EnigMap: Signal Should Use Oblivious Algorithms for Private Contact Discovery

Afonso Tinoco [1,2]
atinoco@andrew.cmu.edu

Sixiang Gao [1]
sixiangg@andrew.cmu.edu

Elaine Shi [1]
runting@gmail.com

[1] Carnegie Mellon University, [2] Instituto Superior Técnico

***Abstract*—Leveraging hardware enclaves technology, Signal was the first to offer a *privacy-preserving* contact discovery service, where users can discover whether their friends have signed up for the service, without divulging their entire address books. The crux of their design is an algorithm to search for the user's contacts such that the access patterns are independent of the queries.**

To achieve this, Signal implemented a naïve batched linear scan algorithm that scans through the entire database for each batch of queries. Signal published a high-profile blog post arguing that for billion-sized databases, batched linear scan outperforms the asymptotically superior oblivious algorithms. While subsequent works revisited the same question, we still do not have conclusive evidence why Signal should use oblivious algorithms instead.

Our work is motivated by the observation that the previous enclave implementations of oblivious algorithms are suboptimal both asymptotically and concretely. We make the key observation that for enclave applications, the number of page swaps should be a primary performance metric. We therefore adopt techniques from the external-memory algorithms literature, and we are the first to implement such algorithms inside hardware enclaves. We also devise asymptotically better algorithms for ensuring a strong notion of obliviousness that resists cache-timing attacks. We complement our algorithmic improvements with various concrete optimizations that save constant factors in practice. The resulting system, called EnigMap, achieves 5.5× speedup over Signal's linear scan implementation, and 21× speedup over the prior best oblivious algorithm implementation, at a realistic database size of 256 million and a batch size of 1000. The speedup is asymptotical in nature and will be even greater as Signal's user base grows.

1. Introduction

Mobile messenger applications such as Signal and WhatsApp provide a contact discovery service to its users. In contact discovery, a client with a local address book wants to discover which of the friends in its address book have signed up for the messenger service. The most straightforward way to accomplish this is for the client to upload its entire address book to the server — indeed, this is the approach taken by WhatsApp [1]. The drawback is clear: the client completely loses control of its private data.

To protect the client's privacy, Signal, a private messaging service, deployed hardware-assisted techniques for private contact discovery [2]. In this way, Signal users can discover contacts without revealing their address book to the server. Signal published a blog post [2] that explains their algorithm in detail. The Signal server is equipped with an Intel SGX secure processor. Such a secure processor creates a trusted enclave on the server that acts as a root of trust. The client can encrypt its address book to the enclave's public key, such that the address book can be decrypted only within the enclave, and any data that leaves the enclave is encrypted again. It is a long known fact that encryption alone is not sufficient for achieving privacy. In the case of contact discovery, the server has a large database of users (e.g., a billion entries [2]) that resides in external memory, i.e., untrusted memory and disk. If the server can observe which entries are matched through the access patterns, then privacy is lost. To clarify, the server's database itself is not secret, but the client's address book is.

A standard approach for hiding a program's access patterns is called Oblivious RAM (ORAM), originally proposed by Goldreich and Ostrovksy [3], [4]. A line of subsequent works aimed to make ORAM practical [5]–[7]. Among these works, Path ORAM [6] stands out as one of the most practical ORAM algorithms known to date. With Path ORAM, we can compile any program to an oblivious counterpart incurring only an $O(\log^2 N)$ factor overhead in runtime. Therefore, one straightforward solution is to use Path ORAM to compile a standard (non-oblivious) binary search tree algorithm (e.g., AVL tree [8]), resulting in an oblivious binary search tree. This approach would allows us to perform each search in $O(\log^3 N)$ time.

1.1. Oblivious Algorithms vs. Batched Linear Scan: The Race is On

Signal's findings. Naturally, Signal investigated using Path ORAM for their private contact discovery application. In 2017, they published a high-profile blog post [2] discussing their findings. They argue that in practice, for billion-sized databases, Path ORAM is not as efficient as a naïve batched linear scan algorithm. Specifically, the batched linear scan algorithm makes a pass over the entire database to answer a batch of β queries, where each query specifies a user's handle (e.g., email address or phone number), and recall that our goal is to return the corresponding entry from the database if found. Clearly, the linear scan incurs $O(N)$ runtime for each batch, and amortized to each query, the cost is $O(N/\beta)$. Although batched linear scan is asymptotically much worst than Path ORAM, Signal argued that it has better concrete performance than Path ORAM. Therefore, this is the algorithm they eventually implemented in their open source repository [9].

Oblix's findings. Oblix [10] pointed out that rather than using Path ORAM to compile a non-private binary search tree algorithm into an oblivious counterpart, Signal should have adopted an oblivious binary search tree [11], [12] instead. At a very high level, an oblivious binary search tree [12] piggybacks the binary search tree's index structure on top of the position-map data structure of Path ORAM [11]. This trick allows us to support each search query in $O(\log^2 N)$ time, which is a logarithmic factor more efficient than the generic compilation approach.

The authors of Oblix then implemented an efficient oblivious binary search tree inside the SGX enclave, and evaluated its performance. They showed that at a batch size of 1000, their implementation starts to outperform Signal's implementation when the database contains billions of entries; at a batch size of 100, their implementation starts to outperform Signal at a database size of roughly 100 million.

In summary, Oblix's findings show that oblivious algorithms are definitively more competitive when the batch sizes are small (e.g., 100 or smaller). However, if batch sizes are large in practice (e.g., 1000), then the reason for Signal to switch to oblivious algorithms is not compelling enough.

Other related work. Some other related works such as ObliDB [13] can also be used to support Signal's contact discovery application. However, as acknowledged in their own paper [13], their performance is not as competitive as Oblix for this application, since ObliDB aims to support general relational database queries. We therefore use Oblix as a state-of-the-art baseline in this paper. We review additional related work in Section 1.4.

1.2. Sub-Optimality of Oblix's Implementation

Our work was initially motivated by the observation that Oblix's implementation [10] turns out to be sub-optimal both asymptotically and concretely. There are two main reasons that leads to the sub-optimality.

Choice of metric. Most of the theoretical work on oblivious algorithms [3]–[6] use the program's *runtime* (i.e., number of instructions) in a word-RAM model as a primary performance metric, and consequently Oblix adopted the runtime as primary metric too. Henceforth in the paper, whenever we say *runtime*, we mean the *number of instructions* — this is standard terminology from the algorithms literature.

However, for an enclave-based setting, runtime should only be considered as a secondary metric. The more significant performance bottleneck is actually the overhead of page swaps between the enclave and the untrusted operating system (OS). As we know, the hardware enclave has limited resident memory (e.g., 128MB) and cannot cache the entire database (stored in an oblivious data structure). Whenever the enclave wants to access the oblivious data structure, it needs to ask the OS to help perform a page swap. The page swap is an expensive operation for a few reasons. First, it incurs a context switch which is a heavy-weight operation itself. Second, the page swap must fetch and store data at a 4KB granularity even if the algorithm only wants to access a single byte. Third, although the original database is not secret, the oblivious data structure must nonetheless be encrypted. During this page swap, the enclave needs to decrypt the data fetched from external memory, and when writing the page back, the enclave must re-encrypt it. We stress that this encryption/decryption process happens in software and should be differentiated from the enclave's memory encryption mechanism which is performed in hardware. Further, even if the enclave's algorithm wants to read only a single byte out of the 4KB (i.e., it only needs to decrypt 1 byte), when writing the page back, it must nonetheless re-encrypt the entire page such that the OS cannot tell which byte was modified.

In summary, the combination of the context switch, page transfer, and encryption/decryption overhead makes page swap a major bottleneck for enclave applications. Oblix did not optimize for this metric in their algorithm and implementation.

Sub-optimal algorithm for strong obliviousness. The original Path ORAM [6] and oblivious data structure [12] papers presented the ORAM algorithm in a client-server setting. Since the client is trusted, the ORAM client algorithm itself need not be oblivious. In the context of hardware enclaves, a security concern is cache-timing attacks [14]–[17]. By exploiting the fact that the enclave shares the CPU caches with the untrusted applications running on the same machine, the adversary can potentially exploit cache-timing attacks to learn the access patterns *within* the enclave.

To defend against cache-timing attacks, several works in this space [18], [19], including Oblix [10] suggested a strong notion of obliviousness, often referred to as *strong obliviousness* [18], [19] or *double obliviousness* [10]. With strong obliviousness, we not only require that the page-level access patterns be oblivious, but also that the access patterns within the enclave be oblivious too. In other words, we want to make the ORAM client itself oblivious too.

Although there exist known approaches for making the ORAM client oblivious [7], [20], Oblix [10] came up with their own techniques for this purpose. In particular, to perform the eviction algorithm along an ORAM tree path, part of their algorithm performs a double-loop over the logarithmically sized tree path, thus incurring $O(\log^2 N)$ overhead. This is asymptotically suboptimal in comparison with the best known algorithm [7], [20].

1.3. Our Results and Contributions

We revisit the "oblivious algorithm vs. linear scan" question in the context of hardware enclaves. We design

TABLE 1: **Asymptotical comparison.** N is the maximum number of entries in the multimap, B denotes the page size (typically 4KB), M is the size of the enclave's resident memory (up to 128MB), and β is the batch size. $\widetilde{O}(\cdot)$ hides $\mathsf{poly}\log\log$ factors.

Scheme	Cost per batch of operations		Cost of initialization	
	page swaps	runtime	page swaps	runtime
Signal [2]	$O(N/B)$	$O(\beta^2 + N)$	$O(N/B)$	$O(N)$
Oblix [10]	$O(\beta \log^2 N)$	$O(\beta \log^3 N)$	$O(\frac{N}{B} \log^2 N)$	$O(N \log^3 N)$
EnigMap	$O(\beta \log_B N \cdot \log N)$	$\widetilde{O}(\beta \log^2 N)$	$O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$	$\widetilde{O}(N \log N)$

and implement EnigMap, an enclave-assisted multi-map data structure that can be used to perform key-value lookups and support private contact discovery. **Under a realistic database size of 256 million, we achieve 5500×, 530×, 53×, 5.5× speedup** w.r.t. Signal's open source implementation, at a batch size of 1, 10, 100, and 1000, respectively. In comparison with Oblix, we achieve a **21× speedup** regardless of the batch size. Our performance gain is asymptotical in nature, so for billion-sized databases, our speedup w.r.t. both Signal and Oblix will be even greater.

At first sight, it may seem surprising that we can achieve such a big speedup over Oblix despite the relative maturity of this line of work — specifically, oblivious algorithms are simple data structures that have been thoroughly explored in numerous practical settings [13], [21]–[27]. Indeed, our savings come not just from system-level optimizations but also from asymptotical improvements (see Table 1).

We adopt ideas from an elegant line of work that originated in the algorithms community, called *external-memory* algorithms [28], [29]. In external-memory algorithms, we care about minimizing the number of *cache misses* of a program. It turns out that the external-memory model is a perfect fit for enclave applications where we can think of the enclave memory as a cache, and think of page swaps as cache misses. Although external-memory algorithms are a well-known body of work [28], [29], to the best of our knowledge, we are the *first to implement and evaluate such algorithms in the hardware enclave context*. We now explain our contributions more concretely.

Locality friendly layout. Recall that in an oblivious binary search tree algorithm that leverages a Path ORAM tree as an underlying data structure [12], every search query requires visiting $O(\log N)$ paths in the ORAM tree (where each path travels from the root to some leaf). Inspired by known external-memory algorithms [30], [31], we adopt a *locality-friendly layout* for storing the (encrypted) ORAM tree in external memory. This allows us to incur only $O(\log_B N)$ page swaps for visiting a path where B denotes the page size. In comparison, Oblix uses a simple heap layout for storing the ORAM tree, and they incur $O(\log N)$ overhead for visiting a path.

Efficient initialization algorithm. We devise new algorithms for initializing the oblivious data structure that achieve asymptotical savings relative to Oblix's approach. Our new initialization algorithm also adopts ideas from the external-memory algorithms literature such that we can optimize the number of page swaps. As shown in Table 1, our initialization algorithm incurs $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps and $O(N \log N)$ runtime, where B is the page size and M is the enclave's resident memory size. In comparison, Oblix's initialization algorithm incurs $O(N \log^2 N)$ page swaps and $O(N \log^3 N)$ runtime.

Ensuring strong obliviousness. In comparison with Oblix, we employ asymptotically more efficient algorithms [20] for ensuring strong obliviousness that allows us to save a logarithmic factor in runtime (see Table 1).

We examined Oblix's code and point out various subtleties for ensuring strong obliviousness at an implementation level. We rely on known techniques [32], [33] to avoid these pitfalls and guarantee both data obliviousness and instruction-trace obliviousness within the enclave.

Practical optimizations. We introduce various optimizations that would save us a constant factor in practice. We adopt a multi-level cache design that involves a page-level cache outside the enclave, a bucket-level cache inside the enclave, and a binary-search-tree-level cache. Moreover, we suggest a new optimization that saves a $2\times$ to $3\times$ factor for the data structure's insertion algorithm. Our locality-friendly layout also achieves a constant-factor saving in comparison with the standard Emde Boas layout [30], [31] from the external-memory algorithms literature.

Implementation and open source. We have implemented EnigMap and open sourced the code at https://github.com/odslib/EnigMap (the site has been anonymized for the submission). Besides the numbers mentioned earlier, we provide a more in-depth performance evaluation, and show the breakdown of performance overhead, as well as the effect of various optimizations.

Signal has between 40 million to billions of users [2], [34], [35]; moreover, their user base has been rapidly growing [35]. Based on our findings and these known statistics, *we conclude that it would be compelling for Signal to switch to using oblivious algorithms* for their private contact discovery service.

1.4. Additional Related Work

Oblivious RAM and oblivious algorithms. Oblivious RAM was first proposed by Goldreich and Ostrovsky [3], [4] who gave a hierarchical construction with $O(\log^3 N)$

overhead, assuming the existence of one-way functions. Several subsequent works [36]–[41] made some further improvements on top of the hierarchical framework. Shi et al. [5] first proposed a new binary-tree based paradigm for constructing ORAM, which removes the assumption on one-way functions. The framework was improved in several subsequent works [6], [7], [23], and Path ORAM [6] stands out as one of the most practical algorithms for a secure processor or client-server setting.

While ORAM allows us to compile any program into an oblivious counterpart generically, a line of work has focused on customized oblivious algorithms [12], [32], [42] such that we can outperform generic compilation for specific (classes of) computation tasks. Wang et al. [12] showed oblivious algorithms for binary search trees that outperform generic ORAM compilation by a logarithmic factor — both Oblix and our work are based on their algorithm.

Oblivious algorithms meet hardware enclaves. Besides the aforementioned works Oblix [10] and ObliDB [13], the recent work Snoopy [21] also implemented oblivious algorithms in a hardware enclave context. Snoopy's focus, however, is how to *parallelize* multiple instances of oblivious data structures to increase *throughput*. In their experiments, they used Oblix [10] as one choice of a single instance. In this sense, ENIGMAP is orthogonal and complementary to Snoopy, and it should not be hard to replace Oblix with ENIGMAP in Snoopy's implementation which should lead to significantly better performance.

Earlier, several works such as Raccoon [43], Zero-Trace [27] and Kloski [44] also implemented variants of Path ORAM inside an SGX enclave. However, they did not implement an efficient oblivious binary search tree, and thus would not be competitive for our application.

Oblivious algorithms in a client-server setting. Another line of works, including Ring ORAM [23], Obladi [22] and others [12], [36], [37], [45] implemented oblivious algorithms in a client-server setting. In a client-server setting, the main performance bottleneck is the bandwidth, and the client-side algorithm need not be implemented obliviously.

Other approaches for private contact discovery. Private set intersection (PSI) [46] is a cryptographic protocol that allows two parties to compute the intersection of their respective sets, without revealing additional information about their private sets. PSI techniques can also be applied to private contact discovery. Moreover, a line of works [47]–[52] have focused on optimizing the asymptotical and concrete performance of psi. However, the recent work of Kales et al. [48] pointed out that PSI is not ready yet to scale to billion-sized databases such as in Signal's scenario.

2. Problem Statement

In private contact discovery [2], the server has a database of users each identified by some handle such as email or phone number. Each user's record may store some extra information of the user, e.g., geographical location, last time online, etc. The client has a list of friends, and it wants to retrieve the records corresponding to its friends. The client is privacy-conscious, i.e., it does not want to leak to the server who its friends are.

2.1. Architecture and Threat Model

We assume that the server is equipped with secure hardware enclaves such as Intel's SGX. Although recent works have uncovered some attacks for off-the-shelf trusted hardware [53], [54], it is outside the scope of this paper to consider how to design provably secure trusted hardware — an orthogonal line of work focuses on this goal [55]–[57]. Although we use Intel's SGX as a testbed to demonstrate our ideas, our algorithmic constructions are generic and apply to known hardware enclave technologies in general.

As our threat model, we assume that the server's operating system may be compromised, and there may be insiders in the facilities hosting the server who can perform physical attacks — we assume that the physical attacks cannot break the tamper-resistance of the hardware enclave.

Hardware encryption of enclave memory. A physical attacker can perform cold-boot style attacks to read off memory contents. However, recall that all the enclaves' memory pages are *hardware-encrypted* which prevents a cold-boot attacker from learning their contents.

Enclave's page swaps and software encryption of pages. The enclave's page swaps are performed by the untrusted operating system (OS). To prevent the OS from observing secret contents of any page that is swapped out, the pages must be encrypted before they are handed over the OS. Similarly, pages swapped into the enclave must be decrypted before computation can take place. The above encryption/decryption processes are performed in software.

Access pattern leakage channel. When handling the enclave's page swaps, the untrusted OS can directly observe the page-level access patterns of the enclave program, i.e., the indices of the memory pages that the enclave swaps in and out. We also assume that the attacker can perform cache-timing style attacks to glean information about the memory accesses made by the program running inside the enclave.

2.2. Oblivious Multimap

The private contact discovery problem can be solved with a map data structure that implements a key-value store where all keys are distinct. Just like in the Oblix [10] paper, we define a slightly more expressive data structure, that is, a *multimap*, whose abstraction is defined below:

- $\mathsf{Init}(\mathbf{I})$: on receiving an input array $\mathbf{I}$ containing key-value pairs, initialize a multi-map data structure;
- $\mathsf{Size}(k)$: return the number of occurrences of the key k in the data structure;
- $\mathsf{Find}(k, i, j)$: on receiving a key k, two indices i and j where $j \geq i$, return the i-th to the j-th key-value pair whose key matches k. If multiple entries exist with the same key k, we order all the entries based on the value

v and index them based on this ordering. Note that the array output by Find is of fixed length $j - i + 1$, and if there are not enough key-value pairs with the key k, we simply pad the output array with filler entries of the form $\perp$;

- $\mathsf{Insert}(k, v)$: insert a key-value pair of the form (k, v) into the data structure;
- $\mathsf{Delete}(k, v)$: delete one occurrence of the key-value pair (k, v).

Correctness is implied by the above definition of the abstraction. We now define obliviousness.

Strong obliviousness. Since all data that leaves the secure hardware enclave will be encrypted, we may assume that the adversary can only observe the access patterns but not the data content itself. The following definition intuitively says that the adversary cannot learn any secret information (besides the types of the requests and the lengths of inputs and outputs) by observing the access patterns.

Let $\{\mathsf{op}_\ell\}_{\ell\in[m]}$ be a request sequence of length m where each op_ℓ is of the form (Size, k), (Find, k, i, j), (Insert, k, v), or (Delete, k, v). $\{\mathsf{op}_\ell\}_{\ell\in[m]}$We say that two request sequences $\{\mathsf{op}_\ell\}_{\ell\in[m]}$ and $\{\mathsf{op}'_\ell\}_{\ell\in[m]}$ are *trace-equivalent*, iff

1) they have the same length;
2) each op_ℓ and op'_ℓ have the same type of operation; and
3) if $\mathsf{op}_\ell = (\mathsf{Find}, i, j)$ and $\mathsf{op}'_\ell = (\mathsf{Find}, i', j')$ are both Find operations, then it must be that $j' - i' = j - i$.

Given some initial array $\mathbf{I}$, let $\mathsf{Accesses}(\mathbf{I}, \{\mathsf{op}_\ell\}_{\ell\in[m]})$ be the access patterns observed when initializing the array with $\mathsf{Init}(\mathbf{I})$ followed by executing the request sequence $\{\mathsf{op}_\ell\}_{\ell\in[m]}$. Here the access patterns including the sequence of physical locations visited as well as whether each physical access is a read or write operation. Moreover, the access patterns include both the *instruction fetches* as well as the *memory requests* made by the program. We say that a multi-map implementation satisfies *obliviousness*, iff there exists a negligible function $\mathsf{negl}(\cdot)$, such that for any two input arrays $\mathbf{I}$ and $\mathbf{I}'$ of the same length, for any two trace-equivalent request sequences $\{\mathsf{op}_\ell\}_{\ell\in[m]}$ and $\{\mathsf{op}'_\ell\}_{\ell\in[m]}$, the random variables $\mathsf{Accesses}(\mathbf{I}, \{\mathsf{op}_\ell\}_{\ell\in[m]})$ and $\mathsf{Accesses}(\mathbf{I}', \{\mathsf{op}'_\ell\}_{\ell\in[m]})$ have $\mathsf{negl}(\lambda)$ statistical distance, where λ is a security parameter, and we assume that the multi-map is invoked with the security parameter λ.

We stress that our obliviousness notion is very strong and inherently resists cache-timing-style attacks. Since we require that even *word-level* (not just page-level) access patterns be oblivious, it means that even if the adversary can observe the memory accesses of the program inside the enclave (e.g., through cache-timing attacks), it cannot learn any secret information. Earlier works have referred to this notion as either *strongly oblivious* [18], [19] or as *doubly oblivious* [10]. In a practical implementation, our notion requires obliviousness not only on the data accesses, but also the *instruction trace*, and we will discuss how we ensure instruction-trace obliviousness in Section 5.2.

2.3. Microbenchmarks and Performance Metrics

What should be the primary performance metric? To the best of our knowledge, almost all prior works [10], [27], [43], [44] that implement oblivious algorithms for hardware enclaves focus on optimizing the *runtime* (i.e., number of instructions) of the algorithm — using standard algorithmic terminology, these works consider the enclave program as a Random Access Machine (RAM), and use the RAM's runtime as a primary metric. We observe that this is actually not the right metric for characterizing the performance of enclave programs.

In hardware enclave architectures, the enclave typically has a limited amount of resident memory. When the resident memory is not enough, the enclave needs the operating system's help to swap pages in and out of the enclave, from insecure memory (or disk). In common hardware enclave architectures today, such page swaps are performed at a 4KB granularity — even if the enclave just wants to read a single byte that resides outside the enclave, it has to perform a 4KB page swap, since the page is the atomic unit of I/O in and out of the enclave.

Therefore, besides the algorithm's runtime, *another primary metric for an enclave program is the number of page swaps*. A page swap is a heavy-weight operation since 1) context switches are necessary for performing page swaps; and 2) we need to perform software encryption and decryption of memory pages (see also Section 2.1). Both of the above are expensive operations. To better illustrate the performance profile, we conducted microbenchmarking of the overheads of various operations. Figure 1 shows the cost of the following operations relative to `MOV-4096`, i.e., moving 4KB data inside the enclave's secure memory without triggering any page swap:

- Encryption/Decryption - encrypting/decrypting 4KB of data using AES256-GCM via AES-NI.
- OCall - moving 4KB of data from inside the enclave to outside followed by moving 4KB of data to the enclave.
- DiskSwap - writting a 4KB page to disk followed by reading a 4KB page from disk.

In the figure, the cost of `MOV-4096` is normalized to $1\times$. We see that the cost of a page swap is $30\times$ to $40\times$ more expensive than `MOV-4096`.

The external memory model. In the algorithms literature, there is an elegant line of work called *external-memory* algorithms [28], [29] that is a perfect fit for modeling the performance of hardware enclave architectures. To understand the external memory model, it is easiest to contrast it with the standard RAM model. In the RAM model, the atomic unit of data for CPU instructions and for I/O are the same, and thus the number of CPU instructions (commonly referred to as the RAM's *runtime*) also coincide with the number of I/Os, assuming each CPU step performs a single memory read and write. The external memory model, first proposed Aggarwal and Vitter [28], considers an abstract machine where the atomic unit of data for I/O, often called a *block*, is larger than the unit of data that CPU instructions

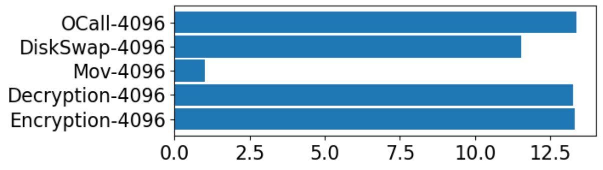


Figure 1: **Microbenchmarks:** A page swap between inside and outside the enclave incurs the OCall overhead, the decryption overhead, and possibly part of the encryption overhead depending on how many bytes are read. A page swap is about 30× to 40× more expensive than moving 4KB in memory *within* the enclave.

operate on. Even if the CPU wants to read just one word from memory, it has to fetch the entire block that the word resides in. Further, the CPU has some cache of size M that can cache some blocks it has recently fetched. The primary performance metric for an external-memory algorithm is the number of *cache misses* or equivalently, the *I/O cost* (i.e., the number of blocks fetched from or written to memory).

In the context of hardware enclaves, the page size (typically 4KB) corresponds to the block size B, and the enclave's resident memory size corresponds to the cache size M (typically 128MB). We will focus on optimizing the number of page swaps, which corresponds to the I/O cost of an external-memory algorithm.

In the external-memory model, intuitively we want algorithms with good *locality*. In other words, once the CPU fetches an entire block from memory, it had better make maximal utilization of this block before caching it out to memory again.

3. Background on Oblivious RAM and Oblivious Data Structures

In this section, we provide some background on Path ORAM [6] and oblivious data structures [12].

Terminology. As mentioned, in this paper, we study algorithms in the external-memory model. To avoid terminology collision, we differentiate the terms *entry* and *page*. An *entry* is an atomic unit that the data structure operates on, whereas a *page* is the minimal unit of I/O between the enclave and the outside world. Typically a page is 4KB in size, and it can contain multiple entries.

In comparison, the standard literature on oblivious algorithms adopts the RAM model, and thus does not differentiate an entry and a page — both are referred to as a *block* [5], [6]. However, we must differentiate them since we are now in the external-memory model.

3.1. Non-Recursive Path ORAM

Path ORAM, proposed by Stefanov et al. [6], is an efficient instantiation of the tree-based ORAM framework [5]. We will actually use the Path ORAM data structure to realize an oblivious AVL tree — to do this, we do not need the particular recursion structure of Path ORAM [6] since we will override it with the logical indexing structure of the AVL tree. Therefore, below we introduce the background on the pre-recursion data structure of Path ORAM.

Data structure. The primary data structure of Path ORAM is a binary tree (henceforth called the *ORAM tree*) with N leaves where N is the maximum number of elements in the multimap. Each node in the tree is called a bucket. A bucket can hold up to 4 entries, and an entry can either be *real* or a *filler*. Filler entries do not store actual information, they are just there for security. Jumping ahead, later in our oblivious data structure application, each real *entry* will correspond to a node in the logical AVL tree.

Additionally, there is also a *stash* whose size is super-logarithmic in the security parameter for holding overflowing entries. Henceforth, we simply assume that the stash is part of the root node, i.e., the root node has somewhat larger size than the remaining nodes in the ORAM tree.

Path invariant. Each entry is assigned to a random path (i.e., from the root to some random leaf node) in the ORAM tree. The entry can reside in any bucket along its designated path. Henceforth, we use the term *position identifier* denoted pos to refer to the path assigned to an entry.

Operations on the ORAM tree. For the time being, we make a simplifying assumption and assume that all entries in the logical multimap has *distinct* keys. *Later on in Section 3.2, we shall discuss how to remove this assumption.* We may assume that each entry is of the form $(k, \mathsf{data}, \mathsf{pos})$, where k denotes the key, data is a payload string, and pos is the position identifier of the entry. The oblivious AVL tree application requires a specific format for the data field which we shall elaborate on in Section 3.2.

There are three types of operations on the ORAM tree:

- $\mathsf{ReadRm}(k, \mathsf{pos})$: Given a key k and a position identifier pos, read the path identified by pos. If an entry with the key k is found, remove the entry from the path and return the fetched entry.
- $\mathsf{Add}(k, \mathsf{data}, \mathsf{pos})$: Given a key k, some payload string data, and a new position identifier pos, add the entry $(k, \mathsf{data}, \mathsf{pos})$ to the root bucket.
- $\mathsf{Evict}(\mathsf{pos})$: Given a path identified by pos, perform an eviction operation on the path (including the stash). An eviction operation re-arranges the entries on the path such that they are packed as close to the leaf level as possible, while still respecting the path invariant.

If we want to read or write an entry identified by k in the ORAM tree, we first need to find out its position identifier pos — we will describe how to achieve this in an oblivious AVL tree in Section 3.2. Once we know both k and pos, we perform the following:

1) first call $\mathsf{data} \leftarrow \mathsf{ReadRm}(k, \mathsf{pos})$;
2) next call $\mathsf{Add}(k, \mathsf{data}', \mathsf{pos}')$ where data' is the new data to overwrite with or the same as the returned data if no update is needed, and pos' denotes a randomly selected new path;

3) next call $\mathsf{Evict}(\mathsf{pos})$ where pos is the path which we have just read from.

3.2. Oblivious AVL Tree

A multimap data structure can be realized with an AVL tree, and our goal is to make the AVL tree obliviousness by relying on the ORAM tree data structure introduced in Section 3.1. We first give some background on the AVL tree. An AVL tree is a balanced binary search tree [8]. The logical data structure, henceforth referred to as the *AVL tree* or the *logical tree*, imposes a balancing invariant: each node's left subtree and right subtree can have a height difference of at most 1. As such, the maximum depth of an AVL tree with N nodes is $1.44 \log_2 N$ [8], [12].

We want to make the AVL tree oblivious, and we rely the oblivious data structure techniques of Wang et al. [12]. The idea is to store each entry (i.e., node) of the AVL tree inside a single physical ORAM tree. Henceforth, we refer to a pair $\mathsf{ptr} := (k, \mathsf{pos})$ as a *pointer*, which contains a key k and a position identifier — as mentioned, for the time being, we assume that each key in the multimap data structure is distinct and we later remove this assumption in Section 3.2. Each entry (i.e., node) of the AVL tree, stored in the physical ORAM tree, is of the following format:

$$(\mathsf{ptr}, \mathsf{lptr}, \mathsf{rptr}, v)$$

where v denotes the value, and the fields ptr, lptr, and rptr denote the pointers of the current node, its left child, and its right child in the logical AVL tree. If a node does not have a left or right child, the corresponding lptr or rptr is $\perp$.

Supporting the AVL tree operations. We now describe how to support the AVL tree operations when the AVL tree nodes are stored in an ORAM tree. Henceforth, we may assume that the entry corresponding to the *root* of the AVL tree is always stored at a fixed position. We first provide an explanation ignoring the issue of padding, and then we explain the padding that is necessary for hiding the lengths of each AVL tree path and ensuring obliviousness.

$\mathsf{Find}(k)$ starts from the root node (stored at a fixed position) and walks down a logical path in the AVL tree. To fetch each node in the AVL tree, we need to read a path in the ORAM tree. The key is how to discover its position identifier. This is easy due to the way that each entry stores the position identifiers of its two children. In this way, once we find the parent node, we immediately learn the keys and position identifiers of the two children. Now, depending on whether the logical path wants to go left or right, we can look up the corresponding path in the ORAM tree (i.e., ReadRm), and find the next node along the logical path. After we call ReadRm for any entry looked up, we assign it a random new path and add it back by calling Add, and then perform an eviction by calling $\mathsf{Evict}(\mathsf{pos})$ on the read path identified by pos.

We now describe how to perform $\mathsf{Insert}(k, \mathsf{data})$. Here we use the notation data to denote the entire payload string of an entry besides the key, where the format of each entry was explained above. To insert an element with the key k, we first perform a lookup for the key k, which walks down some path in the logical AVL tree. This identifies the right position in the logical tree to insert the new entry. After inserting the entry into the logical tree, we perform a rebalancing operation to maintain the balancing invariant of the AVL tree. The AVL tree's rebalancing operation touches exactly the same path that was looked up. These nodes can be fetched using the same way as in Find — recall that whenever we find a parent, we immediately know the position identifiers of both its children. Rebalancing might modify some of the nodes' parent-children relationships. During the rebalancing operation, we assign a new random path to all entries that are touched during by the rebalancing. Each node modifies its children's keys (if needed) and position identifiers. Now, all of the modified entries will be added back to the root bucket, and we perform eviction on every path that was involved during the ReadRm phase.

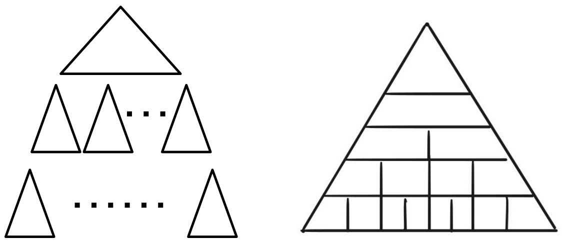
$\mathsf{Delete}(k)$ can be supported in a similar manner as the insertion, since it also walks down a logical path, and then performs rebalancing involving the logical path just looked up, as well as the sibling of each node on the path.

To realize the full multimap, we additionally have to support $\mathsf{Size}(k)$ and the more general version $\mathsf{Find}(k, i, j)$ which take the multiplicity of each key into account, as well as $\mathsf{Init}(\mathbf{I})$ — we will describe how to support these operations in Section 3.2.

Padding. In the logical AVL tree, each path may have different length, and the maximum length is $1.44 \log N$. Recall that we store each node of the AVL tree in an ORAM tree. To make the scheme fully oblivious, we need to hide the length of the AVL tree paths visited. Therefore, we always pad the number of requests to the ORAM tree to some worst-case amount for every operation.

Supporting key multiplicity. So far, we have assumed that all keys are distinct. In our final multimap abstraction, there may be multiple entries sharing the same key. We can easily support key multiplicity using standard techniques described by Cormen et al. [58] — the same techniques were adopted by Oblix [10]. We need to following modifications to AVL-tree nodes: 1) Make the key of each AVL node now be a triplet `(original_key,value,uid)` to ensure key uniqueness where `uid` is a unique identifier (e.g., a monotonic counter that counts the total number of insertions when the entry is first inserted); and 2) add a counter field to each AVL node that counts how many nodes with the same original_key are on the left and right subtrees of that node. The former modification makes sure that the new keys of all AVL tree nodes are distinct. The latter modification makes it possible to efficiently search for the i-th to the j-th occurrences of some specified key; moreover, it allows us to efficiently support the $\mathsf{Size}(k)$ operation. Using standard techniques, we can easily modify the algorithm to always maintain correctness of the counter fields during the insertion and rotation processes.

Recall that in our multimap definition earlier in Section 2.2, we require that entries with the same key be sorted



(a) ENIGMAP's locality-friendly layout

(b) Oblix's heap layout

Figure 2: Layout of ORAM tree in external memory. Each page contains roughly *the same number of nodes* (although their area does may not appear equal in the drawing).

by their value field. In many practical applications, we need not sort entries with the same key by their values. For example, it may also be ok to sort entries of the same key by the time of insertion. In this case, we can make the new key simply `(original_key,uid)`. In this way, we can store the value field in a separate data ORAM as an optimization when the value field is large in size (see Appendix B.2).

4. ENIGMAP: Algorithmic Improvements

4.1. Locality-Friendly ORAM Tree Layout

As mentioned, for secure enclave programs, the primary performance metric should be the number of page swaps in and out of the enclave (also called the I/O cost). We now describe a locality-friendly layout that allows us to accomplish each AVL-tree operation with only $O(\log N \cdot \log_B N)$ page swaps where B denotes the page size. In comparison, earlier works such as Oblix [10] and ObliDB [13] incur $O(\log^2 N)$ page swaps, and thus we achieve both asymptotical and concrete improvement over prior works.

Our locality-friendly layout adopts an elegant idea that originally comes from the algorithms community [30], [31]. Recall that in our oblivious AVL tree algorithm, all logical operations eventually boil down to accessing paths in the ORAM tree. Our goal is to minimize the page swaps necessary whenever a path in the ORAM tree needs to be visited. The most naïve way to pack the ORAM tree in physical memory is to use a standard heap layout: i.e., simply write the root node first, then its two children, then the four nodes at the next level of the tree, and so on (see Figure 2b). However, this approach requires $O(\log N)$ page swaps for accessing a path. By contrast, in ENIGMAP, we pack each subtree (represented by a triangle in Figure 2a) of depth $L = \lfloor \log_2 B \rfloor$ into a memory page of size B. This way, accessing a path incurs only $O(\log_B N)$ page swaps, which is asymptotically better than the naïve scheme above.

An alternative but slightly different approach is to use the standard Emde Boas layout [30], [31]. The Emde Boas layout relies on a clever recursion to pack the tree nodes into memory pages, with the advantage that the algorithm is *cache-agnostic* [30], [31], i.e., it need not know the page size B and the enclave's resident memory size M. However, to achieve the cache-agnostic property, the price is a factor of up to $2\times$ blowup in performance. This up to $2\times$ blowup comes from two main factors 1) Emde Boas's recursion may not stop at the concretely optimal choice of L; and 2) if there is some remainder empty space in a page after packing a triangle, the Emde Boas layout would start to pack the next triangle into this remaining space. While this saves space by a factor of at most 2, it may increase the number of page swaps by a small constant factor in practice.

Fortunately, in an enclave setting, we know the exact page size B and the enclave resident memory size M. Therefore, it is better to use a cache-aware (as opposed to cache-agnostic) memory layout as shown in Figure 2a, which saves up to $2\times$ factor in the number of page swaps in comparison with the standard Emde Boas layout.

4.2. Efficient Initialization Algorithm

We describe a new initialization algorithm that achieves significant asymptotical as well as concrete improvement over prior works such as Oblix [10]. Our initialization algorithm can be accomplished with $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps and $O(N \log N)$ runtime. In comparison, the initialization algorithm of Oblix [10] incurs $O(N \log^3 N)$ runtime and $O(N \log^3 N)$ page swaps[1].

The task of initialization is the following: we are given an initial array $\mathbf{I}$ containing (k, v) pairs stored in memory. We want to obliviously initialize a data structure that would support the Size, Find, Insert and Delete operations mentioned in Section 2.2.

Our new initialization algorithm proceeds in two stages:

- *Stage 1:* Stage 1 aims to accomplish the following: 1) each entry picks a random position identifier in the ORAM tree; 2) assign all entries to a unique node in an AVL-tree; and 3) each parent in the AVL-tree learns the keys and position identifiers of its two children.
- *Stage 2:* By the end of stage 1, we have created all entries of the ORAM tree, and assigned them random position identifiers. The goal of stage 2 is to insert all these entries into the ORAM tree while packing them as close to the leaf level as possible. At the end of stage 2, we should output the ORAM tree in memory using the locality-friendly layout mentioned in Section 4.1.

4.2.1. Algorithm for Stage 1.

We devise the following algorithm to accomplish stage 1:

1) Sort the initial array $\mathbf{I}$ in increasing order of the key field k, and let the resulting array be $\mathbf{X}$. Since the initial database $\mathbf{I}$ is not secret, we can use a non-oblivious, external-memory sorting algorithm. We recommend using a multi-way merge sort which achieves $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps and $O(N \log N)$ runtime.

1. It is possible to turn it into $O(N \log N)$ page swaps by using an external memory optimized sort.

At this moment, we will encrypt the sorted array. Henceforth, although not explicitly noted, all memory is assumed to be encrypted.

2) Each entry in the sorted array $\mathbf{X}$ picks a random position identifier for itself. Henceforth, we regard the sorted array $\mathbf{X}$ as the in-order traversal of the logical AVL-tree. Given each entry in position i of the sorted array $\mathbf{X}$, we use $\mathsf{left}(i)$ and $\mathsf{right}(i)$ to denote the indices of its two children.
3) Let root be the index of the root node of the AVL-tree. Call the following recursive algorithm $\mathsf{Propagate}(\mathsf{root})$ such that each parent learns the keys and position identifiers of its two children.

```
Propagate(r)
a) let lptr = Propagate(left(r)),
b) let rptr = Propagate(right(r)),
c) Store (lptr, rptr) to X[r],
d) Return the key and position identifier in X[r].
```

Performance bounds. We now analyze the performance of the above algorithm. The sorting step takes $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ number of page swaps and $O(N\log N)$ runtime. Encrypting the entire memory array and assigning a random position identifier to each element in the array takes only $O(N/B)$ page swaps and $O(N)$ runtime. For the third propagation step, clearly its runtime is $O(N)$. Its number of page swaps $Q(N)$ can be analyzed through the following recurrence:

$$Q(n) = \begin{cases} 2Q(n/2)+1 & \text{if } n > B \\ 2 & \text{o.w.} \end{cases}$$

Thus, we conclude that the propagation step consumes at most $O(N/B)$ page swaps.

In summary, stage 1 costs $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ number of page swaps and $O(N\log N)$ runtime.

4.2.2. Algorithm for Stage 2 (Warmup).

By the end of stage 1, we have prepared all entries to be inserted into the ORAM tree, and each entry has been assigned a random position identifier. Our goal is to place these entries into the ORAM tree, and pack them as close to the leaf as possible.

To achieve this, we compute the contents of each level of the ORAM tree one by one, starting from the leaf level. Initially, each level is stored in a contiguous memory region. At the end of the algorithm, we need to convert the layout to the locality-friendly layout mentioned in Section 4.1.

We shall employ an oblivious bin placement algorithm denoted BinPlace which obliviously places the real elements in an input array into bins, and outputs the output bins as well as a remainder array containing all overflowing elements — we review oblivious bin placement in Appendix A. Our stage 2 algorithm is described below where level $\log_2 N$ denotes the leaf level, and level 0 denotes the root level:

```
1) Let $\mathbf{Y}$ be the array output by stage 1, which contains all $N$ entries to be inserted into the ORAM tree, and each entry stores its own randomly chosen position identifier.
2) For $\ell = \log_2 N, \ldots, \frac{1}{3}\cdot\log_2 N$:
   • scan $\mathbf{Y}$ and mark each (real) entry's destination as the bucket in level $\ell$ of the ORAM tree that it can reside in;
   • $\mathsf{TreeLevel}[\ell], \mathbf{Y} \leftarrow \mathsf{BinPlace}(\mathbf{Y})$ where BinPlace is parametrized with the bin size $Z = 4$, and the total number of bins $m = 2^\ell$;
   • truncate $\mathbf{Y}$ and preserve only the first half.
3) For each level $\ell = \frac{1}{3}\cdot\log_2 N - 1, \ldots, 0$, for each bucket in level $\ell$, for each slot in the bucket:
     linearly scan through $\mathbf{Y}$ and and fill the slot with an entry that can reside in the bucket, replace the chosen entry with a filler in $\mathbf{Y}$.
4) Change the tree's layout from level-by-level layout to the locality-friendly layout described in Section 4.1.
```

The above stage 2 algorithm incurs $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ page swaps and $O(N\log N)$ runtime. We defer its correctness and performance analysis to Appendix A.

Remark 4.1. *Oblix's initialization algorithm [10] is similar to our warmup algorithm. They used vanilla bitonic sort that is not optimized for the number of page swaps, and their paper states their algorithm's runtime to be $O(N\log^3 N)$. Oblix also did not evaluate their initialization algorithm.*

4.2.3. Improved Algorithm for Stage 2.

We describe how to employ techniques from Ramachandran and Shi [19] in a *non-blackbox* manner to simplify the algorithm (in a practical implementation) and improve its concrete performance by a constant factor.

In stage 2, roughly speaking, we want to sort elements into the tree nodes they are destined for. Our key observation is that the elements' destinations are *randomly chosen*.

Building block: oblivious random bin assignment. We will leverage the *oblivious random bin assignment (ORBA)* algorithm which is a building block in Ramachandran and Shi's oblivious sorting algorithm [19]. Let $Z = \omega(\log N)$ and suppose we have n/Z bins each of capacity $2Z$. Given an input array with a total of n real or filler elements, suppose that each real element chooses a random bin as a destination. An ORBA algorithm allows us to route the real elements into their destination bins without revealing their destinations. If a bin receives fewer than $2Z$ elements, it will be padded with fillers to a capacity of $2Z$. The probability that each destination bin receives more than $2Z$ elements is negligibly small in N as long as $Z = \omega(\log N)$.

Improved algorithm for stage 2. We will assume that the enclave's resident memory size $M = \omega(\log N)$ which is a standard "tall-cache" assumption and is also true in practice.

1) Let $Z = M/C$ for a sufficiently large constant $C > 1$, and let ℓ^* be a level in the tree with N/Z nodes, and let $T_1, \ldots, T_{N/Z}$ be the subtrees with roots in level ℓ^*.

Imagine that each subtree is associated a super-bin of capacity $2Z$, and each element's position identifier determines which super-bin it wants to go to. Use ORBA to route all real entries into their destined super-bins.

2) For $i = 1$ to N/Z, do the following. Fetch the i-th super-bin into the enclave. We know that this super-bin should be packed into subtree T_i, and some elements may be leftover afterwards. We can accomplish this packing through invoking an oblivious bin placement algorithm at each layer (unlike the earlier stage 2 algorithm, we do not truncate the remainder array at each layer). Since the entire subtree and the super-bin fits within the enclave's memory, we can compute the subtree T_i and the leftover elements within the enclave.
Let $R_1, \ldots, R_{N/Z}$ be the leftover arrays at the end of this step, one for each subtree, and each array has size $Z' = \omega(\log N) \leq Z$ except with negligible in N probability. We can make sure that $R_1, \ldots, R_{N/Z}$ are stored in a contiguous region in external memory.
3) Now, for layer $j = \ell^* - 1$ down to the root level, we will compute layer j of the tree in the following manner:
   - Let k be the number of tree nodes in the current layer, this also means that we start with exactly $2k$ leftover arrays. Group the leftover arrays into k pairs such that each array is paired with its neighbor.
   - The analysis in Stefanov et al. [6] implies that the total number of real elements in each pair is upper bounded by Z' except with negligible probability. We now merge each pair and sort all the real elements to the front in the merged array. We truncate each merged array from the end to a size of Z'.
   - Let $R_1, \ldots, R_k$ be the k merged arrays. The buckets in the current tree level are defined as $R_1[1 : 4], \ldots, R_k[1 : 4]$.
   - Replace the first four elements of each $R_1, \ldots, R_k$ with fillers, and the resulting arrays are the new leftover arrays to be input to the next iteration. If this is the root level, the singleton leftover array is the stash.
4) Finally, use the same approach as the earlier stage 2 algorithm to transform all levels j from ℓ^* down to the root to a locality-friendly layout.

Performance bounds. Ramachandran and Shi [19] suggest an ORBA algorithm that achieves $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ page swaps and $O(N \log N)$ runtime. Step 1 takes only one ORBA invocation on $O(N)$ elements. Step 2 can be accomplished with $O(N/B)$ page swaps and $O(N(\log\log N)^3)$ runtime if we use bitonic sort. In Step 3, for a level of the tree with k nodes, we consume $O(k \cdot (\log\log N)^2)$ runtime and $O(kZ'/B)$ page swaps. Since even for the largest level, $k = N/Z$, the total number of page swaps is at most $O(N/B)$. Step 4 takes at most $O(N/B)$ page swaps and $O(N)$ runtime due to the same analysis as before.

Summarizing all steps, the total page swaps is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ and the total runtime is $O(N \log N)$. We note that when we instantiate Ramachandran and Shi's ORBA algorithm, if we use bitonic sort to sort poly-logarithmically sized arrays, then there will be an extra $\log\log N$ factor in the runtime, but the number of page swaps is unaffected.

5. EnigMap: Architecture and Practical Optimizations

5.1. Secure Multi-Level Caching

5.1.1. Caches for Physical Storage. EnigMap relies on multiple levels of caching to speed up performance.

- *Page-level cache outside the enclave.* Outside the secure enclave, we implement a page-level LRU cache that stores the most recently used, software-encrypted memory pages, to reduce the disk I/O.
- *Bucket-level cache inside the enclave.* Inside the enclave, we implement a bucket-level LRU cache that stores the most recently used buckets (of the ORAM tree) to reduce the number of page swaps in and out of the enclave.

Both of these caches are caching physical accesses — since physical accesses are already made secure by our oblivious algorithms, the caches here do not leak any information.

Observe that an LRU cache is a simple method for approximating a "tree-top" cache. Since every access to the ORAM tree always accesses the root bucket, and will more likely access buckets near the root, earlier works in this space have suggested the idea of *tree-top* caching [23], [25], [26], i.e., caching a small number of levels near the root.

5.1.2. AVL-Level Cache. Observe that to insert an element into the AVL tree, we first access a path in the AVL tree to find where to insert. We then access exactly the same path to perform rebalancing. Recall that accessing each node in the AVL tree translates to accessing a path in the ORAM data structure; thus accessing a path in the AVL tree translates to accessing $O(\log N)$ paths in the ORAM tree. Now, since the second pass touches exactly the same AVL tree nodes as the first pass, we would like to save these nodes in a cache such that during the second pass, we need not get them again from the ORAM data structure.

The most naïve approach is to cache all $O(\log N)$ ORAM tree paths during the first pass. During the second pass, we need not make additional accesses to the ORAM data structure. After the second pass, we make $O(\log N)$ evictions altogether — roughly speaking, this is the approach taken by Oblix [10]. The drawback is that we will need a $O(\log^2 N)$-sized cache, which is costly in terms of enclave memory. Recall that the enclave has limited resident memory, thus the AVL-level cache is competing with other caches such as the bucket-level cache mentioned earlier.

Our approach: sticky entries. We use a different approach called *sticky entries*. During the first pass, we fetch $O(\log N)$ ORAM-tree paths. Each time we fetch an ORAM-tree path,

1) We mark the entry of interest (i.e., the entry that stores the AVL-tree node that we care about) as "sticky" in the ORAM's stash;

2) We then perform eviction on the read path, however, sticky entries are pinned on the stash and will not get evicted.

Now, during the second pass, we simply fetch all the AVL-tree nodes needed directly from the stash without having to make any access request to the ORAM data structure. More concretely, we make a linear scan over the stash for every AVL-tree node needed during the second pass[2]. At this moment, we also remove the sticky marks from the relevant entries in the stash so they can get evicted in the future.

Thus, using sticky entries, we effectively implemented an AVL-tree level cache. Below, we argue the following: *i)* the ORAM's stash size will be upper bounded by $O(\log N + R)$ with probability $1 - \exp(-\Omega(R))$, a significant improvement over the $O(\log^2 N)$-sized cache of the earlier naïve approach; and *ii)* our "sticky entries" cache does not affect security.

Batched eviction and stash size bound. Using this approach, essentially, each time we add $k = O(\log N)$ elements to the stash, we perform the same number of evictions. This approach is equivalent to "batched evictions" in earlier works on Oblivious Parallel RAM [59] — however earlier works used batched evictions for a different purpose than us. Furthermore, earlier works [59] showed a stochastic domination result: the number of blocks in heights $\log_2(2k)$ or smaller of the ORAM tree are stochastically dominated by non-batched eviction — this stochastic domination result holds also for Path ORAM. Thus, we can reuse the same stochastic analysis as Path ORAM [6], and we conclude that the stash does not exceed $O(\log N + R)$ except with $\exp(-\Omega(R))$ probability.

Security of sticky entries. Recall that earlier bucket-level and page-level caches are caches for *physical storage* — in this case security is automatically guaranteed by the obliviousness of the algorithm. By contrast, the sticky entries implement a cache for the logical AVL-tree. It is not hard to see that this optimization does not break security, since the fact that the second pass of the AVL-tree's Insert algorithm touches the same path as the first pass is publicly known.

5.2. Ensuring Strong Obliviousness

As mentioned, to provably defend against cache-timing attacks, we want the algorithm to have memory-trace obliviousness even within the enclave. Earlier works have referred to this notion as *strongly oblivious* [18], [19] or *doubly oblivious* [10]. To achieve strong obliviousness, we make sure that 1) the data accesses are oblivious even within the enclave; and 2) the instruction traces are oblivious.

2. One optimization here is to obliviously sort the stash after the first pass, and move all the sticky entries to the front. This way, we only need to scan through the first $1.44 \log_2 N$ entries of the stash to find each sticky entry — this optimization, however, does not matter too much to the concrete performance since most of the *computation* overhead comes from the ORAM-tree's eviction algorithm.

5.2.1. Data Obliviousness Within the Enclave.

To ensure that the data trace is oblivious within the enclave, we rely on 3 oblivious sorts on the path (including the stash) to implement the Evict algorithm of Path ORAM. The algorithm was described in the earlier work of Wang et al. [20] (see Figure 2 in their paper), although they employ this idea for a different setting: they want an ORAM suitable for secure computation, and therefore they use the same idea to convert the ORAM's eviction algorithm to a circuit.

Combining the oblivious-sort-based eviction algorithm and our locality-friendly ORAM tree layout, a ReadRm and an Evict operation along an ORAM-tree path costs $O(\log_B N)$ number of page swaps and $\widetilde{O}(\log N)$ runtime, where $\widetilde{O}(\cdot)$ hides $(\log \log N)^2$ factors (assuming that Bitonic sort [60] is used as the oblivious sorting algorithm). For both metrics, we achieve asymptotical improvement over Oblix [10]: Oblix's path read and eviction algorithm incurs $O(\log N)$ page swaps and $\Omega(\log^2 N)$ runtime. This is because they run a double loop on the metadata of the path, and they do not use the locality-friendly layout.

5.2.2. Instruction-Trace Obliviousness.

We use standard techniques for ensuring instruction-trace obliviousness [32], [33]. If there is a secret conditional (e.g., a `if` instruction with a condition that depends on a secret variable), we must ensure that the two branches have the same memory trace, including both the data trace and the instruction trace — this also implies that the *length* of these traces must also be identical for both branches. For example, in the program below, we have an `if` statement conditioning on a secret variable X.

```
if (X) { B = C; }          |||  CMOV( X,B,C);
else   { D = E; }          |||  CMOV(!X,D,E);
```

To ensure that it is instruction-trace obliviousness, we implement it as two `CMOV` instructions. A `CMOV(X, U, V)` instruction checks whether the bit `X` is set. If so, it assigns the register V to the register U, and else it does nothing.

Function calls inside secret branches are a more complex problem. We use the phantom function call idea from prior work [32]. Specifically, we add an extra "phantom flag" to the relevant function calls. If the phantom flag is set, the function call would incur the same memory trace but effectively do nothing and cause no side effect. Whenever convenient, we simply hoist function calls outside secret `if`s to avoid this issue.

Example. As a concrete example, Figure 3 is Oblix [10]'s code for searching the AVL tree. Their implementation is not instruction-trace oblivious for several reasons:

1) Lines 7 and 8 check if the current key is what we are searching for, and if so, immediately exit the `while` loop. This leaks information about the structure of the AVL tree, and thus breaks instruction-trace obliviousness.
2) Lines 7-14 are a secret `if` statement, however, different branches incur different instruction traces.

In our implementation of the same algorithm, we always make $1.44 \log_2 N$ ORAM accesses regardless of when we

```
fn find_helper(avl_key: &AVLKey<K> /.../) //...
{
 let mut root_key = root_key.clone();
 while let Some(r_key) = root_key {
  let cur_node = self.ods_ref.borrow_mut().
    access(Read(ActualOp, &r_key), server)? //
    ...
  if cur_node.key() == avl_key {
   return Ok(Some(cur_node));
  } else if avl_key < cur_node.key() {
   root_key = cur_node.left_key();
  } else {
   root_key = cur_node.right_key();
  }
 }
 return Ok(None);
}
```

Figure 3: Oblix's AVL tree search code violates instruction-trace obliviousness.

actually find the key. Further, we use the CMOV trick mentioned earlier to ensure that the instruction traces are always identical for all branches of secret ifs.

5.3. Additional Concrete Optimizations

We perform some other optimizations that would lead to constant-factor savings in practice. In the interest of space, we discuss them in Appendix B.

6. Implementation and Experimental Results

6.1. Implementation and Evaluation Methodology

Implementation. We implemented our oblivious multimap as an extensible library `enigmap_lib` that can easily be integrated with any enclave framework. The library `enigmap_lib` consists of 5000 lines of C++ code (3500 lines of code, 1500 lines of tests). In our experiments, we integrated our library `enigmap_lib` with the Intel SGX SDK — integration takes less than 100 lines of C++ code plus 10 lines of edl definitions.

Open source. All of our code has been open sourced and is publicly available at https://github.com/odslib/EnigMap.

Experimental setup and baselines of comparison. All the experiments (for ENIGMAP and Signal's implementation) were run on the same desktop machine with an AMD 5900X processor, running at 4.2Ghz, in SGX simulation mode. Our machine is equipped with 64 GB of RAM and a Samsung 970 EVO M.2 SSD drive. In our experiments, the key and value sizes are both 64 bits.

We compare ENIGMAP with the following baselines:

- **Signal's private contact discovery**. We downloaded Signal's private contact discovery implementation [9], which uses (batched) linear scans to answer queries.
- **Oblix.** Oblix's code is not open source, but we were able to obtain a copy of their code from the authors of Oblix [10]. We were not able to compile their code though, since their code is compatible with only certain versions of Rust packages. We could not find any information regarding which version to use.
  Fortunately, we are still able to compare with Oblix despite not being able to compile their code. In the Oblix paper, they reported the speedup of Oblix over Signal's implementation running on the same machine (i.e., their machine). By comparing our relative speedup with their reported relative speedup both over Signal's implementation, we are also able to compare with Oblix.

Besides Oblix [10], other implementations of oblivious multimaps exist, e.g., ObliDB [13], which implements a more general oblivious database, can also be used as an oblivious multimap. However, as acknowledged in the ObliDB paper [13], their oblivious multimap performance is not as good as the Oblix implementation since ObliDB is geared towards general database queries. Therefore, we conclude that Oblix and Signal's implementation are the state-of-the-art baselines of comparison.

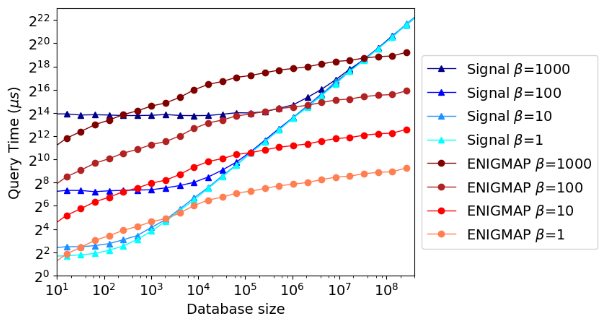


Figure 4: Comparison of ENIGMAP and Signal

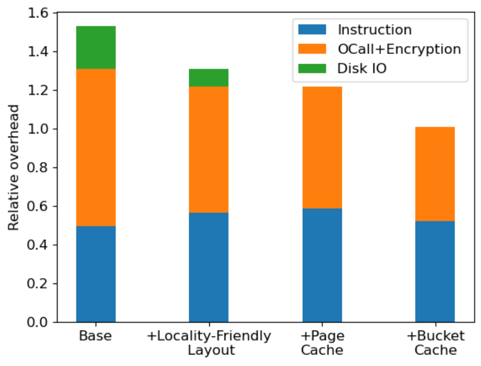


Figure 5: Cost breakdown and the effects of several optimizations for $N = 2^{20}$.

6.2. Experimental Results

Should Signal switch to using oblivious algorithms? Figure 4 compares ENIGMAP's search performance against Signal. Signal can support a batch of queries through through a single linear scan of the database. To support a batch of β

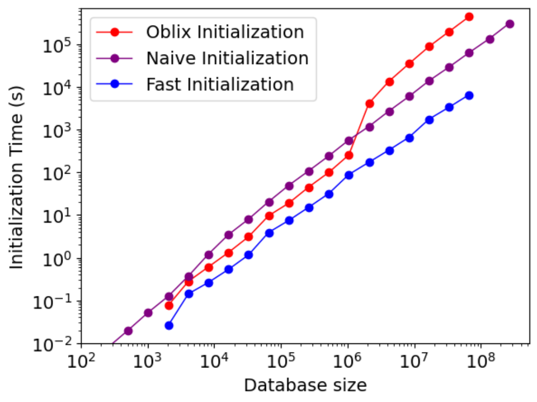


Figure 6: Initialization cost of ENIGMAP

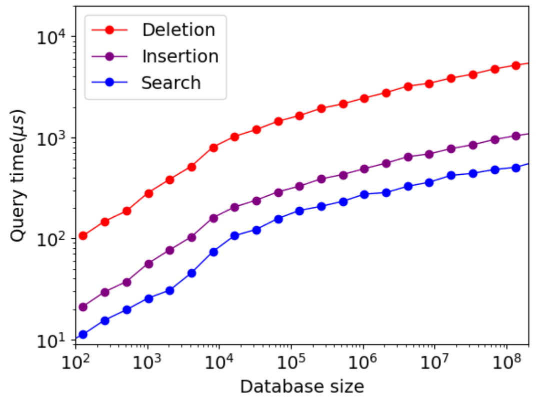


Figure 7: ENIGMAP: cost of different operations

queries, their algorithm incurs $O(\beta^2 + N)$ runtime [10] and $O(N/B)$ page swaps. In our implementation, we process the requests in the batch one by one.

Signal's number of monthly active users increased from 20 million at the end of 2020, to 40 million in 2021 (see also Figure 9 of Appendix 9). The had more than 105 million downloads as of May, 2021, and more than 50 million installs on Android devices [34], [35]. In Signal's blog post [2], they stated that they want to support billion-sized databases. Therefore, we conclude that Signal has between 40 million to billions of registered users.

Our experiments show that at a database size of 2^{26} (i.e., 67 million), our speedup over Signal is $1750\times$, $170\times$, $20\times$, $2\times$, at a batch size of 1, 10, 100, 1000, respectively. **At a database size of 2^{28} (i.e., 256 million), our speedup over Signal is 5500$\times$, 530$\times$, 53$\times$, 5.5**$\times$ at a batch size of 1, 10, 100, 1000, respectively. At a batch size of 1, 100, and 1000, ENIGMAP starts to outperform Signal at a database size of 2^{11}, 2^{21}, and 2^{25}, respectively.

Although the prior work Oblix [10] considered batch sizes of 1, 10, 100, 1000 in their evaluation, we observe that Signal's implementation actually supports a maximum batch size of $\beta = 8192$. Even at $\beta = 8192$, our experiments show that ENIGMAP starts to outperform Signal when the database size is more than 256 million.

We conclude that there are convincing reasons why, given Signal's rapidly growing user base, *they should switch an oblivious algorithms for realistic batch sizes*.

Comparison with Oblix. The Oblix paper [10] reported a speedup of $128\times$ over Signal for a database size of 2^{26} at a batch size of 1. Comparing our speedup (reported earlier) and their speedup over Signal, we achieve a relative **speedup of 14**$\times$ **over Oblix** at a database size of 2^{26}, and a relative **speedup of** $21\times$ **over Oblix** at a database size of 2^{28}.

For any database our minimum speedup over Oblix is 6 at a database size of 2^{14}, with an increasing speedup as N increases due to the difference in asymptotics.

Cost breakdown. Figure 5 shows the breakdown of our costs into three categories: 1) computational overhead, 2) the cost of page swaps (including encryption/decryption and OCall overhead); and 3) disk I/O. In this figure, we use a database size of 2^{20}.

The rightmost bar is with all of our optimizations turned on, whereas the leftmost bar named "Base" is without any optimization. The y-axis plots the relative slowdown over the rightmost bar. From the leftmost bar to the rightmost bar, we turn on the following optimizations one by one, which shows the effect of these optimizations: 1) locality-friendly layout; 2) page-level caching; and 3) bucket-level caching. The figure confirms that absent our optimizations, the majority of the overhead comes from the page swap and disk I/O overhead. These optimizations together significantly reduce the page swap and disk I/O overhead, such that the total performance is improved by a $1.5\times$ factor. At a database size of 2^{20}, our final construction outperforms Oblix by $8\times$. This means that *even our unoptimized base performance is roughly* $5\times$ *to* $6\times$ *faster than Oblix*. This is because even without the optimizations and the locality-friendly layout, the runtime overhead of our algorithm is asymptotically better than Oblix's implementation (see Table 1). Another interesting observation is that with all the optimizations turned on, the page swap and disk I/O overhead is somewhat even with the computational overhead. Finally, the runtime in the base version looks slightly less than the rest, and this is due to the side effects of running the profiler.

Initialization cost. Figure 6 shows initialization cost of ENIGMAP in comparion with the following two baselines:

1) Using ENIGMAP's insertion algorithm to insert the entries one by one (called "naïve" in the figure); and
2) Oblix's initialization algorithm (we implemented their initialization algorithm for comparison).

At a database of size 2^{26}, our initialization algorithm is $10\times$ faster than our own naïve initialization algorithm, and even our naïve algorithm performs $5\times$ faster than Oblix's algorithm. At a database size of roughly one million, Oblix's initialization algorithm starts to perform even worse than our naïve initialization — this is the moment when the database does not fit in enclave memory, and Oblix's initialization algorithm is not efficient in terms of page swaps.

Cost for different operations. Figure 7 shows the cost for different operations w.r.t. the database size. As we can see, insertion is about $1.5\times$ to $2\times$ more expensive than search, and deletion is $5\times$ more expensive than insertion. This is because insertion needs to perform rebalancing in one node, and deletion needs to perform more rebalancing than insertion.

References

[1] Whatsapp legal. https://www.whatsapp.com/legal/.

[2] Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/, 2017.

[3] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

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

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

[6] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 65(4), April 2018.

[7] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.

[8] Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, 1962.

[9] Signal's private contact discovery open-source implementation. https://github.com/signalapp/ContactDiscoveryService/.

[10] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296. IEEE, 2018.

[11] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[12] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[13] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2), 2019.

[14] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, New York, NY, USA, 2009. ACM.

[15] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[16] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *CCS*, 2014.

[17] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

[18] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, 2018.

[19] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 373–384, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. In *ACM CCS*, 2014.

[21] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 655–671. ACM, 2021.

[22] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 727–743, 2018.

[23] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[24] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[25] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

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

[27] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[28] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[29] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[30] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[31] Harald Prokop. Cache-oblivious algorithms. Master thesis, MIT, 1999.

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

[33] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. CSF '13, pages 51–65, 2013.

[34] Signal (software). https://en.wikipedia.org/wiki/Signal_(software).

[35] Khushi Agrawal. Signal statistics: Usage, revenue, & key facts. https://www.feedough.com/signal-statistics-usage-revenue-key-facts/.

[36] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

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

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

[39] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

[40] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[41] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[42] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015.

[43] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 431–446. USENIX Association, 2015.

[44] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. *Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks*, page 1263–1276. 2020.

[45] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to Large-Scale data in the data center. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[46] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, 2005.

[47] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, 2011.

[48] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[49] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[50] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018.

[51] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, 2018.

[52] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.

[53] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *Commun. ACM*, jun 2020.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, August 2018.

[55] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[56] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, 2019.

[57] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, 2015.

[58] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2nd edition, 2001.

[59] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC (2)*, volume 10678 of *Lecture Notes in Computer Science*, pages 72–107. Springer, 2017.

[60] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings*, pages 307–314, 1968.

[61] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Asiacrypt*, 2017.

Appendix A. Deferred Details for the Warmup Initialization Algorithm

In this section, we give more details on the warmup stage 2 algorithm for initializing the data structure.

Preliminary: oblivious bin placement. Oblivious bin placement [59] solves the following problem. Suppose we are given an input array of length n containing real and filler elements[3]. Each real element in the input array wants to go to some destination bin among a total of m destination bins, and each destination bin has a maximum capacity of Z. We want to output the following two arrays:

- a result array of length $m \cdot Z$ representing the concatenation of all destination bins, where each bin is packed with as many elements destined for it as possible, subject to a maximum capacity of Z. Moreover, any unfilled slots in a destination bin is padded with fillers; and
- a remainder array of length n, which contains the elements that cannot fit into its destined bin, padded with fillers at the end to a length of n.

Chan and Shi [59] showed that the above task can be accomplished with $O(1)$ number of oblivious sorts and linear scans through arrays of length at most $O(n+m \cdot Z)$ (see Appendix A of their paper). Therefore, the algorithm costs $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ number of page swaps and $O(N \log N)$ runtime, if we use an external-memory oblivious sorting algorithm such as the one by Ramachandran and Shi [19]. In practice, we use bitonic sort rather than AKS to sort the poly-logarithmically sized instances and there is an extra $\log \log N$ factor in the runtime.

Correctness. For the warmup stage 2 algorithm to be correct, we need to argue that except with negligibly small probability, it must be that in Step 2, every level $\ell = \log_2 N, \ldots, \frac{1}{3} \cdot \log_2 N$ can successfully pack at least half of the remaining (real) entries. If so, then the truncation

3. We often use n to denote the size of a problem instance, where N represents the size of the database globally.

of the remainder array $\mathbf{Y}$ does not drop any real element. To see this, observe that if we throw n balls into n bins, the expected number of empty bins is n/e for large n. Due to Azuma's inequality, the probability that the number of empty bins exceeds $n/2$ is upper bounded by $\exp(-\Omega(n))$. Observe also that the number of elements that cannot be packed into the buckets in the current tree level is upper bounded by the number of empty bins.

Performance bounds. In Step 2, each level ℓ costs $O(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B})$ page swaps and $O(n\log n)$ runtime where $n = 2^{\ell}$. Step 3 costs $O(n^{1/3}\cdot n^{1/3}/B) = O(\frac{n^{2/3}}{B})$ page swaps and $O(n^{2/3})$ runtime.

It remains to analyze the performance bound for Step 4. Observe that we can have $L = \lfloor\log_2 B\rfloor$ outstanding readers that scan through L levels of the tree, and have a writer that creates the pages (represented by the triangles in Figure 2a) for these L levels along the way. As long as the enclave's resident memory M can fit at least $L+1$ pages, i.e., $M \geq c\cdot B\log_2 B$ for some suitable constant c, then, Step 4 can be accomplished with $O(N/B)$ page swaps and $O(N)$ runtime.

In summary, the entire stage 2 of the algorithm incurs $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ page swaps and $O(N\log N)$ runtime.

Appendix B. Additional Optimizations

B.1. Optimizing the AVL-Tree Insertion Algorithm

In the AVL-tree Insert algorithm, we rely on a new technique that reduces the number of ORAM-tree accesses by a factor of $2\times$ to $3\times$.

In the earlier work of Wang et al. [12] as well as Oblix [10], the AVL-tree insertion algorithm is implemented in a recursive manner. Specifically, after first walking down a path and finding a place to insert, the algorithm now makes a second pass. It starts at the root, and compares the key to be inserted with the current node. Depending on the comparison result, it recursively calls the insertion algorithm on the left or right subtree.

For example, Figure 8 is Oblix's recursive implementation. The issue with this implementation is similar to that of Figure 3, i.e., the implementation is not doubly oblivious. Specifically, the big `if-else` statement in Lines 6-22 conditions on a secret variable. However, depending on which branch is taken, sometimes we perform a recursive call to `insert_helper` and the `balance` operation, but sometimes not. In other words, different branches exhibit different instruction traces. As a result, the total total number of recursive calls also depends on the key that is being inserted.

To fix this problem, one can always make fake recursive calls and *balance* operations as we walk down the path, even when it is actually not needed. However, this would result in a $2\times$ to $3\times$ blowup in runtime (depending on whether we adopt AVL-level caching tricks) since we will need to access two additional sibling nodes for every node along the path — indeed, the work of Wang et al. [12] incurs an extra $3\times$ overhead for this reason.

```
1  fn insert_helper(
2   node: AVLNode,
3   root_key: &Option<AVLKey>
4  ) -> AVLNode
5  {
6   if let &Some(ref r_key) = root_key {
7    let mut cur_node = self.ods_ref
8     .borrow_mut()
9     .access(Read(ActualOp, r_key), server)?
10    .expect("Node␣should␣be␣in␣the␣cache.")
      ;
11   if node.key() < cur_node.key() {
12    child = self.insert_helper(node, &
      cur_node.left_key(), server)?;
13    cur_node.set_left_child(Some(child));
14    server.Write(ActualOp, cur_node);
15    self.balance(cur_node, server)
16   } else if (node.key() > cur_node.key())
      {
17    // same as lines 15-18, but for right
      subtree (...)
18   } else /*...*/
19  } else {
20   server.Write(ActualOp, node.clone());
21   self.root_size += 1;
22   Ok(node.into_child())
23  }
24 }
25
```

Figure 8: Oblix's AVL tree insert code violates instruction-trace obliviousness.

We avoid this constant-factor blowup through the following idea: during the first pass, we not only find where the insertion point is, we also compute all the AVL-tree nodes that will be involved in the rotation operation — there are at most 3 such nodes. At the end of the first pass, we can also compute a rotation plan, including whether a rotation is needed, what type of rotation it is, and the new parent-child relationships of the nodes involved in the rotation. In the second pass, we walk down the same path again. If the node is not involved in the rotation, we make a fake update to its contents; otherwise we make a real update based on the rotation plan we have computed during the first pass. In both passes, we always pad the length of the AVL-tree path to the worst case, i.e., $1.44\log_2 N$ even if the actual length is shorter.

B.2. Data and Metadata ORAM Trees

Recall that our data structure stores key-value pairs. In some applications, the value field can be large. For example, in Signal's application, the key corresponds to a user's ID, and the value field can store the user's record, including

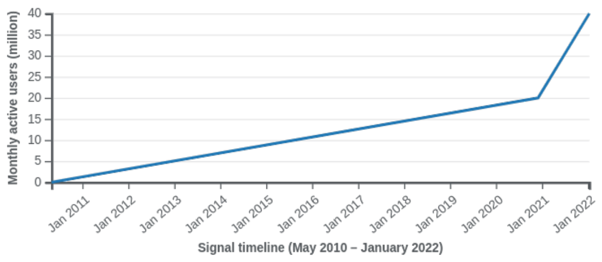


Figure 9: Signal's monthly active users [34]

email, phone number, and location. In case that the value field is large, we can rely on an idea from earlier works [12], and use two separate ORAM trees: one *metadata* tree, and one *data* tree. In the data ORAM tree, we store the value fields of all entries. In the metadata ORAM tree, we store the AVL tree nodes, and each entry is of the following modified format:

$$(\mathsf{ptr}, \mathsf{lptr}, \mathsf{rptr}, \mathsf{vptr})$$

where ptr, lptr, and rptr are defined in the same way as Section 3.2, and vptr stores the the position identifier of the value field in the data ORAM tree. In this way, whenever we read an AVL tree node (i.e., a metadata entry), we know exactly which path to visit to retrieve its value field. Of course, at this moment, the value field in the data ORAM tree will gain a new position identifier, and the corresponding metadata entry is notified of this change. Note that this optimization is applicable when we need not order entries of the same key by the value field (see also the paragraph "supporting key multiplicity" in Section 3.2). In Signal's private contact discovery application, the keys are distinct, so this optimization is applicable.

Appendix C. Supplemental Figure

Figure 9 shows the rapid growth of Signal's active monthly users. Note that the number of actively monthly users is a conservative lower bound on the total number of registered users.

Appendix D. Additional Discussions

Upon receiving a batch of requests, Signal creates a hash table for these requests using a *strong oblivious* algorithm. Due to the need to be strong oblivious, their algorithm for initializing the hash table takes $O(\beta^2)$ time [10]. A more efficient solution is to employ the oblivious two-tier hash table suggested by Chan et al. [61]. Although Chan et al. [61] pointed out that such an oblivious hash table can only support non-recurrent lookups, it is nonetheless safe to use it in Signal's scenario, even if the keys in the database may have multiplicity which seemingly violates the non-recurrent property. This is because in this particular scenario, the database itself is not private and only the user's queries are private.

Unfortunately, just improving the hash table initialization will not help Signal too much, since from Figure 4, we can see that after the database size exceeds roughly one million, Signal's overhead is strictly dominated by the linear scan rather than the hash table initialization.