leaf $\texttt{pos}(a)$; or if it is full, in the lowest parent of $\texttt{pos}(a)$ that is not yet full. To access item a, the client retrieves $\texttt{pos}(a)$ from the position map, then reads the path $\mathcal{P}(\texttt{pos}(a))$ on the server. Thanks to the invariant, that path contains the item a. Item a is then assigned a new uniformly random leaf. Finally, a special *eviction* procedure is called, which re-inserts item a somewhere on the path to its newly assigned leaf, and may also move other items.

Pseudo-code for the `Evict` procedure is given in Algorithm 1, with additional parameters Z (the number of blocks per bucket, specifided by the Tree ORAM scheme; to reflect the fact that Z is an internal parameter of the ORAM construction, and not part of its interface, it is written between brackets), and random coins r. It makes use of the following subroutines:

- `ReadBucket(`*bucket*`)` retrieves a set of pairs $(a_i, \texttt{data}_i)$ from the tree node *bucket*.
- `RemoveBlock(`*bucket*, a`)` removes the item with address a from the tree node *bucket*.
- `ChooseEvictionPath` outputs a path for eviction, which differs depending on the specific Tree ORAM scheme.

Pseudo-code for the `Access` procedure is given in Algorithm 2, with additional parameters Z (the number of blocks per bucket, specifided by the Tree ORAM scheme), and random coins r. It makes use of the following subroutines:

- `Size`(X) returns the number of items $|X|$ in X.
- `ChooseNextBlock`(*stash*, *bucket*, *path*) pops an item from the stash, to be stored in the bucket, or outputs $\perp$.
- `WriteBucket`(*bucket*,X,Z) writes the items in X to the node *bucket*, padding the node to size Z if needed.

We will discuss in Section 5 how several existing Tree ORAM schemes are captured by the above paradigm.

Algorithm 1 Access algorithm of a Tree-ORAM.

Access$[Z; r](\text{op}, a, \textit{newdata})$:

1: $\textit{leaf} \leftarrow \texttt{pos}[a]$
2: $\texttt{pos}[a] \leftarrow$ uniformly random leaf
3: **for** *bucket* in $\mathcal{P}(\textit{leaf})$ **do**
4: **if** $(a, \texttt{data}) \in \texttt{ReadBucket}(\textit{bucket})$ **then**
5: $\texttt{RemoveBlock}(\textit{bucket}, a)$
6: **if** op = write **then**
7: data = newdata
8: $\textit{stash} \leftarrow \textit{stash} \cup \{(a, \textit{data})\}$
9: $\textit{path} \leftarrow \texttt{ChooseEvictionPath}(\textit{leaf})$
10: Evict$[Z; r](\textit{path})$
11: **return** data

Algorithm 2 Generic eviction algorithm.

Evict$[Z; r](\textit{path})$:

1: Move all blocks in *path* to the stash
2: **for** *bucket* in *path* **do**
3: $X \leftarrow \varnothing$
4: **while** $\texttt{Size}(X) < Z$ **do**
5: $\textit{block} \leftarrow \texttt{ChooseNextBlock}(\textit{stash}, \textit{bucket}, \textit{path})$
6: **if** $\textit{block} = \perp$ **then**
7: break
8: **else**
9: $X \leftarrow X \cup \{\textit{block}\}$
10: $\texttt{WriteBucket}(\textit{bucket}, \text{X}, \text{Z})$
11: **return**

Correctness of Tree ORAM Since Tree ORAM is a special case of ORAM, the correctness definition remains the same (Definition 5). However, because of the specificities of Tree ORAM, it can be reformulated in a more convenient manner. That is, the only correctness failure that can occur in a Tree ORAM scheme is that the stash overflows. (The reader familiar with Tree ORAM may object that some Tree ORAM schemes do not use a stash; that case will be handled in Section 5.)

Recall that the stash is always padded to size RB, *i.e.* it can store up to R items. Hence, correctness amounts to the following statement: at the outcome of any sequence of legal accesses $(\mathtt{Setup}, \mathtt{Access}_1, \ldots, \mathtt{Access}_k)$, it holds that

$$\Pr[\mathtt{Size}(\mathit{stash}) > R] = \mathrm{negl}(\lambda).$$

3.3 The ∞-ORAM Model

Consider a Tree ORAM instantiation $ORAM^Z \leftarrow \mathtt{Setup}[Z](N, B, \mathsf{D})$, with bucket capacity Z. If $\mathbf{s}$ is a sequence of accesses, we call $st(ORAM^Z[\mathbf{s}])$ the stash usage, that is, the number of items in the stash at the outcome of the accesses.

In Path ORAM and many Tree ORAM schemes derived from it, the proof of correctness follows similar steps:

- Consider an *infinite* ORAM structure $ORAM^\infty$, which is the same protocol, except buckets have infinite capacity (that is, $Z = \infty$).
- Define a post-processing algorithm G_Z that moves items in the tree produced by running $ORAM^\infty$ (arranging in particular that each tree node contains at most Z items). Denote the stash usage of the post-processed ∞-ORAM by $st^Z(ORAM^\infty[\mathbf{s}])$.
- Prove that $st(ORAM^Z[\mathbf{s}]) = st^Z(ORAM^\infty[\mathbf{s}])$ when using the same random coins on both sides.
- Prove that $\Pr[st^Z(ORAM^\infty[\mathbf{s}]) > R] = \mathrm{negl}(N)$.

The last two points imply that $\Pr[st(ORAM^Z[\mathbf{s}]) > R] = \mathrm{negl}(N)$, *i.e.* the original ORAM scheme is correct. We say that such a protocol admits a *proof via infinite ORAM*.

4 Generic Construction of wORAM from Tree ORAM

Our goal is to give a generic way to transform an existing standard Tree ORAM design into one that supports blocks of variable size. The transformation presented in this section is perhaps the most natural transformation one could imagine for that purpose. The main challenge is not the transformation itself, but rather proving that it preserves correctness.

Still, there is one aspect of the transformation whose necessity may not be immediately apparent. Namely, if the original ORAM has bucket capacity Z, then its weighted version will have bucket capacity $Z+1$. This change is not necessary for the main theorem of this section to hold. However, it will be necessary when applying the main theorem to specific ORAMs, notably Path ORAM: see the discussion at the end of Section 5.2. In practice, experiments in Section 4.4 suggest that this increase in bucket capacity can be heuristically dispensed with.

4.1 The `Weighted` Transform

We define a general transform `Weighted` that takes as input a standard Tree ORAM scheme $\mathsf{ORAM}^Z = (\mathtt{Setup}, \mathtt{Access})$ following the framework of Section 3.2, and outputs a weighted ORAM scheme $\mathtt{Weighted}(\mathsf{ORAM}^Z) = \mathsf{ORAM}^{*Z} = (\mathtt{Setup}^*, \mathtt{Access}^*)$.

Let us first consider the setup. We say that the starting scheme ORAM^Z has a *regular* setup if its setup procedure is equivalent to creating an empty tree with all items in the stash, then doing

repeated evictions towards every leaf in the tree from left to right. Here, by "equivalent" we mean that the output of this process and the output of the normal setup process are identically distributed. In our main theorem, we will require that the starting Tree ORAM ORAM^Z has a regular setup. Although that notion of regularity is non-standard, it has the benefit that the behavior of the setup process can be deduced from that of the eviction process. For our purpose, this means it will be enough to explain how to transform the eviction process to handle blocks of variable size.

ORAM^{*Z} is defined in the following way, making only minimal modifications to ORAM^Z to handle items of variable size.

- $\texttt{Setup}^*(N, B, \mathsf{D})$ initializes a tree with N leaves, whose nodes can each hold data of size $(Z+1)B$, and a stash of the same size RB as the standard instance ORAM^Z. It initializes a position map where each address $\mathtt{a}$ in D is mapped to a uniformly random leaf. Finally, it performs a *regular* setup: that is, all items in D are placed in the stash, and the $\texttt{Evict}^*$ procedure is called on the path from the root to each leaf, from left to right.
- $\texttt{Access}^*$ is identical to $\texttt{Access}$, except that it calls the modified subroutine $\texttt{Evict}^*$.
- $\texttt{Evict}^*$ is identical to $\texttt{Evict}$, except that it calls the modified subroutines $\texttt{Size}^*$ and $\texttt{WriteBucket}^*$.
- $\texttt{Size}^*(X)$ returns the sum of the sizes of all items in X divided by B, instead of the number of items in X.
- $\texttt{WriteBucket}^*(\mathit{bucket}, X, Z)$ still writes the items in X to node bucket, the only difference is that it pads the bucket to size $Z+1$ instead of Z.

4.2 Suitable Tree ORAM Schemes

For the $\texttt{Weighted}$ transform to preserve correctness, the original standard ORAM scheme must satisfy certain conditions. This section serves to define those conditions.

Given a sequence of accesses $\mathbf{s}$, some fixed random coins r used during those accesses, and a subset S of nodes in an ∞-ORAM scheme ORAM^*, define the *usage* of S, written $u^S(\mathsf{ORAM}^{*\infty}[\mathbf{s}; r])$, to be the total number of items assigned to the nodes in S. For a wORAM scheme, the usage of S is defined to be the total size of the items assigned to nodes in S, divided by the block size B.

As discussed in Section 3.2, a correctness failure for a Tree ORAM scheme ORAM occurs if and only if, at the outcome of a series of accesses $\mathbf{s}$ with random coins r, the stash receives strictly more than R elements. Using the notation from Section 3.3, this translates to $st(ORAM^Z[\mathbf{s}; r]) > R$. We say that a subset S of nodes *witnesses* the failure if, in the corresponding ∞-ORAM scheme ORAM^* when performing the same sequence of accesses using the same random coins (*viz.* the choices of fresh uniformly random leaves for the position of each accessed item remain the same), $u^S(\mathsf{ORAM}^{*\infty}[\mathbf{s}; r]) > |S| \cdot Z + R$. Intuitively, since the nodes in S can store at most $|S| \cdot Z$ items, it is clear that more than R items must be reassigned to the stash in the original ORAM, which breaks correctness: that is why we say that S witnesses the failure.

Definition 7 ($\mathbf{F} \Rightarrow \mathbf{W}, \mathbf{W} \Rightarrow \mathbf{F}$). *We say that* ORAM *satisfies the $F \Rightarrow W$ property (read: "failure implies witness") with respect to a set $\mathscr{S}$ of subset of nodes, iff for all access sequences $\mathbf{s}$ and all choices of random coins r, $st(ORAM^Z[\mathbf{s}; r]) > R$ implies $\exists S \in \mathscr{S}, u^S(\mathsf{ORAM}^{*\infty}[\mathbf{s}; r]) > |S| \cdot Z + R$. We say that* ORAM *satisfies the $W \Rightarrow F$ (read: "witness implies failure") property if the converse is true.*

Moreover, we say that ORAM *satisfies the $F \Rightarrow W$ (resp. $W \Rightarrow F$) property* via union bound *if the scheme also satisfies that $\sum_{S \in \mathscr{S}} \Pr[u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}]) > |S| \cdot Z + R] = \mathrm{negl}(\lambda)$. Informally, this means the statement "the probability that a failure witness exists is negligible" can be proved via a union bound over all possible witnesses $S \in \mathscr{S}$.*

The definitions remain the same for a wORAM scheme. In particular, for a wORAM scheme ORAM^*, a subset S witnesses a failure if $u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}]) > |S| \cdot Z + R$ (and not $|S| \cdot (Z+1) + R$, even though our construction of wORAM uses buckets of size $(Z+1)B$).

Definition 8 (Suitable Tree ORAM). *We say that a Tree* ORAM *scheme is* suitable *if it satisfies the following conditions.*

1. *It admits a proof via infinite ORAM. That is, for all access sequence* $\boldsymbol{s}$ *and random coins* r, $st(ORAM^Z[\mathbf{s}; r]) > R$ *iff* $st^Z(ORAM^\infty[\mathbf{s}; r]) > R$.
2. ORAM *satisfies the* $W \Rightarrow F$ *property with respect to some set* $\mathscr{S}$*, via union bound.*
3. `Weighted`(ORAM) *satisfies the* $F \Rightarrow W$ *property with respect to the same* $\mathscr{S}$.
4. ORAM *allows* free evictions*. That is, if the client is allowed to trigger evictions on uniformly random leaves at will during a sequence of accesses, correctness still holds.*

Requiring all those properties may seem demanding, but they naturally hold for several existing Tree ORAM schemes, including Path ORAM and Simple ORAM. This will be shown in more detail in Section 5. Intuitively, this is because many schemes admit a proof via infinite ORAM, either explicitly (in the case of Path ORAM), or trivially (in the case of Simple ORAM, where the ORAM and its infinite variant are identical up to correctness failures). Similarly, the F $\Rightarrow$ W property is either already known, or trivial; and the *free eviction* property is immediate. The only property that requires some care is to show that `Weighted`(ORAM) satisfies the F $\Rightarrow$ W property. However, it is much more tractable than trying to analyze the correctness of a wORAM scheme directly.

4.3 Main Result

Theorem 1 (Main Theorem). *Let* ORAM *be any suitable Tree ORAM scheme. If* ORAM *is a correct ORAM scheme, then* `Weighted`(ORAM) *is a correct wORAM scheme.*

Before diving into the proof proper, we sketch the underlying approach. Because of the F $\Rightarrow$ W and W $\Rightarrow$ F properties required by the suitability assumption, showing the the wORAM scheme is correct essentially amounts to showing that no set $S \in \mathscr{S}$ witnesses a failure. We wish to analyze the function that maps the sizes of items to the usage of S (*i.e.* the sum of sizes of all items in S). Ultimately, we want to show that the probability that the usage of S exceeds $|S| \cdot Z + R$ is negligible, regardless of item sizes.

The proof strategy is to upper-bound the previous probability by a Schur-convex function (Section 2.1), and show that this function is negligible. The idea behind this strategy is that if a function of item sizes is Schur-convex, then in order to upper bound the function for *all* possible vectors of item sizes, it is enough to upper-bound it for a set of maximal vectors for the majorization order. Luckily, due to the requirement that item are of size at most B, and that the sum of items sizes is at most NB, a single weight vector majorizes all others, namely the vector $(B, \ldots, B, 0, \ldots, 0)$. Hence, it is enough to upper-bound the function for that specific vector. This is quite convenient, because that weight vector essentially amounts to having all items be of the same size, which reduces to the correctness of the original (unweighted) ORAM instance.

To instantiate the proof idea outlined above, the core argument is to show that there exists a suitable Schur-convex function. This is done via a first-moment argument (Lemma 2), which allows us to work with expectancies instead of probabilities. Expectancies are much better behaved with respect to convexity (due to the linearity of expectation). Eventually, we massage the upper bound into a suitable Schur-convex function (in the proof, this is the map $\mathbf{w} \mapsto \mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$), and show it is convex essentially by showing that it is structured as a composition of convex maps. Using Lemma 1, we deduce that it is Schur-convex.

Proof. We start with a small self-contained lemma.

Lemma 2. *Let* X *be an integral random variable defined over* $[0, t] \subset \mathbb{N}$*, with* $t \in \mathsf{Poly}(\lambda)$*. Then* $\Pr[X > R] = \mathrm{negl}(\lambda)$ *if and only if* $\mathbb{E}(\max(0, X - R)) = \mathrm{negl}(\lambda)$.

Proof. Recall that the expectation of a positive integral variable Y can be written as:

$$\mathbb{E}(Y) = \sum_{i \geq 0} \Pr[X > i].$$

As a corollary, for any integral variable Y satisfying $0 \leq Y \leq t$:

$$\Pr[Y > 0] \leq \mathbb{E}(Y) \leq t \Pr[Y > 0]. \tag{1}$$

Observe that the event $X > R$ is equivalent to $\max(0, X - R) > 0$. Using that observation, and applying (1) to the variable $Y = \max(0, X - R)$, we get:

$$\Pr[X > R] \leq \mathbb{E}(\max(0, X - R)) \leq t \Pr[X > R].$$

Since $t \in \mathsf{Poly}(\lambda)$, we are done. ■

Let ORAM be a suitable and correct Tree ORAM scheme. Let $\mathsf{ORAM}^* \leftarrow \mathtt{Weighted}(\mathsf{ORAM})$. Let $\mathbf{s}$ be a legal sequence of accesses for ORAM^*. We need to show that $\Pr[st(\mathsf{ORAM}^*[\mathbf{s}]) > R] = \mathrm{negl}(\lambda)$.

Since ORAM satisfies the F $\Rightarrow$ W property with respect to some set $\mathscr{S}$, it suffices to show that the probability that there exists $S \in \mathscr{S}$ witnessing the failure is negligible, *i.e.* $\Pr[\exists S \in \mathscr{S}, u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}]) > |S| \cdot Z + R]$ is negligible.

Let us fix $S \in \mathscr{S}$. We want to show that $\Pr[u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}]) > |S| \cdot Z + R]$ is negligible. (This is not enough to imply that the probability that there *exists* such an S is negligible, since $\mathscr{S}$ may have superpolynomial cardinality; we will come back to this point later.) A crucial observation is that in $\mathsf{ORAM}^{*\infty}_L$, the sizes of data items plays no role. In particular, given an access sequence $\mathbf{s}$ and associated random coins r, the location of each item in the tree is entirely determined *independently of the size of the data items.*

Given an access sequence $\mathbf{s}$ with m items in total, and a *size allocation vector* $\mathbf{w} = (w_i)_{i \leq m} \in [0,1]^m$, define $\mathbf{s}(\mathbf{w})$ to be the access sequence $\mathbf{s}$, modified such that at the outcome of the sequence the i-th item has size w_i. Let $\mathit{\Pi}$ be the set of permutation matrices of size m. Let $X_{\mathbf{s},L,S}(\mathbf{w}) = \max_{\mathbf{P} \in \mathit{\Pi}}(\max(0, u^S(ORAM^{*\infty}_L[\mathbf{s}(P\mathbf{w})]) - (|S| \cdot Z + R)))$. By Lemma 2, $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ is an upper bound on $\Pr[u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}]) > |S| \cdot Z + R]$, so it is enough to show that $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ is negligible. This will follow from the next lemma.

Lemma 3. *Let* $\mathbf{s}$ *be a legal sequence of accesses, and let* $S \in \mathscr{S}$*. Then the map* $f : \mathbf{w} \mapsto \mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ *is Schur-convex.*

Proof. First, we show that $X_{\mathbf{s},L,S}$ is convex when the random coins used in the ORAM construction are fixed. Until further notice, we assume that all random coins are fixed. Only $\mathbf{w}$ varies. Let $\lambda \in [0,1]$, and let $\mathbf{v}$, $\mathbf{w}$ be two size allocation vectors. Observe that the map $g : \mathbf{w} \mapsto u^S(ORAM^{*\infty}_L[\mathbf{s}(P\mathbf{w})]$ is linear. This is because, as already noted, whether an item is stored in a node from S or not is independent of the weight of the items. As a consequence, $g(\mathbf{w})$ is equal to the sum of the weights of items stored in S, *i.e.* it is a fixed linear combination of w_i's (with binary coefficients). Since g is linear, it is trivially convex.

Also observe that for any constant C, the map $h : x \mapsto \max(0, x - C)$ is increasing and convex. Since the composition of an increasing convex function with a convex function is convex, we deduce that the map $h \circ g$ is convex. Since $X_{\mathbf{s},L,S}(\mathbf{w}) = \max_{\mathbf{P} \in \mathit{\Pi}} h \circ g(\mathbf{P}\mathbf{w})$, it is a maximum of convex maps, so it is also convex.

On the other hand, $X_{\mathbf{s},L,S}(\mathbf{w})$ is symmetric by construction, since it takes the maximum over all permutations of $\mathbf{w}$. By Lemma 1, since $X_{\mathbf{s},L,S}(\mathbf{w})$ is both symmetric and convex, it is Schur-convex.

It remains to show that Schur convexity still holds when considering the expectation of $X_{\mathbf{s},L,S}(\mathbf{w})$. (From now on, we no longer assume that random coins are fixed.) However, it is straightforward to

show that if a probabilistic map is Schur-convex for every fixed choice of random coins (sometimes called stochastical Schur-convexity), then its expectation is also Schur-convex [MOA79]. We conclude that $\mathbf{w} \mapsto \mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ is Schur-convex. ■

Corollary 1. *Let* $\mathbf{s}$ *be a legal sequence of accesses with weight vector* $\mathbf{w}$*, and let* $S \in \mathscr{S}$*. Then* $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ *is negligible.*

Proof. For an access sequence to be legal, its weight vector $\mathbf{w}$ must satisfy that $w_i \leq B$ for all i, and $\sum w_i \leq NB$. Observe that all such vectors are majorized by the vector $\mathbf{v} = (B, \ldots, B, 0, \ldots, 0)$ containing N initial B's. Since $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$ is Schur-convex, it follows that $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})] \leq \mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{v})]$: in order to upper bound $\mathbb{E}[X_{\mathbf{s},L,S}(\mathbf{w})]$, it suffices to focus on the weight vector $\mathbf{v}$. (This is the point of using a Schur-convexity argument.)

But in the case of the vector $\mathbf{v}$, all items are of the same size B, or of size 0.[1] In that case, ORAM^* behaves exactly like ORAM, except that accesses to items of size 0 translate to evictions without any prior item access. In particular, the usage of S is the same for ORAM^* and ORAM. Since we assume that ORAM has the free eviction property, it remains correct when allowing eviction queries by the client. Since it is also assumed to be correct and to satisfy W $\Rightarrow$ F, it follows that the usage of S cannot exceed $|S| \cdot Z + R$ except with negligible probability, hence the same holds for ORAM^*, and we are done. ■

So far, we have shown that the probability that any given S witnesses a failure in ORAM^* is negligible. To conclude the proof, it remains to show that the probability that there *exists* an $S \in \mathscr{S}$ witnessing a failure is negligible. This does not follow immediately from the previous statement, because $|\mathscr{S}|$ may be superpolynomial. However, looking at the proof of Lemma 2, we see that when switching from expectation to probability and back, we only lose a factor t. In our case, the stash size is a random variable bounded by NB, so we have that for every S,

$$\Pr[u^S(\mathsf{ORAM}^{*\infty}_L[\mathbf{s}(\mathbf{v})]) > |S| \cdot Z + R] \leq NB \Pr[u^S(\mathsf{ORAM}^{\infty}_L[\mathbf{s}]) > |S| \cdot Z + R].$$

Since ORAM is assumed to satisfy W $\Rightarrow$ F *via union bound*, and $NB \in \mathrm{poly}(\lambda)$, we know that the sum of the latter quantity over all $S \in \mathscr{S}$ is negligible, hence ORAM^* inherits the same union bound property. It follows that the probability that there exists a failure witness S for ORAM^* is negligible. Since ORAM^* satisfies the F $\Rightarrow$ W property, we conclude that ORAM^* is correct. □

4.4 Experimental Results

To test empirically the correctness of our weighted ORAM, we implemented a Path ORAM structure and performed simulated accesses. We did two experiments: one with N object of the same size (which simulates the standard case) and one with objects of variable sizes (the sizes are uniformly random, but sum to N). Our results are presented as graphs in Figure 1.

We took inspiration from the experiment in Section 7 of [SvS+13]. The experiment went as follows:

- We generated ORAM structures for N objects, with $N = 2^L$ and $L \in \{10, 11, \ldots, 22\}$. The bucket size is $Z \in \{3, 4\}$
- We chose the maximum block size to be $B = 512$.

[1] The reader may observe that items of size 0 are not technically legal per the earlier definition of wORAM, which asks that items are of size at least 1; however, `Weighted`(ORAM) remains well-defined even for items of size 0, so nothing stops us from using them within the proof —the reason we forbade items of size 0 is that they would allow for an unbounded number of items, which would require a position map of unbounded size, but this is irrelevant for the current line of reasoning.

– For the standard ORAM simulation, all blocks were of size B. For the variable ORAM simulation, blocks were taken uniformly at random in $[B]$, with the total sum of the sizes being $N \cdot B$. The number m of blocks generated is roughly $2 \cdot N$.
– We start with the Path ORAM loaded randomly with the objects at its leaves, and perform between $10 \cdot m$ and $50 \cdot m$ accesses in the order $\{1, 2, \ldots, m, 1, 2, \ldots\}$.

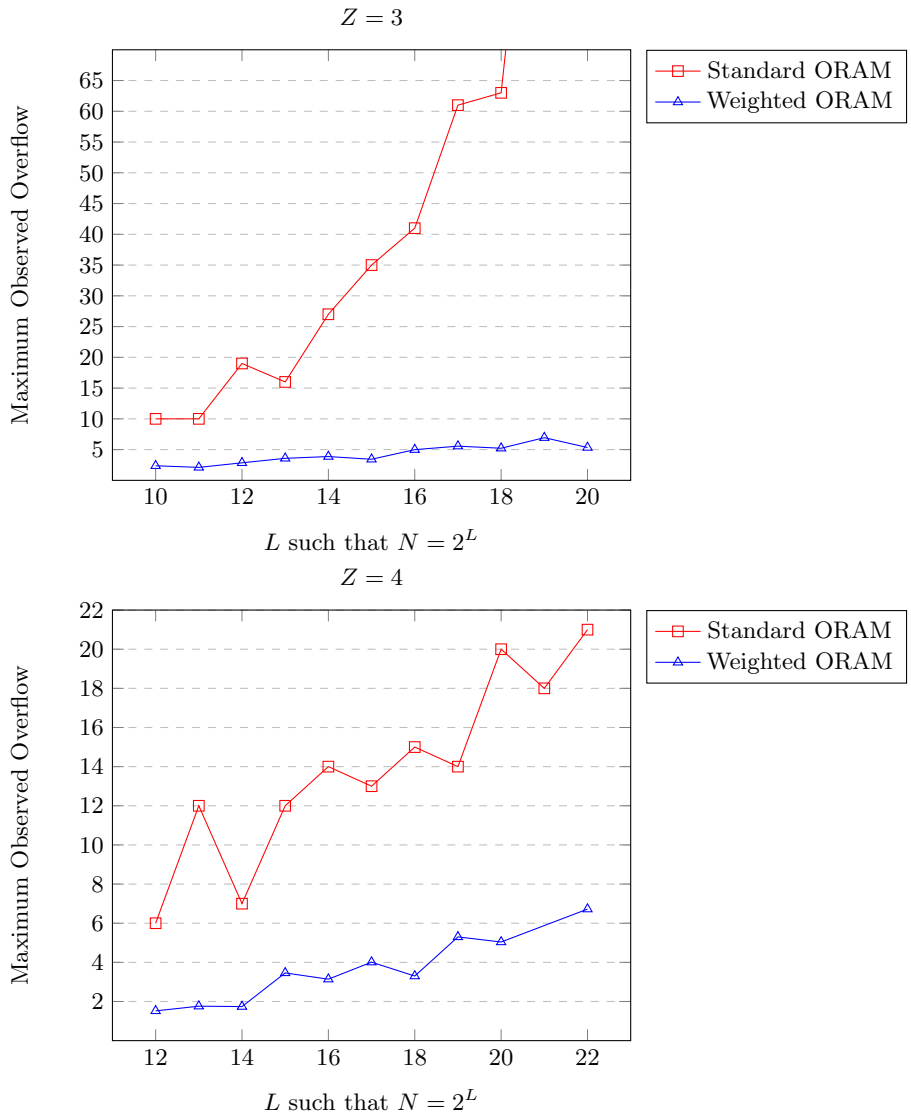

Fig. 1. Experimental results for $Z \in \{3, 4\}$

Figure 1 suggests that objects of variable size are even less prone to stash overflows than the standard case. Regarding bucket size, we make an observation similar to that of [SvS+13]: even though the correctness of the ORAM was proved for $Z + 1 = 6$, the construction appears resilient enough to work

correctly even when $Z + 1 = 4$. In [SvS+13], the empirical results suggest that Path ORAM can be used with Z as low as 4.

5 Application to Existing Tree ORAMs

In this section we present several concrete constructions: a weighted Simple ORAM, based on Chung and Pass's Simple ORAM [CP13], and a weighted Path ORAM, based on the seminal work by Shi *et al.* [SvS+13]. The construction for Path ORAM can be easily adapted to build a weighted *Random-Index ORAM* from the one presented in [HK22], as the block-holding structure is virtually the same. We also sketch the application to Circuit ORAM [WCS14] and OPRAM [CCS17]. By Theorem 1, in each case, it suffices to show that the scheme is suitable. The weighted variant is then obtained by applying the `Weighted` transformation.

5.1 Weighted Simple ORAM [CP13]

In `SimpleOram`, each bucket has a capacity of $Z = \mathcal{O}(\log N)$, and the ORAM overflows if and only if there is a bucket with more than Z items: there is no stash. From the perspective of the Tree ORAM framework from Section 3.2, a stash does exist, however, it is required that it is empty at the outcome of any (legal) sequence of accesses. That is, we set the stash bound R to 0.

In `SimpleOram`:

- The `ChooseEvictionPath`(*leaf*) method is implemented by choosing a path uniformly at random (the *leaf* argument is ignored).
- The `ChooseNextBlock`(*stash*, *bucket*, *path*) method is implemented by returning the first item among items whose position is such that its meet with the current path is exactly *bucket*. (In other words, all items are stored as low as possible along the eviction path, while preserving the invariant that they are stored above their associated leaf.) The correctness of `SimpleOram` relies on the fact that all such items will fit in the current bucket; items are never pushed somewhere else in case a bucket is full.

We want to show that `SimpleOram` is suitable. Define $\mathscr{S}$ to be the set containing the singleton $\{bucket\}$ for each tree node *bucket*. The fact that the correctness of `SimpleOram` is equivalent to the fact that no element of $\mathscr{S}$ witnesses a failure is immediate, since the correctness of `SimpleOram` requires precisely that no node overflows. Hence, `SimpleOram` satisfies W $\Rightarrow$ F (and F $\Rightarrow$ W) with respect to $\mathscr{S}$. The fact that `Weighted`(`SimpleOram`) satisfies F $\Rightarrow$ W is immediate for the same reason. `SimpleOram` also satisfies the union bound requirement, because its analysis in [CP13] relies on just such a union bound. The fact that it supports free evictions also follows directly the analysis in [CP13] (additional evictions translate to more success chances in the dart game argument at the center of the analysis). We conclude that `SimpleOram` is suitable.

Theorem 2. `Weighted`(`SimpleOram`) *is a correct wORAM scheme.*

5.2 Weighted Path ORAM [SvS+13]

Let $ORAM^Z \leftarrow$ `PathOram.Setup`(N, Z) be an instance of Path ORAM. The bucket capacity Z for Path ORAM is a small constant; the scheme is proven correct for $Z = 5$, we shall use this value. The stash capacity is $R = \mathcal{O}(\log N)$.

In `PathOram`:

- `ChooseEvictionPath`(*leaf*) simply returns $\mathcal{P}(leaf)$.

– `ChooseNextBlock`(*stash*, *bucket*, *path*) returns an item from the buffer such that its associated leaf is below *bucket*, and the meet between the position of the item and *path* is lowest among such items.

Theorem 3. `Weighted`(`PathOram`) *is a correct wORAM scheme.*

Proof. Define $\mathscr{S}$ to be the set of all subtrees of the ORAM tree, where a *subtree* is defined to be a subset of nodes closed for the *parent* relation (*i.e.* if the set contains a node, it also contains its parent). The analysis of [SvS+13] proves that `PathOram` satisfies both F $\Rightarrow$ W and W $\Rightarrow$ F with respect to $\mathscr{S}$, via a union bound. The fact that `PathOram` supports free evictions also follows from the analysis. In the remainder, we focus on showing that `PathOram`$^* \leftarrow$ `Weighted`(`PathOram`) satisfies F $\Rightarrow$ W.

We follow a similar approach to the initial part of the analysis in [SvS+13]. Let us define a post-processing algorithm G_Z, which is applied to $ORAM^{*\infty}$ after a sequence of accesses. Like in the original analysis of Path ORAM, the G_Z algorithm is a "thought experiment": that is, the algorithm is used within the proof, but never actually run by any party in the ORAM protocol. As a consequence, we can allow G_Z to perform operations that are not normally allowed within the wORAM framework. In particular, we let G_Z "split" any object of size w in two objects of sizes w_1 and w_2 such that $w_1 + w_2 = w$, storing the two chunks at distinct locations. G_Z repeats the following process, as long as there are overfull buckets (*i.e.* whose size is strictly more than Z—to avoid cluttering the notation, all sizes are implicitly divided by the block size B):

1. Select a bucket b that has load strictly more than Z. Remove blocks from the bucket (splitting if needed) so that it ends up having load exactly Z. Removed blocks are stored in a temporary buffer.
2. Find the nearest ancestor of b that has load strictly less than Z. Store items from the buffer into this ancestor, until its load is exactly Z (or the buffer is empty), splitting if needed.
3. As long as the buffer is not empty, repeat the previous step, working our way up the path $\mathcal{P}(b)$, until reaching the stash. At that point, any remaining items in the buffer is stored in the stash.

First, let us prove that the stash usage (i.e. the cumulated size of all items in the stash) of the post-processed ∞-ORAM is greater than the stash usage of $ORAM^{*Z}$:

$$st^Z(ORAM^{*\infty}[\mathbf{s}]) \geq st(ORAM^{*Z}[\mathbf{s}]). \tag{2}$$

First, notice that the order in which overfull blocks are processed by G_Z has no bearing on the final occupancy of the nodes and stash. In particular, $st^Z(ORAM^{*\infty}[\mathbf{s}])$ is well-defined, even though we put no restriction on the order in which G_Z processes overfull buckets. To show this, we can generalize the argument from [SvS+13]: assume that G_Z processes blocks from the bucket β_1 at level l_1 on path p_1, then blocks from the bucket β_2 at level l_2 on path p_2. We want to show that the loads in the buckets in $p_1 \cup p_2$ do not change if we let G_Z process β_2 before β_1. Without loss of generality, we can assume that those buckets are siblings (i.e. $l_1 = l_2 = l$), since only $p_1 \cap p_2$ will be affected by a change in the order. Assume that the post-processed blocks from β_1 (resp. β_2) are of total size W_1 (resp. W_2). G_Z first distributes a "mass" of size W_1 in the buckets from level $l-1$ to -1 in $p_1 \cap p_2$, and then a mass of size W_2 in those same buckets. Before the distribution, let us call V_i the available space in the bucket at level $i \in \{-1, 0, \ldots, l-1\}$ on path $p_1 \cap p_2$. When distributing a mass W, G_Z performs Algorithm 3 (we assume that $V_{-1} = \infty$): The $\{V_i\}$ are the same after a successive application of Algorithm 3 on W_1 then W_2 or after its application on W_2 then W_1.

Remark. We wish to draw the reader's attention to one point: in what precedes, we consider for simplicity that blocks are taken in bulk from the buckets, whereas in what follows it is more convenient to assume that G_Z processes them individually. It doesn't make a difference for the same reason that the order doesn't matter.

Algorithm 3 Distribution of mass of blocks

Distribution(W):

```
1: i ← l
2: while W > 0 do
3:     i ← i − 1
4:     if V_i ≥ W then
5:         V_i ← V_i − W
6:         W ← 0
7:     else
8:         V_i ← 0
9:         W ← W − V_i
```

We can finally prove (2). Informally, we can see that during the accesses, $ORAM^{*Z}$ stores blocks in buckets and in the stash in a more lenient way than G_Z, since it allows blocks to "stick out" of the buckets. More precisely, after the accesses of $\mathbf{s}$ in $ORAM^{*\infty}[\mathbf{s}]$, there exists a way to move blocks from the buckets they reside in to their final destination from $ORAM^{*Z}[\mathbf{s}]$ (in another bucket or the stash). Since the order in which we post-process blocks from the buckets does not matter, we can assume that this particular order is accessed by G_Z. If that is the case, after the processing of each block, G_Z puts that block in the bucket where it belongs according to $ORAM^{*Z}[\mathbf{s}]$. However, should a part of the block (or its entirety) stick out of the bucket (i.e. causes the load to become $\geq Z$), this part will be moved to a higher block or the stash. Thus the processing of each block by G_Z causes the stash size to either stay the same or to increase. Thus at the end of the processing, $st^Z(ORAM^{*\infty}[\mathbf{s}]) \geq st(ORAM^{*Z}[\mathbf{s}])$.

We are now ready to show that `Weighted(PathOram)` satisfies F $\Rightarrow$ W. By (2), if correctness fails for `Weighted(PathOram)`, then it also fails (with the same random coins) for the post-processed ∞-ORAM variant. That is, $st^Z(ORAM^{*\infty}[\mathbf{s}]) > R$. Let us define T to be the maximal subtree that only contains buckets of size at least Z after the post-processing. If a bucket b is not in T, it has an ancestor b' that has usage strictly less than Z, so the blocks of b cannot go to the stash. Thus all blocks in the stash came from buckets in T, and thus $u^T(ORAM^{*\infty}[\mathbf{s}]) > n(T) \cdot Z + R$. This shows that T is a failure witness. □

Discussion. In the proof above, a "splitting" variant of wORAM was introduced. In that splitting variant, items can be split into pieces of arbitrary sizes, and each piece can be stored at a different tree node (still maintaining the invariant that all pieces belonging to the same item are stored somewhere along the path from the root to the leaf associated with the item). Splitting is a convenient intermediate abstraction, because of its "continuous" behavior, which avoids rounding issues.

It is worth noting that splitting is not just a proof abstraction, but could also be used in actual tree-based wORAM algorithms. In fact, we could have defined our weighted transformation to allow splitting. In that case, the proof above shows that the weighted variant of Path ORAM would have the same bucket capacity Z as the original Path ORAM.

By contrast, our actual construction, which does not use splitting, needs to increase the bucket capacity to $Z+1$ to avoid "rounding" issues. Intuitively, without splitting, if a bucket of capacity ZB words contains data of total size $(Z-1)B+1$, and the client wants to insert a block of maximum size B, it cannot fit in that bucket. This means each bucket can have up to $B-1$ words of "wasted" space. Note that this cannot happen in standard ORAM, where all items have size B: it is an issue specific to weighted ORAM (and weighted allocation mechanisms in general). Splitting solves the issue without the need to increase bucket capacity: the ability to split avoids any wasted space.

Our construction solves the issue differently, without splitting, by increasing bucket capacity to $(Z+1)B$ words (or $Z+1$ full blocks). This guarantees that if a bucket cannot receive a new block,

then it already contains at least ZB words. In short, the reason we increase bucket capacity to $Z+1$ in our constructions is to resolve rounding issues inherent to fitting weighted items into buckets of fixed capacity.

As noted, we could have chosen to define our `Weighted` transformation to allow splitting. We avoided that option, because it complicates implementation in practice, and may require significantly more metadata (each piece needs an identifier tag). Nevertheless, it is a valid option, and may be preferable if a slightly lower Z is desirable.

5.3 Weighted Oblivious Parallel RAM [CCS17]

Boyle, Chung, and Pass's protocol [CCS17] is based on Simple ORAM. Their framework present ORAM protocols for parallel algorithms, i.e. with multiple processors (clients). The `Weighted` function is not impacted by the fact that there are several clients: The modifications to the subroutines still capture this case, and the correctness analysis holds. The only new component is the broadcast routine, where one of the CPUs broadcasts information about a certain block to the others CPUs. These messages are bounded by the size of the block. That could lead to a leakage, however because of the need to index the blocks, which will take a size of at least $\log m \geq \log N$. Thus, we can lower bound the size of the messages by $\mathcal{O}(\log N)$: their size will not leak information on the block. This yields a correct weighted OPRAM.

5.4 Weighted Circuit ORAM [WCS14]

Circuit ORAM is a variant of Path ORAM, where the client only needs local space to hold one block. To achieve this, the eviction algorithm is slightly different and the stash is stored by the server (as the parent of the root node) instead of locally.

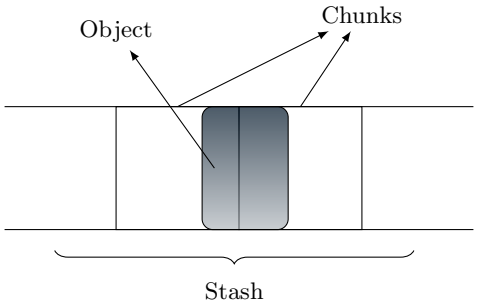

Fig. 2. A block of size $< B$ in 2 chunks

The correctness analysis of this scheme is based on the same principles as the one for Path ORAM. The function `Weighted` yields a correct ORAM here too. To prove this, we only need to show how we can adapt for the fact that the stash is stored on top of the tree. Figuring out which way to do this is not obvious since, because of the varying block size, the client cannot simply stream the content of the stash block by block: it would leak block size information. We propose a simple fix, which we also use when dealing with Trivial ORAM:

– We allow the client to store an additional space of size B (the maximal size of a block). This gives the client a local space of at least $2 \cdot B$.

– Whenever the client needs to access the stash, the client streams the content of the stash *chunk by chunk*, where each chunk is of size B. That way, since a block must always reside inside at most 2 chunks (see Figure 2), the client will read every object at the end of the stash stream.
– When the client wishes to write back a block, it is done locally among two chunks.

This way, the scheme stays correct and secure even with objects of variable size. The application of the suitability criterion shows that Circuit ORAM is compatible with blocks of variable size.

6 Searchable Encryption from Weighted ORAM

With Searchable Symmetric Encryption (SSE), a client can delegate the storage of a database to a honest-but-curious server. The client is then able to perform searches on the database by issuing `Search` queries to the server. In the case of *dynamic* SSE, the client may also update the database by issuing `Update` queries to the server. The security goal is that the information leaked to the server during these operations should be limited, in a sense that will be defined soon.

Here, we focus on the case of single-keyword search. In that setting, the client's database consists of a collection of documents, and `Search` queries ask to retrieve all documents that contain a given keyword. In modern SSE schemes, this functionality is realized efficiently by building a reverse index: For each keyword w, a list of the identifiers of documents matching the keyword, written $\mathsf{DB}(w)$, is maintained on the server side in some encrypted form. *Response-revealing* SSE allows the server to learn the list of document identifiers, while response-hiding SSE does not: they are sent back to the client in encrypted form. Once having retrieved the desired document identifiers, the client may perform some additional computation, such as intersecting the results with other queries, or may fetch the documents on the same or a different server. In the case of response-revealing SSE, if the same server stores the reverse index and the documents, the server can immediately send back the documents without the need of an additional roundtrip, at the cost of possibly leaking additional information to the server. We note that the documents could be stored in an ORAM to avoid additional leakage, and that a weighted ORAM would reduce the performance cost of this approach. However, as in most SSE literature, we focus on the reverse index.

For efficiency reasons, SSE typically does not seek to have minimum leakage, but rather to strike a compromise between security and performance by allowing a controlled amount of leakage. In the security model, the leakage allowed by the scheme is expressed by a *leakage function* $\mathcal{L} = \{\mathcal{L}_{\texttt{Setup}}, \mathcal{L}_{\texttt{Search}}, \mathcal{L}_{\texttt{Update}}\}$. The security model asks that during a `Setup` operation (resp. `Search`, `Update`) with input x, the information leaked to the server is included in $\mathcal{L}_{\texttt{Setup}}(x)$ (resp. $\mathcal{L}_{\texttt{Search}}(x)$, $\mathcal{L}_{\texttt{Update}}(x)$). More formally, it is required that there must exist a simulator $\mathcal{S}$ such that the view of the server during $\texttt{Setup}(x)$ (resp. $\texttt{Search}(x)$, $\texttt{Update}(x)$) should be indistinguishable from $\mathcal{S}(\mathcal{L}_{\texttt{Setup}}(x))$ (resp. $\mathcal{S}(\mathcal{L}_{\texttt{Search}}(x))$, $\mathcal{S}(\mathcal{L}_{\texttt{Update}}(x))$).

Response-revealing SSE schemes leak the *access pattern*: That is, the server learns the identifiers of all documents matched by a query. In some use cases, access pattern leakage can be quite damaging, and allow the server to infer a sizable amount of information about the database [CGPR15, GLMP19]. Even in the case of response-hiding SSE, the server can typically learn the *query pattern*: that is, the server can learn whenever the client repeats the same query. In many cases, the server can learn the *volume* of the answer: that is, the number of documents matched by the query.

In some use cases, these different types of information leakage can be quite damaging, as shown by so-called *leakage-abuse* attacks [CGPR15, GSB+17]. To thwart those attacks, recent works have developed various protections, such as volume-hiding SSE [KM19, PPYY19], and the line of work on *leakage suppression* [KMO18]. The strongest form of protection, considered for instance in [GMP16, MM17, KMO18], involves the use of ORAM, or specialized variants of ORAM. This raises some questions about how to optimize the use of ORAM, in order to preserve the high efficiency goal of

SSE. In particular, as discussed *e.g.* in [GMP16], since reverse indexes contain lists that can greatly vary in size, it is not obvious how to fit them into a (fixed-block size) ORAM. Our main point in this section is that the weighted construction introduced here fits this setting perfectly. Concretely, we propose two SSE constructions based on weighted ORAM: ZeroSSE and BlockSSE. A brief overview is given Figure 3. We note that the main point of TWORAM, not reflected in the table, is to reduce the number of roundtrips in the iterative version of Path ORAM, thanks to a clever use of garbled circuits. However garbled circuits add a considerable overhead in practice.

Scheme	client storage	bandwidth overhead
TWORAM [GMP16]	$\mathcal{O}(1)$	$\mathcal{O}(\lambda \log^2 N)$
ZeroSSE	$\mathcal{O}(W)$	$\mathcal{O}(\log(N/U))$
ZeroSSE′	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 W + \log(N/U))$
BlockSSE	$\mathcal{O}(W)$	$\mathcal{O}(\log(N/B))$
BlockSSE′	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 W + \log(N/B))$

Fig. 3. Overhead of ORAM-based SSE constructions. U is an upper bound on the longest list size, $W \leq N$ is the number of keywords, B is the ORAM block size.

6.1 Preliminaries

We follow the standard definition of SSE. A dynamic SSE scheme Σ consists of four protocols, defined as follows.

- $\Sigma.\mathtt{KeyGen}(1^\lambda)$: Takes as input the security parameter λ. Outputs the master secret key K.
- $\Sigma.\mathtt{Setup}(\mathsf{K}, N, \mathsf{DB})$: Takes as input the client secret key K, an upper bound on the database size N, and a database DB. Outputs an encrypted database EDB.
- $\Sigma.\mathtt{Search}(\mathsf{K}, w, \mathsf{st}; \mathsf{EDB})$: The client receives as input the secret key K, and keyword w. The server receives as input the encrypted database EDB. Outputs updated encrypted database EDB' for the server.
- $\Sigma.\mathtt{Update}(\mathsf{K}, (w, e); \mathsf{EDB})$: The client receives as input the secret key K, and a pair (w, e) of keyword w and document identifier e. The server receives as input the encrypted database EDB. Outputs updated encrypted database EDB' for the server.

The security model expresses that the view of the server can be simulated by an efficient simulator, receiving as input only the output of the leakage function. In more detail, we define two games, SSEReal and SSEIdeal. First, the adversary chooses a database DB. In SSEReal, the encrypted database EDB is generated by $\mathtt{Setup}(\mathsf{K}, N, \mathsf{DB})$, whereas in SSEIdeal, the encrypted database is simulated by a (stateful) simulator $\mathcal{S}$ on input $\mathcal{L}_{\mathtt{Setup}}(\mathsf{DB}, N)$. After receiving EDB, the adversary can issue search and update queries. In SSEReal, queries are answered using the real-world protocol. In SSEIdeal, the `Search` queries (resp. `Update`, `Setup`) on input x are simulated by $\mathcal{S}$ on input $\mathcal{L}_{\mathtt{Search}}(x)$ (resp. $\mathcal{L}_{\mathtt{Update}}(x)$, $\mathcal{L}_{\mathtt{Setup}}(x)$). Finally, the adversary outputs a bit b.

The scheme is said to be $\mathcal{L}$-*secure* (*i.e.* secure with respect to the leakage function $\mathcal{L}$) if for all PPT adversaries, there exists a PPT simulator such that the transcripts in the real and ideal world are computationally indistinguishable.

6.2 ZeroSSE

A line of recent work has aimed to hide volume leakage: that is, to hide the number of identifiers matching a given query [KM19, PPYY19]. Hiding volume leakage seems sensible when using ORAM

technique to hide the query pattern, since volume leakage reveals information about the repetition of queries. This leads to the question of building on ORAM that also hides volume. For that purpose, an upper bound U is assumed on the volume of the longest list (that is, the longest query answer). As discussed in [GMP16], the first approach one may think of is to use an ORAM with block size U; however, this would require padding all lists to U, which would be prohibitive in many use cases, since the longest list may be several orders of magnitude larger than the average list size. In the worst case, the blowup in storage is $\Omega(U)$, even before considering ORAM overheads. Another approach would be to use a smaller block size, at the cost of a larger ORAM overhead.

The idea of ZeroSSE is simply to use the weighted variant of Path-ORAM, $\texttt{Weighted}(\texttt{PathOram})$ with $B = U$ as the block size upper bound. Relative to the previous two approaches, this minimizes both the overhead due to padding, which is nonexistent since no padding is necessary, and the overhead due to ORAM, since we use the largest block size possible. In fact, since our main result is that Path ORAM can handle items of variable size at essentially no overhead, we contend that this is both the most natural and most efficient solution to building SSE with minimum leakage.

We define ZeroSSE in more details as follows. We note that `Setup` takes as input additional parameters U, which is an upper bound on the longest list size, and W, an upper bound on the number of keywords.

- $\mathsf{ZeroSSE}.\texttt{Setup}(\mathsf{K}, N, \mathsf{DB}, U)$: Initializes $\texttt{Weighted}(\texttt{PathOram})$ with block size U and number of leaves $\lceil N/U \rceil$, containing as (variable-size) blocks $\mathsf{DB}(w)$ for each keyword w. The position map, of size $\mathcal{O}(W)$ memory words, is stored on the client side.
- $\mathsf{ZeroSSE}.\texttt{Search}(\mathsf{K}, w; \mathsf{EDB})$: The client queries the ORAM for keyword w to retrieve $\mathsf{DB}(w)$.
- $\mathsf{ZeroSSE}.\texttt{Update}(\mathsf{K}, (w, e); \mathsf{EDB})$: The client queries the ORAM for keyword w to retrieve $\mathsf{DB}(w)$, and simply writes back $\mathsf{DB}(w) \cup \{e\}$. (Recall that our weighted construction allows modifying the size of blocks on the fly.)

ZeroSSE uses the non-iterative variant of Path-ORAM. This is because a client storage of $\mathcal{O}(W)$, while undesirable in general, is often accepted in forward-secure SSE [BF19]. Alternatively, we define $\mathsf{ZeroSSE}'$ to use the fully iterative version of Path-ORAM, which reduces the client storage to $\mathcal{O}(1)$ memory words, at the cost of additional roundtrips, and an additional $\mathcal{O}(\log^2 W)$ bandwidth overhead. (In theory, this could be reduced to $\mathcal{O}(\log W)$ using an optimal ORAM for the position map, but current constructions of optimal ORAM are not practical.)

Theorem 4 (Security of ZeroSSE). *Assuming Path-ORAM is a correct and secure ORAM scheme,* ZeroSSE *is $\mathcal{L}$-secure with respect to the leakage function $\mathcal{L} = \{\mathcal{L}_{\texttt{Setup}}, \mathcal{L}_{\texttt{Search}}, \mathcal{L}_{\texttt{Update}}\}$, with $\mathcal{L}_{\texttt{Setup}} = \{N, U\}$ and $\mathcal{L}_{\texttt{Search}} = \mathcal{L}_{\texttt{Update}} = \varnothing$.*

A proof of Theorem 4 is given in Appendix A.

6.3 BlockSSE

An interesting peroperty of ZeroSSE is that updates are indistinguishable from searches. In fact, addition and deletion of an arbitrary number of documents in a list can be performed in a single interaction at no additional cost. However, this also means that adding a single document to a keyword incurs an $\mathcal{O}(U \log^2 U)$ bandwidth cost. If cheaper updates for single documents are desirable, an alternative solution is to use a smaller blocks size. A smaller block size (linearly) reduces the cost of updates, while (logarithmically) increasing the cost of searches.

BlockSSE offers that trade-off, by storing the reverse index in a wORAM with parametrizable maximum block size B. An interesting feature of BlockSSE is that we can choose the block size B such that the size of a Path-ORAM tree node ZB is one memory page (or an integral number of

memory pages). This optimizes the IO-efficiency of the resulting SSE, as discussed *e.g.* in [BBF+21]. This results in the first SSE with both high memory efficiency and no access pattern leakage.

While BlockSSE may perform significantly better than ZeroSSE in update-heavy workloads, the fact that searches and updates are indistinguishable, regardless of the number of documents added or deleted during an update, is lost. BlockSSE also does not support deletions by default, although they can be added generically at some additional cost, as in [Bos16].

To reduce the size of the position map, we borrow the pointer idea introduced in [WNL+14]. Namely, each block belonging to the same list $\mathsf{DB}(w)$ contains the position of the previous block. This allows the position map, stored on the client, to only store the position of the last block, resulting in $\mathcal{O}(W)$ storage.

We define BlockSSE in more details as follows. Note that `Setup` takes as input additional parameters B, which is the desired block size, and W, and upper bound on the number of keywords. `Update` takes as additional parameter U, which is an upper bound on the longest list size.

- $\mathsf{BlockSSE}.\mathtt{Setup}(\mathsf{K}, N, \mathsf{DB}, U)$: Initializes `Weighted`(`PathOram`) with block size B and number of leaves $n = \lceil N/B \rceil$. For each keyword w, the list $\mathsf{DB}(w)$ is split into $\lceil \mathsf{DB}(w)/B \rceil$ chunks of size at most $B - \lceil \log n \rceil$, with no padding. The i-th chunk for keyword w is inserted into the ORAM at a random position, together with the position of the $(i-1)$-th chunk. The position l_w of the last chunk is stored on the client side.
- $\mathsf{BlockSSE}.\mathtt{Search}(\mathsf{K}, w, U; \mathsf{EDB})$: The client queries the ORAM at position l_w, retrieves the last chunk $\mathsf{DB}(w)$, together with the position of the penultimate chunk. The client iteratively retrieves the position of each previous chunk in the same manner. Each chunk i is assigned a new position uniformly at random, updating the position stored within the next chunk accordingly.
- $\mathsf{BlockSSE}.\mathtt{Update}(\mathsf{K}, (w, e); \mathsf{EDB})$: If l_w is not a multiple of $B - \lceil \log n \rceil$, the client accesses the ORAM at position p_w, adds e to the data, replaces p_w by a new uniformly random position, and updates the ORAM according to this new data and position. If l_w is a multiple of $B - \lceil \log n \rceil$, a new block is inserted at a new uniformly random position, containg as data $\{e\}$ together with the position p_w of the previous last block. On the client side, p_w is updated to the position of the newly inserted block.

Theorem 5 (Security of BlockSSE). *BlockSSE is $\mathcal{L}$-secure with respect to the leakage function $\mathcal{L} = \{\mathcal{L}_{\mathtt{Setup}}, \mathcal{L}_{\mathtt{Search}}, \mathcal{L}_{\mathtt{Update}}\}$, with $\mathcal{L}_{\mathtt{Setup}} = \{N, B\}$, $\mathcal{L}_{\mathtt{Search}}(w) = \lceil |\mathsf{DB}(w)|/B \rceil$, and $\mathcal{L}_{\mathtt{Update}} = \varnothing$.*

A proof of Theorem 5 is given in Appendix A. $\mathsf{BlockSSE}'$ is the same as BlockSSE, except the iterative variant of Path-ORAM is used.

Acknowledgments

This work was supported by the ANR project SaFED.

References

AKL+20. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Heidelberg, May 2020.

ANSS16. Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 1101–1114. ACM Press, June 2016.

BBF+21. Angèle Bossuat, Raphael Bost, Pierre-Alain Fouque, Brice Minaud, and Michael Reichle. SSE and SSD: Page-efficient searchable symmetric encryption. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 157–184, Virtual Event, August 2021. Springer, Heidelberg.

BF19. Raphael Bost and Pierre-Alain Fouque. Security-efficiency tradeoffs in searchable encryption. *PoPETs*, 2019(4):132–151, October 2019.

BFHM05. Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. On weighted balls-into-bins games. In Volker Diekert and Bruno Durand, editors, *STACS 2005*, pages 231–243, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

BFHM08. Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. On weighted balls-into-bins games. *Theoretical Computer Science*, 409(3):511–520, 2008.

Bos16. Raphael Bost. $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1143–1154. ACM Press, October 2016.

CCS17. T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 567–597. Springer, Heidelberg, December 2017.

CGPR15. David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 668–679. ACM Press, October 2015.

CP13. Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. https://eprint.iacr.org/2013/243.

DCRS12. Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346. IEEE Computer Society Press, May 2012.

GLMP18. Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 315–331. ACM Press, October 2018.

GLMP19. Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy*, pages 1067–1083. IEEE Computer Society Press, May 2019.

GMP16. Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 563–592. Springer, Heidelberg, August 2016.

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

GSB+17. Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy*, pages 655–672. IEEE Computer Society Press, May 2017.

HK22. Shai Halevi and Eyal Kushilevitz. Random-index oblivious ram. Cryptology ePrint Archive, Paper 2022/982, 2022.

KM19. Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 183–213. Springer, Heidelberg, May 2019.

KMO18. Seny Kamara, Tarik Moataz, and Olga Ohrimenko. Structured encryption and leakage suppression. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 339–370. Springer, Heidelberg, August 2018.

LHL+18. Zheli Liu, Yanyu Huang, Jin Li, Xiaochun Cheng, and Chao Shen. DivORAM: Towards a practical oblivious RAM with variable block size. *Information Sciences*, 447:1–11, 2018.

LN18. Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018.

MLS+13. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 311–324. ACM Press, November 2013.

MM17. Ian Miers and Payman Mohassel. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS 2017*. The Internet Society, February / March 2017.
MOA79. Albert W Marshall, Ingram Olkin, and Barry C Arnold. *Inequalities: theory of majorization and its applications*, volume 143. Springer, 1979.
MR22. Brice Minaud and Michael Reichle. Dynamic local searchable symmetric encryption. In Y. Dodis and T. Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, Lecture Notes in Computer Science. Springer, 2022.
PPYY19. Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 79–93. ACM Press, November 2019.
RAC16. Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy*, pages 178–197. IEEE Computer Society Press, May 2016.
RYF+13. Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. Design space exploration and optimization of Path Oblivious RAM in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 571–582, 2013.
SvS+13. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.
TW07. Kunal Talwar and Udi Wieder. Balanced allocations: the weighted case. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 256–265. ACM Press, June 2007.
WCS14. Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. Cryptology ePrint Archive, Report 2014/672, 2014. https://ia.cr/2014/672.
WHS12. Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 314–328. Springer, Heidelberg, February / March 2012.
WNL+14. Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 215–226. ACM Press, November 2014.

A SSE Proofs

A.1 Proof of ZeroSSE

The simulator $\mathcal{S}$ for ZeroSSE behaves as follows.

- `Setup`: on input $\mathcal{L}_{\texttt{Setup}} = (N, U)$, $\mathcal{S}$ creates a wORAM instance `Weighted(PathOram)` instance with block size U and number of blocks $\lceil N/U \rceil$, initialized with $\lceil N/U \rceil$ items containing dummy data of size B. Let a denote the address of one of these items.
- `Search` and `Update`: in both cases, $\mathcal{S}$ queries the wORAM instance on item a with $\texttt{op} = \text{read}$.

We claim that the resulting transcript is indistinguishable from the real transcript. In fact, this follows immediately from the security of `Weighted(PathOram)`, since the guarantees of a wORAM scheme is that any two sequence of accesses of the same length are indistinguishable, and the simulator makes the same number of accesses as the client in the real game. Moreover, Theorem 1 implies that `Weighted(PathOram)` is a correct wORAM scheme.

Note that the analysis relies on the fact that Path ORAM is correct for the given parameters, which in turn implies that we require $N/U = \Omega(\lambda)$. If $N/U < \lambda$, it may not be worth using an elaborate ORAM scheme, since the overhead of trivial ORAM is less than λ.

A.2 Proof of BlockSSE

The simulator $\mathcal{S}$ for BlockSSE behaves as follows.

- `Setup`: on input $\mathcal{L}_{\texttt{Setup}} = (N, B)$, $\mathcal{S}$ creates a wORAM instance `Weighted(PathOram)` instance with block size B and number of blocks $\lceil N/B \rceil$, initialized with $\lceil N/B \rceil$ items containing dummy data of size B. Let a denote the address of one of these items.
- `Update`: $\mathcal{S}$ queries the wORAM instance on item a with $\texttt{op} = \text{read}$.
- `Search`: on input $\mathcal{L}_{\texttt{Search}} = x$, $\mathcal{S}$ queries the wORAM instance x times on item a with $\texttt{op} = \text{read}$.

We claim that the resulting transcript is indistinguishable from the real transcript. Again, this follows immediately from the security of `Weighted(PathOram)`, since the guarantees of a wORAM scheme is that any two sequence of accesses of the same length are indistinguishable, and the simulator makes the same number of accesses as the client in the real game. Moreover, Theorem 1 implies that `Weighted(PathOram)` is a correct wORAM scheme. As with ZeroSSE, that last point assumes that $N/B = \Omega(\lambda)$.

B A General Transformation

In this section we present a way to transform any standard ORAM into a variable ORAM, at the cost of some blowup in the overhead. We have m objects, of individual size $w_i \in [0, B]$ for $i \in [k]$ such that the sum of the sizes is N. The idea is:

- We create a standard ORAM structure for N objects, each of the same size higher than B (we discuss the size we need later). Let us call these N objects in the standard ORAM *super objects*.
- We want to store our m objects of varying sizes *inside* of the super objects. To do so, we map every object $i \in [m]$ to a super object $H(i) \in [N]$.
- When we wish to access the object i, we make a standard ORAM access to the super object $H(i)$, and read the object from the super object.

This technique yields the same computational, storage and bandwidth metrics as the ORAM we are modifying, with the caveat that the object size must be bigger.

The crucial part here is the function H. We must choose a way of hashing the m objects into the N super objects while having the following properties:

- A super object cannot overflow, except with negligible probability.
- The hash table should be modified as little as possible, so as to save bandwidth.
- The super objects should be as small as possible.

We present two hash algorithms based on the balls-in-bins problem that can be used to achieve this type of weighted ORAM.

B.1 One-Choice Process: $\mathcal{O}(\log N)$ blowup

We let each object be associated with a super object uniformly at random. In other words, $H : [m] \to [N]$ is a random function. At each access, to get object i we call the access algorithm of the ORAM on super object $H(i)$, modify object i in it, re-encrypt the super-object with fresh randomness, and write back the super object. It suffices to have each super object be of size $B \cdot \log N$.

Proof. Let m balls, each of weight $w_i \in [0,1]$ such that $\sum_{i=1}^{m} w_i = N$, be stored in N bins, which we associate uniformly at random. Let us call S the bin with the highest load. Let us call $\mathbf{w}$ the current weight distribution of the balls, and $\mathbf{u}$ be the weight distribution when we have only N balls of size 1, that is:

$$\mathbf{u} = (\underbrace{1,\dots,1}_{N},\underbrace{0,\dots,0}_{m-N}).$$

From the majorization argument in [BFHM05], we know that for any $\mathbf{w}$, $\mathbb{E}(\sum_{i\in S} w_i) \leq \mathbb{E}(\sum_{i\in S} u_i)$. Since the max function is convex and the expectancy is linear, we have $\mathbb{E}(\max(0,(\sum_{i\in S} w_i) - \log N)) \leq \mathbb{E}(\max(0,(\sum_{i\in S} u_i) - \log N))$. From now on, we will let the random variable $X(\mathbf{v})$ denote $\max(0,(\sum_{i\in S} u_i) - \log(N))$.

We have $\mathbb{P}(X(\mathbf{w}) > 0) \leq \mathbb{E}(X(\mathbf{w}))$. The event $X(\mathbf{w}) > 0$ is exactly the event of having at least one overflow in a given bin, so to prove that it happens with negligeable probability, it suffices to prove that $\mathbb{E}(X(\mathbf{u}))$ is negligeable. We have:

$$\begin{aligned}
\mathbb{E}(X(\mathbf{u})) &= \sum_{k=0}^{\infty} \mathbb{P}(X(\mathbf{u}) > k) \\
&\leq N \cdot \mathbb{P}(X(\mathbf{u}) > 0) \\
\mathbb{P}(X(\mathbf{u}) > 0) &= \mathbb{P}[\text{a given bin has more than } k \text{ balls}] \text{ with } k = \log N \\
&\leq \binom{N}{k} \cdot \frac{1}{N^k} \\
&\leq \frac{N^k}{k!} \cdot \frac{1}{N^k} \\
&= \frac{1}{k!} \\
&= \text{negl}(N) \text{ via Stirling's formula.}
\end{aligned}$$

□

B.2 Layered Two-Choice Process: $\mathcal{O}(\log\log N)$ blowup

The Layered 2-choice method of hashing m balls into N bins was introduced in [MR22, Section 4]. The idea is to split the m objects into "layers" according to their sizes. Then we can store them in the super objects by running standard (unweighted) 2-choice balls-in-bins allocation separately for each layer. Such an allocation is achievable with overhead $\mathcal{O}(\log\log N)$.

However this algorithm does not support offline modification of the bins. This means that after an access for object i, we must update the ORAM by selecting 2 new super objects uniformly at random and write i in one of those super objects. The first consequence is that we double the bandwidth. Another problem is that the hash table has to be modified. To prevent client side spatial blowup, we must store the hash table (effectively a position map) on the server. This may be realized using a second ORAM (which need not use the same ORAM protocol).

Overall, while this approach is generic, it is significantly less efficient in practice than our main construction, which incurs almost no overhead relative to standard Tree ORAM.

C A Flaw in DivORAM

DivORAM was introduced in [LHL+18]. The DivORAM construction is (essentially) a weighted ORAM following a Tree ORAM structure: blocks are stored in buckets, and each bucket is located

at the node of a binary tree. The blocks are decomposed into so-called splinters, which are stored in the buckets. A bucket at level $i \in [\log N]$ can store up to 2^i splinters.

Stash analysis is given in [LHL+18, Section 5.3]. Whenever a splinter cannot be stored in a tree bucket, it ends up in the stash. The worst case is when an entire block, composed of $\log N$ splinters, cannot be stored in the tree: all splinters are then moved to the stash. The stash capacity is $2\log(N) - 1$. To avoid overflows, the content of the stash is written back into the tree whenever its size exceeds $\log N$.

Such a behavior reveals to the server when the stash size passes the $\log N$ threshold. Unfortunately, leaking when the stash size passes a threshold breaks Path ORAM security, because certain access patterns fill the stash faster than others. For more information, we refer the reader to [RYF+13, Section 3.1.1], where this exact point is discussed. Because of this leakage, the DivORAM construction of [LHL+18] is insecure.

D Weighted Hierarchical ORAM

Our work focuses on Tree ORAMs, mainly because of their higher practical efficiency. It is natural to ask whether similar results can be obtained for hierarchical ORAM. We briefly discuss this question here.

As shown in this article, there is a natural way to adapt Tree ORAM algorithms to handle weighted items. The problem of building weighted Tree ORAM is not so much an algorithmic problem as a combinatorial one: we need to prove that the "natural" weighted variant of Tree ORAM creates no overflow.

With hierarchical ORAMs, this first step is already a challenge. There is no obvious natural algorithmic modification of hierarchical ORAMs that handles weighted items. Indeed, we would first need weighted oblivious hash tables. That is, we need oblivious hash tables that can handle items of variable size. We also need those tables to support oblivious sorting, or at least oblivious compaction, in a way that is compatible with variable-size items. Building such a data structure looks quite challenging, and we do not know of a natural candidate.

In more details, we know of two constructions that come close to that goal in the literature, but neither quite reaches it. The first is two-choice hashing for weighted items [TW07, MR22]. Two-choice hashing is easy to realize obliviously. The problem is that we would have to pay (at least) a $\log\log$ overhead, and efficient oblivious compaction seems difficult. An alternative approach is to use the "Data-Independent Packing" primitive from [BBF+21]. But the only known instantiation requires computing a maximum flow algorithm. It is not clear how to compute that algorithm in quasilinear time, let alone in an oblivious manner.