# ENIGMAP: External-Memory Oblivious Map for Secure Enclaves

*Afonso Tinoco*
*CMU*

*Sixiang Gao*
*CMU*

*Elaine Shi*
*CMU*

## Abstract

Imagine that a privacy-conscious client would like to query a key-value store residing on an untrusted server equipped with a secure processor. To protect the privacy of the client's queries as well as the database, one approach is to implement an *oblivious map* inside a secure enclave. Indeed, earlier works demonstrated numerous applications of an enclaved-based oblivious map, including private contact discovery, key transparency, and secure outsourced databases.

Our work is motivated by the observation that the previous enclave implementations of oblivious algorithms are sub-optimal 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 15× speedup over Signal's linear scan implementation, and 53× 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

An *oblivious* map data structure [1–3] realizes a key-value store while completely hiding the access patterns. Earlier works such as Oblix [2] and ObliDB [3] showed that an enclave-based oblivious map implementation has numerous privacy-preserving applications, such as key transparency, contact discovery, outsourced database. More generally, imagine that a privacy-conscious client would like to issue (key, value) queries to a database hosted by an untrusted server, which is equipped with a trusted processor such as Intel SGX. There are two important privacy goals, both of which are provided by an oblivious map:

1. *Query privacy.* The client should not leak to the server which keys it is querying; and
2. *Database privacy.* Additionally, depending on the application, the database itself may be private, too.

All of the aforementioned applications desire *query privacy*, while *database privacy* is demanded by the secure outsourced database application, but not needed for the key transparency application where the database is the list of public keys. As for privacy contact discovery, database privacy may be a desirable feature, but it is also meaningful to provide only query privacy (e.g., Signal's privacy contact discovery [4]).

Regarding oblivious maps, the state-of-the-art algorithm is due to Wang et al. [1]. They showed how to leverage techniques used to construct tree-based Oblivious RAM (ORAM) [5–7] in a non-blackbox way to construct an oblivious map, such that the resulting algorithm is a logarithmic factor faster than generically simulating an insecure balanced binary search tree using a state-of-the-art ORAM scheme such as Path ORAM [6] or Circuit ORAM [7]. Indeed, Wang et al.'s ingenious algorithm was optimized and implemented in subsequent works such as ObliDB [3] and Oblix [2].

### 1.1 Sub-Optimality of Prior Work

Our key observation is that existing enclave-based oblivious map implementations turn out to be *sub-optimal both asymptotically and concretely*. There are two main reasons that leads to the sub-optimality — below we will use Oblix [2] as an example since it is the state-of-the-art implementation.

**Choice of metric.** Most of the theoretical work on oblivious algorithms [5,6,8,9] use the program's *computation overhead* (i.e., *number of instructions*) as a primary performance metric. In particular, the theoretical literature on oblivious algorithms

typically measures cost by considering the blowup in terms of the number of instructions when comparing the oblivious version of the algorithm vs. the non-oblivious baseline. Not surprisingly, the recent work Oblix adopted the computation overhead as primary metric, too.

However, for an enclave-based setting, a more significant performance bottleneck is actually the overhead of page swaps between the enclave and insecure memory or disk. As we know, the enclave has a limited amount of secure memory (e.g., 128MB or 256GB [10] depending on the version of the SGX) and the database may not fit within the enclave. Therefore, the enclave must adopt a page swap mechanism to swap a page from insecure memory outside the enclave to secure memory. Sometimes, this page swap can involve a disk swap if the database is too large to fit within the RAM. 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, even if we only need query privacy and not data privacy, the oblivious data structure must nonetheless be encrypted. During this page swap, the enclave needs to decrypt the data fetched from external memory or disk, and re-encrypt it when writing it back. Last but not the least, if the page swap also incurs a disk swap, then we must fetch data at a granularity of 512B or 4KB pages even if we are interested in reading only a single byte.

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 design.

**Sub-optimal algorithm for strong obliviousness.** The original Path ORAM [6] and oblivious data structure [1] papers presented their algorithm in a client-server setting where the ORAM client is fully trusted. By contrast, we are considering a setting where the client-side algorithm is executing in the server's hardware enclave. An important security concern is cache-timing attacks [11–14]. 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 [15, 16], including Oblix [2] suggested a strong notion of obliviousness, often referred to as *strong obliviousness* [15, 16] or *double obliviousness* [2]. 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 even the client-side algorithm oblivious [7, 17], Oblix [2] 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, 17].

## 1.2 Our Contributions

We revisit how to design and implement an efficient oblivious map for hardware enclaves. We design and implement EnigMap, an enclave-assisted multi-map data structure that can be used to perform key-value lookups, insertions, and deletions.

EnigMap differentiates from prior work in this space in the following senses: we adopt ideas from an elegant line of work that originated in the algorithms community, called *external-memory* algorithms [18, 19]. 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 in the algorithms community [18, 19], to the best of our knowledge, we are *among the first to implement and evaluate such algorithms in the hardware enclave context*. We now explain our contributions more concretely.

**Locality friendly layout.** An oblivious binary search tree algorithm leverages a Path ORAM tree as an underlying data structure [1]. Every search query in an oblivious binary search tree requires visiting $O(\log N)$ paths in the underlying ORAM tree (where each path travels from the root to some leaf). Inspired by known external-memory algorithms [20, 21], 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, prior work such as Oblix [2] uses a simple heap layout for storing the ORAM tree, and they incur $O(\log N)$ overhead for visiting a path. Thus, our improvement over the state-of-the-art is asymptotical in nature.

**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 $O(N\log N)$ computation, 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 $O(N\log^3 N)$ computation. Concretely, for a database of size 256 million entries, we can reduce the initialization time from 80.31 hours to 9.5 hours.

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 $\beta$ is the batch size. $\widetilde{O}(\cdot)$ hides $\mathrm{poly}\log\log$ factors.

| Scheme | Cost per batch of operations | | Cost of initialization | |
|---|---|---|---|---|
| | page swaps | compute | page swaps | compute |
| Signal [4] | $O(N/B)$ | $O(\beta^2+N)$ | $O(N/B)$ | $O(N)$ |
| Oblix [2] | $O(\beta\log^2 N)$ | $O(\beta\log^3 N)$ | $O(\frac{N}{B}\log^2 N)$ | $O(N\log^3 N)$ |
| EnigMap | $O(\beta\log_B N\cdot\log N)$ | $\widetilde{O}(\beta\log^2 N)$ | $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ | $\widetilde{O}(N\log N)$ |

**Ensuring strong obliviousness.** In comparison with Oblix, we employ asymptotically more efficient algorithms [17] for ensuring strong obliviousness that allows us to save a logarithmic factor in computation (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 [22, 23] 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 [20,21] 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). Our implementation is general and not tightly coupled with any specific trusted hardware technology. In particular, our main `enigmap_lib` has 5000 lines of code whereas integration with Intel SGX SDK has less than 100 lines of C++ code plus 10 lines of Enclave Description Language (EDL) definitions. This means that it would be very easy to integrate our code with other trusted hardware SDKs.

**Evaluation.** We evaluate the performance of EnigMap and compare with the following two baselines:

1. Oblix [2], which is the state-of-the-art prior to our work;
2. Signal's batched linear-scan algorithm for private contact discovery.[1]

[1] *Concurrent and independent* to our work, Signal updated their private contact discovery implementation to use Path ORAM [24]. Their new design provides only query privacy but not database privacy. We will discuss their concurrent work in more detail in Sections 1.3 and D.

**Under a realistic database size of 256 million, we achieve 15000×, 1500×, 150×, 15× 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 **53× speedup** regardless of the batch size. Our performance gain is asymptotical in nature, so for larger (e.g., 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 [3, 25–31]. Indeed, our savings come not just from system-level optimizations but also from asymptotical improvements (see Table 1).

## 1.3 Additional Related Work

**Signal's private contact discovery, version 1.** In private contact discovery, a client wants to find out which of its friends have also signed up for the messenger service, but without disclosing its friend list to the server. Signal, a privacy-preserving messenger application with 40 million to billions of users [4, 32, 33], is the first messenger app to provide privacy in their contact discovery mechanism.

Signal published a blog post [4] in 2017 that explains their linear-scan algorithm that runs in an SGX enclave. In particular, in their first version, they simply linearly scans through the database to hide the entry the client is interested in. To reduce overhead, they batch multiple queries and process them in a single linear scan.

Signal's 2017 blog post argues that their batched linear scan outperforms oblivious algorithms. The initial motivation of our work was to revisit the question whether batched linear scan indeed has better concrete performance than efficient oblivious algorithms as asserted by Signal. Our findings show quite the opposite: at a database size of 256 million entries, EnigMap outperforms Signal's batched linear scan implementation by 5.5x to 5500x depending on the batch size.

**Concurrent and independent work.** We attempted to contact Signal attaching a copy of our paper in April 2022, hop-

ing to convince them to change their implementation to Path ORAM. In August 2022, Signal launched a new implementation indeed using Path ORAM. However, their approach differs from ENIGMAP. In particular, they compile a hash table generically using Path ORAM. If the queried element has a collision, they retry hashing the element again until it is found. To avoid the number of collisions leaking information, even if the element is found early, they still pad the number of accesses to the maximum number of collision among all the elements. Because they reveal the actual maximum number of collisions in the current execution (and not a worst case that is guaranteed except with negligible probability over the choice of the random execution), their algorithm provides *only query privacy* and *not database privacy*. To additionally provide database privacy, they would have to replace the actual maximum encountered with a worst-case value encountered in all but $1 - 1/2^{\lambda}$ fraction of the random executions, and this change can cause their performance to degrade by one or two orders of magnitude depending on the security parameter desired.

ENIGMAP achieves comparable concrete performance to Signal's new implementation while additionally providing database privacy. This makes ENIGMAP more broadly applicable, e.g., it can be used in secure outsourced databases too where database privacy is a first-class concern.

**Oblivious RAM and oblivious algorithms.** Oblivious RAM was first proposed by Goldreich and Ostrovsky [8, 9] who gave a hierarchical construction with $O(\log^3 N)$ overhead, assuming the existence of one-way functions. Several subsequent works [34–39] 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, 27], 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 [1, 22, 40] such that we can outperform generic compilation for specific (classes of) computation tasks. Wang et al. [1] 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 [2] and ObliDB [3], Snoopy [25] also implements 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 [2] 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 [41], ZeroTrace [31] and Kloski [42] 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.

Ren et al. [28] briefly explored the idea of using a locality-friendly layout in a somewhat different context, namely, in the design of an ORAM-capable secure processor. They did not provide details, so likely they did not employ our optimizations that achieve a 2x factor speedup over the standard Emde Boas layout (see Section 4.1 for details). Tamrakar et al. [43] provide oblivious membership test which is not sufficient to realize a key-value store; further, their algorithm offers only weak privacy: they do not guarantee the privacy of items that are found during the membership test.

Intel's new version of SGX, called SGX2, provides a larger enclave memory than before, e.g., 256GB [10]. Even though this is much larger than the previous 128MB, typical real-world server-side databases easily exceed this capacity, and thus can benefit from external-memory algorithms.

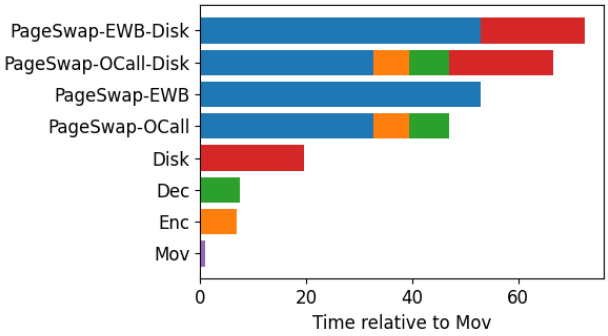
**Oblivious algorithms in a client-server setting.** Another line of works, including Ring ORAM [27], Obladi [26] and others [1, 34, 35, 44, 45] implemented oblivious algorithms in a client-server setting. In a client-server setting, the main performance bottleneck is the bandwidth, and the client-side 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.

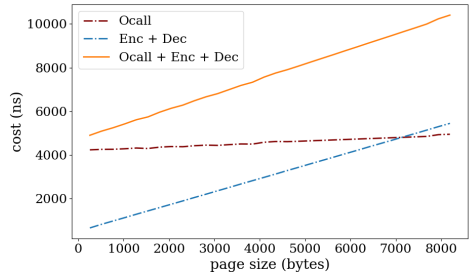
## 2 Problem Statement

### 2.1 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




(a) **Cost of each operation relative to MOV.** A page swap is about 47 times more expensive than moving 4KB in memory *within* the enclave. The costs are color-coded which shows the cost breakdown.




(b) **Effect of page size.** The cost of swapping a page of size $B$ can be viewed as a linear curve $C_{\text{init}} + \gamma B$ where $C_{\text{init}}$ is a startup cost of swapping even 0 bytes, and $\gamma \ll C_{\text{init}}$ is the cost for swapping each extra byte.

Figure 1: **Microbenchmarking results**

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.

When the enclave's secure memory is overcommitted relative to the physical memory available, a page swap mechanism is needed for the enclave to fetch (encrypted) pages from RAM as needed. As explained later in Section 2.2, the OS can observe the page-level accesses during the page swap. Besides observing page-level accesses, we also assume that the OS can observe fine-grained memory accesses within the enclave, e.g., through well-known cache-timing attacks [11, 58–60].

## 2.2 Background on SGX

We review some background on SGX especially mechanisms related to memory protection.

**EPC pages and page swapping.** In SGX, the enclave's secure memory (i.e., EPC pages) are protected from the operating system. The EPC pages are encrypted when they are stored in memory — this encryption is enforced in hardware by the SGX processor.

In our scenario, the size of the database can be much larger than the enclave's secure memory size. In these cases, part of the encrypted database will reside either in insecure RAM outside the enclave, or reside on disk. When the enclave wants to read encrypted data that reside outside the enclave, we need to employ an EPC page swapping mechanism, such that the enclave can swap in and out EPC pages as needed.

In SGX, we have two available mechanisms for performing EPC page swapping — with both mechanisms, the OS can observe the address and content of the page being swapped.

- *SGX-specific instructions.* The first approach is to use SGX-specific instructions. For example, the OS can call the EWB instruction to invalidate an EPC page and write an encrypted version of the page out to RAM. Using this approach, encryption of the invalidated EPC page is enforced in hardware, and the page size is forced to be exactly 4KB. The elegant work of Costan and Devadas [58] gives a detailed description of the EWB-based page swap mechanism.

- *OCall.* Another approach is to use `ocall` which is a call from the enclave to the untrusted application. The `ocall` supports transferring a(n encrypted) buffer between the application and the enclave. Using this approach, the enclave's code must encrypt the buffer written out to RAM, and decrypt it when it is later fetched in and used. We can use the processors AES-NI instructions for fast encryption and decryption. Moreover, with the `ocall`-based approach, it is not necessary to stick to a page size of exactly 4KB, although it turns out that for our implementation, a page size of approximately 4KB is concretely optimal — see Section 2.3 and Section 5 for more details.

The first approach (i.e., using SGX-specific instructions) requires a deeper context switch than the second approach, and thus the performance overhead is larger as reported in Ngoc et al. [61] in a comprehensive SGX performance benchmarking effort — this is also confirmed by our own microbenchmarks described in Section 2.3. Specifically, as explained in detail by Ngoc et al. [61], using SGX-specific instructions, the OS

will cause all CPUs executing inside the enclave to exit using interprocessor interrupts to flush the TLB entries. Only then can the OS call EWB to evict the page from EPC memory. In comparison, with the `ocall` approach, we need not stop processors executing inside the enclave — we only need to transfer some encrypted buffer into the enclave, use it to replace some already existing EPC page, and transfer out an encrypted version the old EPC page.

For this reason, in EnigMap, we use the `ocall` approach to perform EPC page swaps, and the same approach was also used in earlier works such as [2, 3, 62].

Finally, note also that a page swap may or may not be accompanied by a disk swap, depending on whether the page being requested resides in (insecure) physical memory or not. If a disk swap is also incurred, it will introduce some additional overhead — see Section 2.3 for more details.

**SGX evolution.** In SGXv1, the maximum EPC size is limited to 128MB. The second generation of SGX, henceforth called SGXv2, removed this limitation, and increased the capacity of the EPC region to up to 512GB. For real-world applications with with large datasets (e.g., Signal's case), often times the database cannot fit within the physical memory made available to the enclave. In these cases, page swaps are still necessary even with SGXv2.

Since SGXv2 is relatively new, in comparison, SGXv1 is still more available than SGXv2. Currently, only the last generation of server-grade Intel processors (released after 2021 Q2) support SGXv2 – most have 64GB of EPC size, except very few high-end processors (5318S, 8352S, 8368, 8380) with 512GB EPC. Therefore, in our work, we evaluate the performance of EnigMap on both SGXv1 and SGXv2. The SGXv1 experiments are particularly meaningful for blockchain-type applications. Specifically, SGX is used widely in blockchain environments to support *offchain* privacy-preserving smart contracts. In such scenarios, the computing platforms are distributed and likely consumer-grade machines without SGXv2 support.

## 2.3 Microbenchmarks and Motivation for External-Memory Algorithms

To the best of our knowledge, almost all prior works [2, 31, 41, 42] that implement oblivious algorithms for hardware enclaves focus on optimizing the computation overhead of the algorithm. This works for the scenario when the encrypted database is small and fits within the enclave's physical memory. However, when the encrypted database cannot fit within the enclave's physical memory, the *cost of EPC page swaps* become a significant overhead.

**Page swap vs compute overhead.** In Figure 1a, we plot microbenchmarking results that compare the page swap overhead vs. computation overhead. These microbenchmarking results are collected on an Azure SGXv2 machine with 4GB EPC, 8GB RAM, an SSD drive with 80000iops/1.2Gbps and an Intel Sunny Cove 3rd generation Xeon processor.

Specifically, the figure compares the cost of performing a 4KB EPC page swap vs. the cost of moving 4KB of data within the enclave's EPC memory. For performing a page swap using `ocall` or EWB, we plot two cases depending on whether the page swap is also accompanied by a disk swap.

The figure suggests that an EPC page swap using SGX's built-in EWB instruction is 53× or 72× more expensive than moving 4KB of data within the enclave depending on whether there is a disk swap, and an `ocall`-based EPC page swap is 46× or 66× more expensive than moving 4KB of data within the enclave, depending on whether is a disk swap. The work of Ngoc et al. [61] also showed very similar findings as us — see Section 2.2.

Specifically for the `ocall`-based approach, the total cost of the EPC page swap includes the following parts: the cost of an `ocall` including the context switch and transferring 4KB of data (shown in blue), the encryption cost (shown in orange), the decryption cost (shown in green), and optionally the disk swap cost (shown in red) depending on whether the page swap is accompanied by a disk swap. Recall that in our implementation, the encryption and decryption are performed by calling the processor's AES-NI instructions from the enclave. Absent a disk swap, the `ocall`-cost is roughly 70% of the total page swap cost, and the encryption and decryption costs are roughly 30% of the total page swap cost.

**SGXv1 vs. SGXv2.** Figure 1 is collected from an SGXv2 machine. We also repeated the same microbenchmarking tests on a SGXv1 machine with 28MB EPC, 4GB RAM, an SSD drive with 80000iops/1.2gbps, and a Xeon E2200 processor.

The results are almost the same except that the MOVs are about 2× slower on SGXv1 than SGXv2, and thus the relative cost of other operations w.r.t. MOV are about twice the amount shown in Figure 1a.

**Effect of page size.** With the `ocall`-based approach, we have flexibility in deciding the page size. Our microbenchmarking results show that the block size should not be too small due to the significant overhead of the `ocall` itself and the cost of encryption/decryption. As shown in Figure 1b, the cost of swapping a page of size $B$ (without involving disk swap) can be viewed as a linear function $C_{\text{init}} + \gamma B$ where the $y$-intercept $C_{\text{init}}$ is the "startup" cost of swapping a buffer of size 0 (analogous to the network latency), and $\gamma \ll C_{\text{init}}$ is the cost for swapping each extra byte (analogous to the inverse of network bandwidth). Figure 1b does not take disk swap into account — if additionally the page swap involves a disk swap, then we want to choose a page size that is a multiple of the inherent block size for the disk (typically 512B or 4KB).

In our work, the optimal page size $B$ depends not only on these micro-benchmarking results, but also on our algorithm

and the size of each data entry. Jumping ahead, we will show that the optimal choice of $B$ for us is approximately 4KB — see Section 5 for details.

**Why external-memory algorithms?** As mentioned, due to the overhead of `ocall`, encryption, and decrytion, it does not make sense to use a page size that is too small. When we choose a larger page size (e.g., 4KB), we also need to redesign our algorithms such that whenever a page is fetched, it can make maximal use of the page before evicting it from the enclave. By contrast, algorithms access data in a scattered and non-local manner would be undesirable in this context.

For this reason, external-memory algorithms are a perfect fit for SGX enclaves. When the external-memory model was first proposed [18], the desired application scenario was to optimize the I/O cost of algorithms when data cannot all fit within some fast cache, and has to be swapped out to slower storage. Our work is *the first to implement external-memory algorithms within the context of SGX enclaves.*

**Terminology and notation: the literature and the SGX context.** The external memory model, first proposed Aggarwal and Vitter [18], 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 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. Besides minimizing computation, external-memory algorithms typically care about minimizing the number of *cache misses* or equivalently, the *I/O cost* (i.e., the number of blocks fetched from or written to memory). Asharov et al. [63] explore locality-based ORAM; however, they propose a different formulation of the locality notion. In the context of hardware enclaves, the external-memory model is the best fit..

In the context of hardware enclaves, the page size corresponds to the block size $B$, and the enclave's resident memory size corresponds to the cache size $M$. $M$ is typically 128MB for SGXv1, and can be as large as $\min(\text{physical memory size}, 512GB)$ for SGXv2. In our work, the optimal page size is 4KB — see Section 5 for a more detailed explanation.

In our paper, we simultaneously optimize the number of page swaps which corresponds to the I/O cost of an external-memory algorithm, as well as the computation overhead.

## 2.4 Definition: Oblivious Multimap

We want to implement an oblivious multimap data structure that implements a key-value store. Just like in the Oblix [2] 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 $\mathsf{Find}$ 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 contents itself (since it is easy to encrypt the data contents). We adopt the standard notion of strong obliviousness [15, 16] (also called double obliviousness [2]). Strong obliviousness 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. 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. 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 Appendix C.2.

Since strong obliviousness is a standard notion, we defer the formal definition to Appendix C.2.

# 3 Background on Oblivious AVL Tree

In this section, we provide some background on oblivious data structures [1]. We assume familiarity with Path ORAM [6], and we provide a review on Path ORAM in Appendix A for readers who need to refresh their minds.

**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.

**Oblivious AVL tree: data structure.** A multimap data structure can be realized with an AVL tree, and our goal is to make the AVL tree oblivious by relying on the ORAM tree data structure introduced in Appendix A. We first give some background on the AVL tree. An AVL tree is a balanced binary search tree [64]. 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 $1.44\log_2 N$ [1, 64].

To make the AVL tree oblivious, we rely on the oblivious data structure techniques of Wang et al. [1]. The idea is to store the entries (i.e., nodes) 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. 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 $\mathsf{ptr}$, $\mathsf{lptr}$, and $\mathsf{rptr}$ 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 $\mathsf{lptr}$ or $\mathsf{rptr}$ is $\perp$. Note that the tuple $(\mathsf{ptr}, \mathsf{lptr}, \mathsf{rptr}, v)$ is viewed as the data field by the ORAM data structure, and the ORAM data structure will add some extra metadata to the entry.

**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., $\mathsf{ReadRm}$), and find the next node along the logical path. After we call $\mathsf{ReadRm}$ for any entry looked up, we assign it a random new path and add it back by calling $\mathsf{Add}$, and then perform an eviction by calling $\mathsf{Evict}(\mathsf{pos})$ on the read path identified by $\mathsf{pos}$.

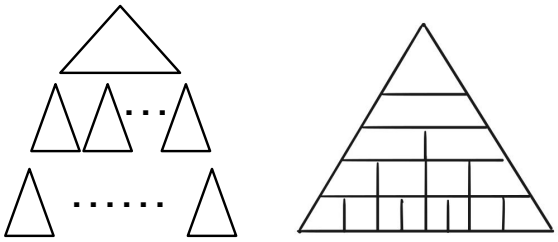
We now describe how to perform $\mathsf{Insert}(k, \mathsf{data})$. Here we use $\mathsf{data}$ to denote the entire payload string of an entry besides the key, where the format of each entry was explained above. To insert an element with the key $k$, we first perform a lookup for $k$, which walks down 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 $\mathsf{Find}$ — 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 $\mathsf{ReadRm}$ 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.

**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



(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 Cormen et al. [65] — the same techniques were adopted by Oblix [2]. We defer the details to Appendix B.1.

# 4 ENIGMAP Design

## 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 [2] and ObliDB [3] 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 [20, 21]. 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 visiting a path in the ORAM tree. 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. 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 [20, 21]. 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* [20, 21], 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 the number of page swaps (in comparison with ENIGMAP). 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 [2]. Our initialization algorithm can be accomplished with $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ page swaps and $O(N\log N)$ computation. In comparison, the initialization algorithm of Oblix [2] incurs $O(N\log^3 N)$ computation and $O(N\log^3 N)$ page swaps.[2]

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 2.4.

Our new initialization algorithm proceeds in two stages:

- *Stage 1:* Stage 1 aims to accomplish the following: 1) pick a random position identifier for each entry in the ORAM tree; 2) assign all entries to a unique node in an AVL-tree; and 3) construct AVL-tree nodes with the correct keys and position identifiers for 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,

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

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 $O(N\log N)$ computation.

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 $\mathbf{X}$ is assigned a random position identifier. 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.

$\underline{\mathsf{Propagate}(r)}$

a) if $r$ is not leaf: let $\mathsf{lptr} = \mathsf{Propagate}(\mathsf{left}(r))$, $\mathsf{rptr} = \mathsf{Propagate}(\mathsf{right}(r))$, and store $(\mathsf{lptr}, \mathsf{rptr})$ to $\mathbf{X}[r]$,

b) Return the key and position identifier in $\mathbf{X}[r]$.

**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 $O(N\log N)$ computation. 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 $O(N)$ computation. For the third propagation step, clearly its computation 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 $O(N\log N)$ computation.

#### 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 $\mathsf{BinPlace}$ 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 B.2. Our stage 2 algorithm is described below where level $\log_2 N$ denotes the leaf level, and level 0 denotes the root level. Intuitively, the algorithm places elements in the ORAM true level by level. For the larger levels, it leverages an oblivious bin placement alorithm; for the smaller levels, it it uses a naïve quadratic-time algorithm.

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 $\mathsf{BinPlace}$ 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 $O(N\log N)$ computation. We defer its correctness and performance analysis to Appendix B.2. In particular, among other things, Appendix B.2 will show why truncating $\mathbf{Y}$ does not cause any elements to be dropped. Observe also that because we always truncate $\mathbf{Y}$ by half, the algorithm does not reveal how many elements go into each level of the ORAM tree.

**Remark 4.1.** *Oblix's initialization algorithm [2] 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 computation overhead to be $O(N\log^3 N)$. Oblix also did not evaluate their initialization algorithm.*

#### 4.2.3 Improved Algorithm for Stage 2

In Appendix B.3, we describe how to employ techniques from Ramachandran and Shi [16] in a *non-blackbox* manner to simplify the algorithm (in a practical implementation) and improve its concrete performance by a constant factor.

### 4.3 Deferred Contents

In the interest of space, we defer the following contents to the appendices:

- *Multi-level caching.* ENIGMAP employs a multi-level caching scheme to optimize concrete performance. We present the multi-level caching scheme in Appendix C.1. In particular, it is important to ensure that the caching does not harm privacy.

- *Ensuring strong obliviousness.* To resist cache-timing attacks, we took extra care to achieve strong obliviousness (also called double obliviousness). We want that not only are the page-level access patterns revealed to the OS oblivious, but also the memory accesses within the enclave. For the latter, we want to ensure both data-trace obliviousness and instruction-trace obliviousness. Appendix C.2 discusses how we achieve strong obliviousness.

- *Practical optimizations.* We discuss several practical optimizations in Appendices C.3 and C.4.

- *Achieving integrity and freshness.* To achieve integrity, we use a special Merkle tree that is overlapped on top of the ORAM tree. Our technique is inspired by several prior works [28, 66, 67]. Our experiments show that the additional overhead for achieving integrity and freshness is only 1-2% — see Appendix C.5 for a more detailed description.

## 5 Implementation and Experimental Results

**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.** Our code has been open sourced and is publicly available at `https://github.com/odslib/EnigMap`.

### 5.1 Experimental Setup and Baselines

We compare ENIGMAP with the following baselines:

- **Signal's private contact discovery**. We downloaded Signal's private contact discovery implementation [68], 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 [2]. 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/slowdown of Oblix over Signal's implementation running on the same machine (i.e., their machine). By comparing our relative speedup with their relative speedup both over Signal's implementation, we are also able to compare with Oblix.

Besides Oblix [2], other implementations of oblivious multimaps exist, e.g., ObliDB [3]. However, as acknowledged in the ObliDB paper [3], 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.

We ran two sets of experiments for ENIGMAP and Signal's implementations, for SGXv1 and SGXv2, respectively. The SGXv1 machine has an intel Xeon E2200 processor, and the SGXv2 machine has an unspecifed Intel Ice Lake Xeon processor (Azure DC32ds_v3). For the SGXv2 machine, the maximum EPC size allowed is 192GB.

### 5.2 Experimental Results

**Comparison with Signal's linear-scan implementation.** Figure 3 (for SGXv1) and Figure 4 (for SGXv2) compare 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 $\beta$ queries, their algorithm incurs $O(\beta^2 + N)$ computation [2] and $O(N/B)$ page swaps. In our ENIGMAP implementation, we process the requests in the batch one by one. In all of our results below, the query times for batches are the total time for the entire batch (not averaged over the batch size). In practice, the actual batch size depends on the rate of the requests. Signal's open source code uses a maximum batch size of 8192, and their tests use batch sizes of 1, and powers of 2 starting at 256 and ending at 8192.

Signal's number of monthly active users increased from 20 million at the end of 2020, to 40 million in 2021 (see also Figure 13 in the appendices). In Signal's blog post [4], they stated that they want to support billion-sized databases.

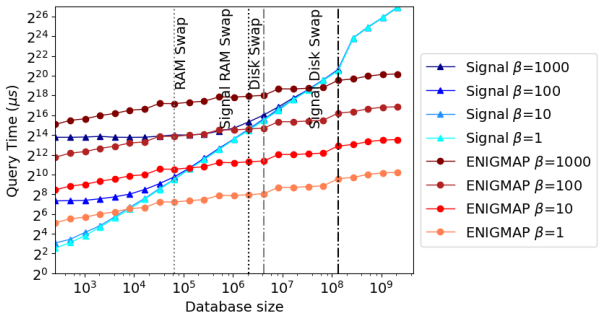



Figure 3: **Comparison of ENIGMAP and Signal on SGXv1**. Enclave memory size is 128MB, RAM size is 16GB. The vertical lines mark when ENIGMAP and Signal start to incur RAM swap and disk swap respectively. The largest database in this plot is 1TB.

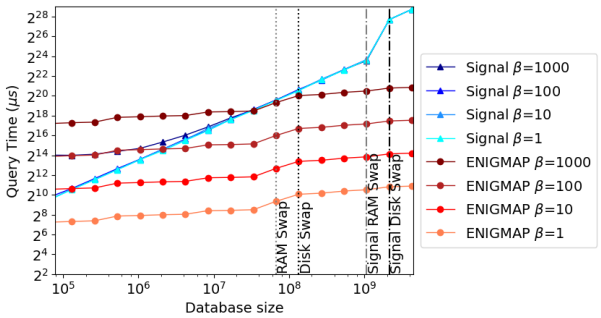



Figure 4: **Comparison of ENIGMAP and Signal on SGXv2.** Enclave memory size is 192GB, RAM size is 256GB. The vertical lines mark when ENIGMAP and Signal start to incur RAM and disk swaps, respectively. The largest database in the plot is **2TB**.

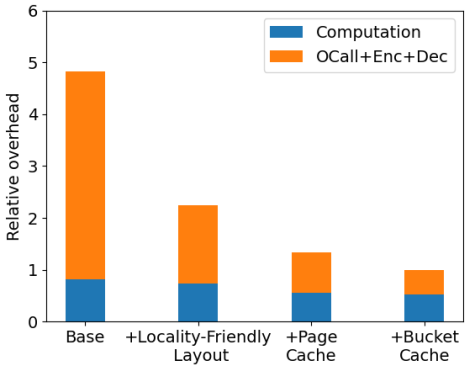



Figure 5: Cost breakdown and the effects of several optimizations for $N = 2^{24}$. In this case, there is only RAM swap but no disk swap.

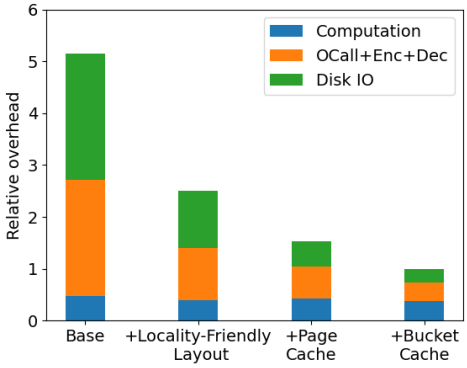



Figure 6: Cost breakdown and the effects of several optimizations for $N = 2^{28}$. In this case, there are both RAM and disk swaps.

At a database size of $2^{28}$ entries (i.e., 256 million), our experiements on SGXv1 show a speedup of **15000×, 1500×, 150×, 15×** 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 $2^{14}$, $2^{22}$, and $2^{25}$, respectively. For our SGXv2 experiments, we used even larger databases up to $2^{32}$ entries (i.e., 2TB). At a database size of $2^{32}$ entries, our SGXv2 experiements show a speedup of 130000×,13000×,1300×,130×, at a batch size of 1, 10, 100, 1000, respectively.

Although the prior work Oblix [2] 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 512 million entries.

**Comparison with Oblix.** The Oblix paper [2] reported a *slowdown* of 12× and 3.5× relative to Signal at a batch size of 1000, and for a database size of $2^{26}$ and $2^{28}$ entries, respectively. In comparison, at the same batch size, ENIGMAP achieves a *speedup* of 2× and 15× and for a database of size $2^{26}$ and $2^{28}$ entries, respectively. This shows that our speedup over Oblix is **24× and 53×** at a database size of $2^{26}$ and $2^{28}$ entries, respectively. For this set of experiments, we ran ENIGMAP and Signal on SGXv1, to match the experimental platform of Oblix.

**Cost breakdown.** Figure 5 and Figure 6 show 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 Figure 5, everything fits in RAM whereas in Figure 6, the database does not entirely fit in RAM, so disk swaps are needed on some page swaps.

In both figures, 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

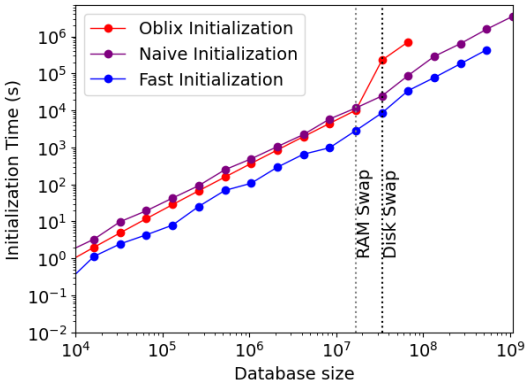



Figure 7: Initialization cost of EnigMap

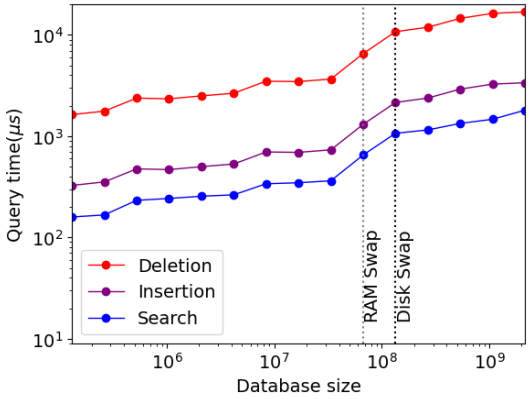



Figure 8: EnigMap: cost of different operations

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 2 to 5.1 factor.[3]

An interesting observation is that without the optimizations, page swap cost (`ocall`, encryption, decryption, and possibly disk IO) occupies 80% to 90% of the total cost. However, with all of our optimizations turned on, page swap cost now occupies only 50% to 60% of the total cost.

For the case without disk swap, Figure 5 also indirectly shows that *even our unoptimized base performance is roughly* 2.5× *faster than Oblix*, since for a $2^{24}$-sized database, our final construction outperforms Oblix by 13×. This is because even without the optimizations and the locality-friendly layout, the computation overhead of our algorithm is asymptotically better than Oblix's implementation (see Table 1).

[3]The computation in the base version looks slightly less than the rest, and this is due to side effects of running the profiler.

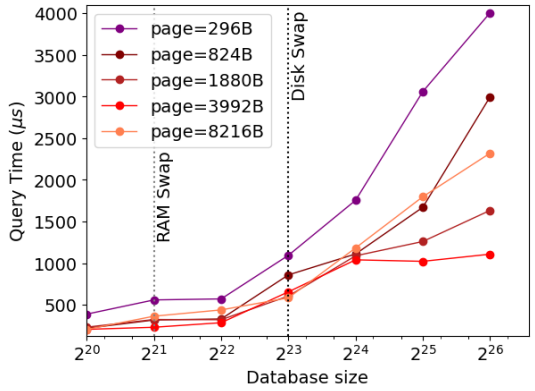



Figure 9: EnigMap: **Effect of page size.**

**Effect of different page sizes.** Figure 9 shows the effect of different page sizes. In this experiment, the enclave memory size is set to 4GB and the RAM size is set to 16GB. In our scheme, the number of page swaps per query is $\lceil \frac{\lceil \log_2 N \rceil}{\lceil \log_2 B \rceil} \rceil$ (where $B$ is the page size measured in terms of how many entries fit in a single page). Let $c_B$ be the cost of swapping a page of size $B$. Roughly speaking, the total cost associated with page swaps is minimized when $c_B \cdot \lceil \frac{\lceil \log_2 N \rceil}{\lceil \log_2 B \rceil} \rceil$ is minimized. Our experiments show that absent disk swaps, a page size of 3992 bytes achieves the optimal concrete performance. On the Azure machine we use, the disk adopts an inherent block size of 4KB. Therefore, in all our experiments, we chose a page size of 3992bytes when there is no disk swap, and this size gets padded to 4KB when the page gets swapped to disk. In the remaining experiments in this section, we always stick to a page size of 3992 bytes.

**Initialization cost.** Figure 7 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 8.7× faster than our own naïve initialization algorithm (a reduction from 80.31 hours to 9.5 hours), and even our naïve algorithm performs 2.1× faster than Oblix's algorithm. At a database size of roughly 32 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 8 shows the cost for different operations w.r.t. the database size. As we can see,

insertion is about 1.5× to 2× more expensive than search, and deletion is 5× more expensive than insertion. This is because insertion needs to perform rebalancing in one node, and deletion needs to perform more rebalancing than insertion.

**Comparison with Signal's concurrent work.** In Appendix D, we additionally compare with Signal's new private contact discovery [24], which is concurrent and independent to our work.

## Acknowledgments

This work is in part supported by a DARPA SIEVE grant, a Packard Fellowship, NSF awards under the grant numbers 2128519 and 2044679, and a grant from ONR under the award number N000142212064.

## References

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

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

[3] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *VLDB*, 2019.

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

[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, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

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

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

[10] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *OSDI*, 2021.

[11] 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 CCS*, 2009.

[12] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *ISCA*, 2012.

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

[14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *ACM CCS*, 2012.

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

[16] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *SPAA*, 2021.

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

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

[19] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. 2002.

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

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

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

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

[24] Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/, 2022.

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

[26] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[27] 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.

[28] 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.

[29] 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 CCS*, 2013.

[30] 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.

[31] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *NDSS*, 2018.

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

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

[34] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

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

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

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

[38] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, 2011.

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

[40] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

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

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

[43] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. 06 2016.

[44] 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 *FAST*, 2013.

[45] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. Efficient oblivious query processing for range and knn queries. *IEEE TKDE*, 2021.

[46] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *CRYPTO*, 2005.

[47] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *HotSec*, 2011.

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

[49] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS*, 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 *USENIX Security*, 2018.

[55] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 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 *MICRO*, 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, *ASPLOS*, 2015.

[58] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[59] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT 17*, 2017.

[60] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EuroSec*, 2017.

[61] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything you should know about intel sgx performance on virtualized systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(1), mar 2019.

[62] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. Adore: Differentially oblivious relational database operators. *Proc. VLDB Endow.*, 16(4):842–855, dec 2022.

[63] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Locality-preserving oblivious RAM. *J. Cryptol.*, 35(2):6, 2022.

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

[65] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[66] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *POPL*, 2014.

[67] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *HPEC*, 2013.

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

[69] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In *TCC*, 2017.

[70] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[71] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[72] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. In *TCC*, 2015.

[73] 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.

# Appendices

## A Background: 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. In particular, a path can be fully identified by the leaf node that the path leads to.

**Operations on the ORAM tree.** For the time being, we make a simplifying assumption and assume that all entries in the logical multimap have *distinct* keys. *Later on in Section 3, 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.

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. 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 $\mathsf{data}'$ is the new data to overwrite with or the same as the returned data if no update is needed, and $\mathsf{pos}'$ denotes a randomly selected new path;
3. next call $\mathsf{Evict}(\mathsf{pos})$ where pos is the path which we have just read from.

# B Deferred Algorithmic Details

## B.1 Support Key Multiplicity

To support key multiplicity, 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 2.4, we require that entries with the same key be sorted 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 C.4).

## B.2 Warmup Initialization Algorithm (Stage 2)

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 [69] solves the following problem. Suppose we are given an input array of length $n$ containing real and filler elements.[4] 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 [69] 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 $O(N \log N)$ computation, if we use an external-memory oblivious sorting algorithm such as

[4] We often use $n$ to denote the size of a problem instance, where $N$ represents the size of the database globally.

the one by Ramachandran and Shi [16]. 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 computation.

**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 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 $\ell$ costs $O(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B})$ page swaps and $O(n\log n)$ computation 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 $O(n^{2/3})$ computation.

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 $O(N)$ computation.

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

## B.3 Improved Initialization Algorithm (Stage 2)

We describe how to employ techniques from Ramachandran and Shi [16] 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 [16]. Fix $Z = \omega(\log N)$ to be any super-logarithmic function in $N$. 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 any 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 assume that the enclave's memory size $M = \omega(\log N)$, commonly referred to as the "tall-cache" assumption and also true in practice.

1. Let $Z = M/C$ for a sufficiently large constant $C > 1$, and let $\ell^*$ 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 $\ell^*$.

   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. Fix any $Z' \leq Z$ that is an arbitrarily small superlogarithnmic function in $N$, each array is guaranteed to have size at most $Z'$ 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, 70] implies that the total number of real elements in each pair is upper bounded by $Z'$ except with negligible probability — see Lemma 5 of their Section 5.6 [6]. 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 $\ell^*$ down to the root to a locality-friendly layout.

**Performance bounds.** Ramachandran and Shi [16] suggest an ORBA algorithm that achieves $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ page swaps and $O(N\log N)$ computation. Step 1 takes only one ORBA invocation on $O(N)$ elements. Step 2 can be accomplished with $O(N/B)$ page swaps and $O(N(\log\log N)^3)$ computation if we use bitonic sort to realize the oblivious bin placement. In Step 3, for a level of the tree with $k$ nodes, we consume $O(k\cdot(\log\log N)^2)$ computation 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 $O(N)$ computation 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 computation 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 computation overhead, but the number of page swaps is unaffected.

# C ENIGMAP: Architecture and Practical Optimizations

## C.1 Secure Multi-Level Caching

### C.1.1 Caches for Physical Storage

ENIGMAP leverages multi-level caching for optimization.

- *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 [27,29,30], i.e., caching a small number of levels near the root.

### C.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 [2]. 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.[5] 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 [69] — however earlier works used batched evictions for a different purpose than us. Furthermore, earlier works [69] showed a stochastic domination

[5] 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.

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.

## C.2 Ensuring Strong Obliviousness

Strong obliviousness [15, 16] (also called double obliviousness [2]) requires that not only the page-level accesses are hidden from the adversary, but also the memory accesses within the enclave.

### C.2.1 Definition of Strong Obliviousness

Let $\{\mathsf{op}_\ell\}_{\ell\in[m]}$ be a request sequence of length $m$ where each $\mathsf{op}_\ell$ is of the form $(\mathsf{Size}, k)$, $(\mathsf{Find}, k, i, j)$, $(\mathsf{Insert}, k, v)$, or $(\mathsf{Delete}, k, v)$. 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 $\mathsf{op}_\ell$ and $\mathsf{op}'_\ell$ 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 $\lambda$ is a security parameter, and we assume that the multi-map is invoked with the security parameter $\lambda$.

To provably defend against cache-timing attacks, we want the algorithm to have memory-trace obliviousness even within the enclave. 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.

### C.2.2 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. [17] (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 $\widetilde{O}(\log N)$ computation, where $\widetilde{O}(\cdot)$ hides $(\log\log N)^2$ factors (assuming that Bitonic sort [71] is used as the oblivious sorting algorithm). For both metrics, we achieve asymptotical improvement over Oblix [2]: Oblix's path read and eviction algorithm incurs $O(\log N)$ page swaps and $\Omega(\log^2 N)$ computation. This is because they run a double loop on the metadata of the path, and they do not use the locality-friendly layout.

### C.2.3 Instruction-Trace Obliviousness

We use standard techniques for ensuring instruction-trace obliviousness [22, 23]. 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 $X$.

```
1 if (X) { B = C; }        |||  CMOV( X,B,C);
2 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 [22]. 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 10 is Oblix [2]'s code for searching the AVL tree. Their implementation is not instruction-trace oblivious for several reasons:

```
1  fn find_helper(avl_key: &AVLKey<K> /.../) //...
2  {
3   let mut root_key = root_key.clone();
4   while let Some(r_key) = root_key {
5    let cur_node = self.ods_ref.borrow_mut().
       access(Read(ActualOp, &r_key), server)? //
       ...
6    if cur_node.key() == avl_key {
7     return Ok(Some(cur_node));
8    } else if avl_key < cur_node.key() {
9     root_key = cur_node.left_key();
10   } else {
11    root_key = cur_node.right_key();
12   }
13  }
14  return Ok(None);
15 }
```

Figure 10: Oblix's AVL tree search code violates instruction-trace obliviousness.

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 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 `if`s.

## C.3 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. [1] as well as Oblix [2], 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 11 is Oblix's recursive implementation. The issue with this implementation is similar to that of Figure 10, 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

```
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 11: Oblix's AVL tree insert code violates instruction-trace obliviousness.

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 computation (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. [1] incurs an extra $3\times$ overhead for this reason.

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.

## C.4 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 email, phone number, and location. In case that the value field is large, we can rely on an idea from earlier works [1], 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, 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). In Signal's private contact discovery application, the keys are distinct, so this optimization is applicable.

## C.5 Achieving Integrity and Freshness

In our implementation, we use AES-GCM mode to encrypt and authenticate every block along with a nonce, which guarantees confidentiality and integrity of the page. There is a simple modification of this that allows us to additionally get freshness, that is, at the ORAM-tree level, we can store the nonces of the two children in the parent, and the nonce of the root is stored in the enclave's cache. This can be viewed as a "customized Merkle tree" specifically optimized for a tree data structure. Our technique for getting freshness, i.e., overlapping the Merkle-tree structure with the ORAM tree, is a slight modification of several earlier works [28, 66, 67]. The difference is that we replaced their hash with MAC + nonces.

Our experiments show that the additional overhead of getting freshness is only 1-2%.

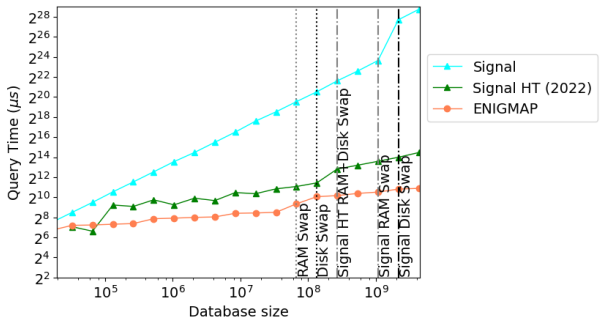



Figure 12: Comparison of ENIGMAP (providing both query and database privacy) and Signal's concurrent work (providing only query privacy but not database privacy)

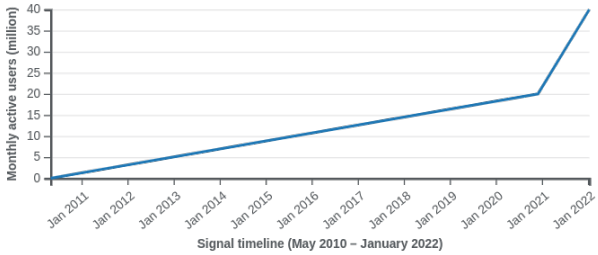



Figure 13: Signal's monthly active users [32]

# D Comparison with Signal's Concurrent Work

As mentioned in Section 1.3, concurrent to our work, Signal released a new design of private contact discovery in August 2022. Their new algorithm, henceforth denoted "Signal HT", uses Path ORAM to obliviously simulate a hash table. Their algorithm guarantees only query privacy but not database privacy, so in this sense ENIGMAP provides stronger security guarantees. We compare Signal HT with ENIGMAP in Figure 12. In this figure, for both ENIGMAP and Signal HT, the enclave memory size is 192GB and the RAM size is 256GB. Since Signal HT does not use batching like their previous implementation, the batch size is 1 in this evaluation. In Signal HT, the computation overhead of the algorithm is randomized — it depends on the maximum level of hash collisions . Therefore, we compare with their average performance in Figure 12. In summary ENIGMAP **achieves 3x speedup over Signal HT at a database size of 256 million, and meanwhile provides stronger security**.

Note also that in Signal's new design, they run several *parallel* instances of their Path-ORAM-based oblivious hash table on multiple machines, using techniques similar to oblivious parallel RAM [69, 72] and Snoopy [25]. In our evaluation, we are comparing with the performance of their single instance, since ENIGMAP can also be used as a drop-in replacement of a single instance in their parallel architecture.

## E 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 [2]. A more efficient solution is to employ the oblivious two-tier hash table suggested by Chan et al. [73]. Although Chan et al. [73] 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 3, 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.

**Supplemental figure.** Figure 13 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.