Oblix: An Efficient Oblivious Search Index

Pratyush Mishra Rishabh Poddar Jerry Chen Alessandro Chiesa Raluca Ada Popa
UC Berkeley
{pratyush, rishabhp, jerry.c, alexch, raluca.popa}@berkeley.edu

***Abstract*—Search indices are fundamental building blocks of many systems, and there is great interest in running them on encrypted data. Unfortunately, many known schemes that enable search queries on encrypted data achieve efficiency at the expense of security, as they reveal access patterns to the encrypted data.**

In this paper we present Oblix, a search index for encrypted data that is oblivious (provably hides access patterns), is dynamic (supports inserts and deletes), and has good efficiency.

Oblix relies on a combination of novel oblivious-access techniques and recent hardware enclave platforms (e.g., Intel SGX). In particular, a key technical contribution is the design and implementation of *doubly*-oblivious data structures, in which the client's accesses to its internal memory are oblivious, in addition to accesses to its external memory at the server. These algorithms are motivated by hardware enclaves like SGX, which leak access patterns to both internal and external memory.

We demonstrate the usefulness of Oblix in several applications: private contact discovery for Signal, private retrieval of public keys for Key Transparency, and searchable encryption that hides access patterns and result sizes.

I. Introduction

A search (or inverted) index is a fundamental building block of many systems, and is often used for sensitive data such as personal or corporate information. A rich line of work [8, 12, 11, 17, 33, 36, 39, 40, 51, 52, 62, 63] aims to protect such sensitive information by encrypting it while still allowing *search on the encrypted data*. In this model, when a client wishes to retrieve documents matching a certain keyword, the client generates a *search token* for the keyword and sends it to the server; the token hides information about the keyword, but enables the server to identify all matching (encrypted) documents and return them to the client, who can then decrypt.

Despite significant progress in constructing such *encrypted search indices*, known schemes with good efficiency suffer from an important limitation, namely, they *leak access patterns*. The exact leakage varies from scheme to scheme, but in its basic form it enables identification of which (encrypted) documents match a keyword, for each searched keyword (this is leakage profile L1 in the categorization of Cash et al. [10]).

A recent line of attacks [4, 10, 26, 30, 34, 42, 53, 79] has demonstrated that such access pattern leakage can be used to recover significant information about data in encrypted indices. For example, some attacks can recover all search queries [10, 34, 42, 79] or a significant portion of the content of encrypted documents [4, 26]. Even hiding access patterns can be insufficient: some attacks cleverly use the *number* of documents that match a search query [10, 37], so it is important to hide the *result size* as well. Clearly, preventing such leakage would drastically improve security of encrypted search indices. The go-to method to hide access patterns is Oblivious RAM (ORAM) [27], but this is an expensive tool [10, 34, 50], and thus few prior schemes try to hide access patterns [24], and even fewer hide result sizes.

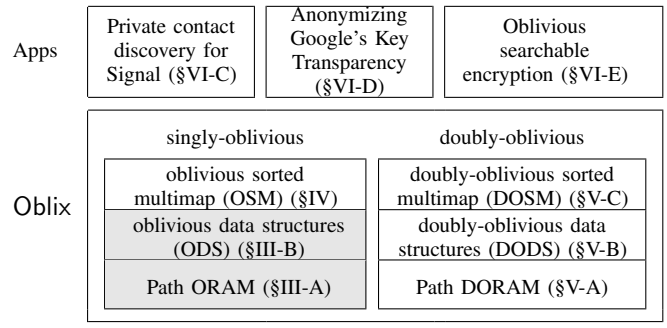

Figure 1: Component stack of Oblix. Grey blocks are components that exist before Oblix, and white blocks are Oblix's contributions.

In this paper, we present Oblix (**OBL**ivious **I**nde**X**), an efficient search index that does not leak access patterns and enables hiding the result size of searches. In particular, Oblix protects against all the aforementioned attacks. At the same time, Oblix supports updates (inserts and deletes) as well as multiple (potentially malicious) users, properties that are challenging to achieve for many prior schemes. While hardware enclaves such as Intel SGX [45] are a key enabler for Oblix, they are far from sufficient, and Oblix leverages a combination of novel cryptographic protocols and systems techniques. Fig. 1 shows the logical layout of our techniques, as well as the three applications that we demonstrate on top of Oblix.

A. Summary of techniques

We discuss the challenges that arose in designing our system, and the techniques that we used to overcome them. Recall that Path ORAM [64] (a popular and relatively efficient ORAM scheme) consists of an ORAM client that stores the secret key and an ORAM server holding the bulk of the data; the ORAM client maintains a *position map*, mapping each item in the search index to a location in the oblivious database, and a stash of temporary values; see Section III-A for details.

Challenge: high round complexity. The position map has size that is linear in the number of entries in the index. In our applications (e.g., contact discovery for Signal), clients cannot store it. The standard solution is to store the position map at the ORAM server in another ORAM instance with its own (smaller) position map, and recurse until the position map is small enough for the client to store. However, this solution implies that each index lookup requires a logarithmic (in the index size) number of requests, and hence roundtrips, to the server. These roundtrips severely degrade latency.

Approach: Our first insight is that, by using recent hardware enclave technology (such as Intel SGX [45]), we can improve latency by reducing network roundtrips, as we now explain. At the server, we place the ORAM client inside the enclave and place the ORAM server in unprotected memory outside the enclave. All accesses to the ORAM server are still oblivious, but interaction between the ORAM client and server happens within a single machine, and not over the network. In Section I-B, we explain that hardware enclaves also let us support multiple users. Unfortunately, simply "throwing the ORAM client inside the enclave" is far from sufficient and, in fact, is insecure.

Challenge: hardware enclaves are not oblivious. Recent attacks have shown that hardware enclaves like Intel SGX leak access patterns in several ways (see Fig. 2).

First, prior work [69, 75] shows that an operating system can observe page-level access patterns and uses this leakage to recover encrypted document contents from the enclave. Second, when the data grows large, it needs to be stored on secondary storage. An attacker can then observe (page-level) access patterns to this secondary storage. Third, an attacker could mount an affordable hardware attack that taps into the memory bus and reads memory addresses coming from an enclave. We note that recent work aiming to prevent a compromised operating system from mounting the page-fault attack [59] does not address the second and third attacks.

In our setting, the above leakage is problematic because the security guarantees of ORAM rely on the attacker not seeing the access patterns of the ORAM client to its *internal* memory. For Path ORAM, this means that if the attacker sees accesses to the client's position map or stash, it can infer access patterns to the ORAM server, defeating the purpose of using ORAM.

Approach: We devise a two-part solution to address this challenge. First, we avoid the need for a position map by constructing an oblivious data structure (ODS) [74] that embeds the position map into the data structure itself.

Second, we make the ORAM client's accesses to its internal state oblivious via novel oblivious algorithms. We call the resulting ORAM scheme *doubly oblivious* because not only are the accesses to the ORAM server oblivious, but so are the accesses to the ORAM client's internal memory. Thus, even if the attacker observes access patterns to the client's internal memory, it learns nothing about the data. We design efficient oblivious algorithms for stash eviction and for initializing the ORAM server. These algorithms were challenging to design because the ORAM client makes complex accesses to the stash and (during initial setup) to the ORAM storage, and cannot rely on any memory location being unobservable to the attacker. We deem our doubly-oblivious algorithms for Path ORAM (called *DORAM*), and also for the ODS framework (called *DODS*), to be of independent interest.

Challenge: hide result sizes. Even if we hide access patterns, we still need to hide the size of the result set for a search query. Indeed, this information can be used to learn the contents of a query or its result set [10, 37]. The simplistic solution is to pad each result size to a worst-case upper bound, but this is too expensive for many applications. For instance, when searching documents, while most keywords might have a modest number of matches, some popular keywords will have a large number of matches, forcing the worst-case upper bound to be large. In fact, Naveed [50] shows that padding to the worst-case size can be more expensive than simply streaming the entire database to the client, obviating the need for ORAM.

Approach: The insight is to examine how the user sees search results in regular systems today. Many applications do not display to users *all* results at once (think of web or email searches), but only a *page* of r results, for some pre-determined m (say, 20). To make these r results meaningful, these applications show the "best" r results according to some order of interest (relevance, chronological, or others [54]). Ordering the results of a search query presupposes embedding *support for scoring* in the search index, which is not traditionally captured by searchable encryption (SE) schemes. We deviate from this tradition and explicitly model scores in the interface of our search index, which supports operations on a *scored* inverted index data structure whose searches return the r highest-scoring search results. Doing so enables us to avoid expensive worst-case padding without compromising security, while providing a meaningful correctness guarantee.

Challenge: ordered lists, efficiently and obliviously. We need to design an oblivious data structure that can (efficiently) search ordered lists, and support insertions and deletions. Simply mapping the multimap $[k \Rightarrow v_1, v_2, \ldots, v_n]$ to a regular oblivious map ([55, 74]) $[(k,1) \Rightarrow v_1, \ldots, (k,n) \Rightarrow v_n]$ is problematic because inserts require shifting $O(n)$ values.

Approach: We design a *doubly-oblivious sorted multimap* (DOSM), a specialized data structure that efficiently supports searching ranges in sorted lists, insertions, and deletions. First, we design a suitable tree data structure, where insertions/deletions run in time $O(\log n)$ instead of $O(n)$ as above, that is compatible with the ODS framework. Next, to achieve double-obliviousness, one might consider simply employing our DODS in place of ODS. However, ODS uses caching to fetch tree nodes more efficiently. Replicating this feature without leaking information about cache contents implies performing a dummy ORAM access upon a cache hit (to give the impression of a cache miss), thus defeating the purpose of the cache. Instead, we carefully analyze our oblivious tree algorithms to allow oblivious caching. For example, for certain tree operations (such as inserts), one can *predict from public information* which nodes will be accessed repeatedly (and thus must be in the cache), and can thus safely retrieve these nodes from the cache without a dummy ORAM access. We make a worst-case access only when such a prediction is not possible.

B. Summary of Applications and Evaluation

We show that Oblix can scale to databases of tens of millions of records while providing practical performance. For example, retrieving the top 10 values mapped to a key takes only $12.7\,\mathrm{ms}$ for a database containing $\sim 16\,\mathrm{M}$ records.

We point out that an *important side effect* of using hardware enclaves is support for *multiple users* even when some users

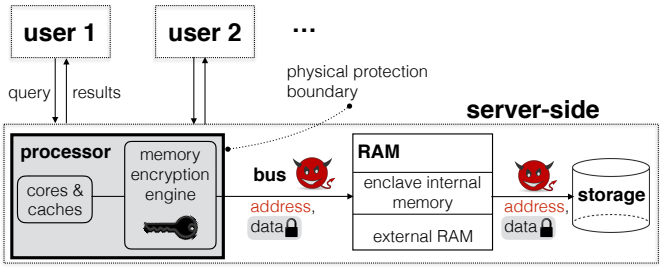

Figure 2: Visibility of access patterns with enclaves such as Intel SGX: the white area is visible to the attacker, while the grey area is not. Data leaving a hardware enclave is encrypted, but memory addresses are not encrypted.

are malicious and want to learn the queries of other users: users submit their queries to the enclave, which executes the queries correctly and privately for each user. A malicious user cannot learn the query of another user. Supporting multiple users is difficult with traditional ORAM systems because they need users to follow an ORAM protocol correctly [6], else they affect each other's privacy.

We also demonstrate the viability of Oblix for three existing applications that can benefit from private search indices. Two of these three applications require support for multiple users. We show how Oblix supports these applications with latencies on the order of a few milliseconds. Since search indices are basic building blocks, we expect Oblix to have further useful applications.

Private contact discovery for Signal. Signal [2] is an encrypted messaging system that recently introduced a service for *private contact discovery*: users can query the service to determine which contacts in their phone also use Signal, *without* revealing their contact list to Signal's servers. The service performs a full scan of Signal's database within an SGX enclave to ensure obliviousness at the server [44]. We show that Oblix provides a solution with lower latency. For every contact in the user's list, Oblix performs a logarithmic search in the database instead of a linear scan.

Anonymizing Google's Key transparency. Google's Key Transparency [1, 46] enables users to discover public keys of other users. The service guarantees the integrity of the retrieved public key, but does not provide anonymity: the server learns the identity of the user whose key it returns. We show that Oblix can be used to anonymize Key Transparency with low latency. In particular, we show that Oblix provides an order-of-magnitude improvement in latency compared to a baseline approach that offers the same level of security.

Oblivious searchable encryption. As discussed above, many searchable encryption (SE) schemes suffer from a long line of attacks [4, 10, 26, 30, 34, 42, 53, 79] that exploit access patterns. We use Oblix to augment the security of searchable encryption by eliminating the leakage from access patterns. We evaluate the augmented SE scheme over the entire Enron email corpus [19] and show that it can support reads/writes with a latency of a few milliseconds.

II. System overview

We provide an overview of Oblix. Fig. 2 shows the architecture of the system: multiple users interact with a server equipped with a hardware enclave. (While our implementation uses Intel SGX enclaves, Oblix's design only requires an abstract notion of an enclave.) The data stored on the server is encrypted with a key held in the enclave. Each user uses remote attestation [35] to ensure that it is communicating with a correctly-setup enclave and establishes a secure (TLS) connection with that enclave. Over this secure channel, the user then sends search, insert, or delete queries to the server, which responds after running Oblix's protocols.

Oblix exposes to applications the interface of a search index expressed as a sorted multimap (see Fig. 1). This index maps a key k to an array of sorted and distinct values $(v_1, \ldots, v_n)$. For example, if one wants to use this index to search words in documents, one maps each word w (the key) to a list of pairs (s_i, d_i), where d_i is the identifier of the document containing w and s_i is a relevance score for that document. As discussed, the scores enable meaningful selection of the top-r documents for a search query, and thus aid in hiding the result size by returning a fixed number of results. Concretely, DOSM provides the function $\mathsf{Map.Find}(\mathsf{k}, i, j) \rightarrow (\mathsf{v}_i, \ldots, \mathsf{v}_j)$, which returns the i-th through j-th values for a key k. The user application can make multiple requests to the same key k for different intervals, and Oblix does not reveal to the attacker that the requests are for the same key or interval. Nevertheless, the user application must exercise caution when issuing many simultaneous requests for the same key to prevent the attacker from correlating them due to their timing. Like in prior work on searchable encryption, we focus *only* on the design of the search index, and recommend standard and complementary techniques [64] to retrieve documents matching a search.

A. Threat model

a) Server: We employ a general and expressive attacker model for a hardware enclave (see Fig. 2). The attacker can perform any hardware attack it wishes on the memory and on the memory bus, but cannot attack the processor in any way, and cannot glean any information from inside the processor (including processor keys). This attacker controls the server's software stack outside of the enclave, including the OS.

With respect to memory access patterns, we assume that the adversary can observe (and modify) memory addresses and (encrypted) data on the memory bus, in memory, or in secondary storage (as in Fig. 2). We divide access patterns into two types: access patterns to data and access patterns to code [69, 75]. The doubly-oblivious algorithms of Oblix prevent leakage of both types of access patterns assuming only a simple "oblivious swap" primitive. Our source code carefully implements these algorithms, but we do not ensure that the final binary hides all access patterns to code because external factors like compiler optimizations and cache replacement policies influence how instructions are fetched into cache. Preventing these factors from introducing data-dependent code accesses

is out of the scope of this paper; complementary prior work exists that can aid in this task [31, 41].

At the same time, Oblix considers as out of scope any side-channel leakage from within the enclave (e.g., cache-timing, branch predictor-based, power analysis, or other timing attacks) [9, 32, 29, 49, 58, 71], as well as rollback attacks [65]. Techniques to mitigate such attacks are complementary to Oblix, and many proposed solutions [14, 16, 31, 59, 60] can be applied to Oblix. Finally, denial-of-service attacks are out of scope: we do not prevent a cloud provider from destroying all client data or denying access to it. Doing so is not in the provider's interest, as clients can choose a different provider.

Oblix achieves protection against modification attacks (e.g., attacker modifies data or queries) via Merkle hash trees [47] both by using Intel SGX's built-in integrity tree and by employing a separate hash tree for data stored outside. These techniques are standard, so in the rest of this paper, we do not elaborate further on them and focus only on how we protect against passive attackers via our (doubly-)oblivious protocols.

b) Client: In Oblix, a client can also misbehave: the client can release its own queries or query results if it wishes, but it cannot affect the privacy of the queries or results of other clients. If a client wants to protect its query privacy, the client should faithfully follow Oblix's protocol.

III. Preliminaries

We use two cryptographic building blocks: *Path ORAM* [64] and *oblivious data structures* [74]. In the next two sub-sections, we recall aspects of these schemes that are relevant to this paper. Below and throughout this paper, we consider algorithms that receive and update a client state; we use the notation "mut st" to emphasize that the state variable st is mutable.

A. Path ORAM

Path ORAM [64] is a type of ORAM protocol [27]. It enables a client to perform oblivious reads and writes to external (server) memory with low bandwidth and latency.

The external (server) memory is arranged in a binary tree of N *buckets*; each bucket stores C *blocks* of B bits each. The client maintains two data structures: (i) a *position map* Positions, which assigns to each block identifier bid a leaf lf in the aforementioned tree such that block bid is stored by one of the buckets on the path from the root to lf; (ii) a *stash* Stash, which maps block identifiers to blocks for all blocks that have not yet been evicted. (The same block identifier may appear in Positions and Stash.) When using Path ORAM for oblivious data structures (see Section III-B), the client does not store Positions in full, but only a small portion of it, reducing the size of the client's state to constant.

Below, we summarize how the client can initialize the external memory (via ORAM.Init), and then read blocks (via ORAM.ReadBlock), modify them, and write them back (via ORAM.Evict). This high-level summary will be useful later.

- **Initialization:** $\mathsf{ORAM.Init}^{\mathcal{S}}(m, [\mathsf{bl}_i]_1^n) \rightarrow \mathsf{st}$**.** On input a maximum number of blocks m, and a list of initial blocks $[\mathsf{bl}_i]_1^n$ (with $n \leq m$), ORAM.Init initializes the server $\mathcal{S}$ with $2^{\log(\lceil m/C \rceil)}$ buckets, and outputs the initial client state st.
- **Read a block:** $\mathsf{ORAM.ReadBlock}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{bid}, \mathsf{lf}) \rightarrow \mathsf{bl}$**.** On input client state st, a block identifier bid, and leaf lf, ORAM.ReadBlock fetches all blocks on the path from the root to lf, inserts these blocks into st.Stash, and outputs the block bl in this path having identifier bid. Furthermore, it assigns bid a new random leaf in st.Positions (ensuring that the next access to this block fetches a random path).

The client can arbitrarily modify the contents of blocks in Stash. To write back modified blocks, the client runs ORAM.Evict on input *all* leaves $[\mathsf{lf}_i]_1^n$ fetched via ORAM.ReadBlock since the last ORAM.Evict call. Informally, eviction reconstructs paths for these leaves using blocks in Stash, and then writes these paths back to the server. Eviction is designed to ensure that Stash has a bounded size, which is crucial for efficiency.

- **Stash eviction:** $\mathsf{ORAM.Evict}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [\mathsf{lf}_i]_1^n)$**.** On input client state st and a list of leaves $[\mathsf{lf}_i]_1^n$, ORAM.Evict constructs buckets on the paths defined by $[\mathsf{lf}_i]_1^n$, as follows. Proceeding layer-by-layer in the ORAM tree, starting from the leaf layer to the root layer, for each block in Stash, determine whether the block may reside in a bucket on this layer (as determined by the corresponding leaf in Positions and whether there is space in that bucket). If so, the block is *evicted* from the stash into that bucket. Any blocks that are not evicted remain in Stash. The final set of buckets is then written back to external memory.

We define security for Path ORAM in Appendix B.

B. Framework for oblivious data structures

Oblivious data structures (ODS) [74] is a framework for designing oblivious analogues of data structures that can be expressed as trees of bounded degree. This property is captured by the next definition.

Definition 1. *A data structure has* **tree-like accesses** *if it is represented via nodes storing (data and) pointers to other nodes such that: (i) every node has a unique predecessor (a node pointing to it); (ii) every operation accesses a unique root node before any other node; (iii) every operation accesses a non-root node's predecessor before it accesses the node.*

The first step of using the ODS framework is to express the desired functionality via a data structure that has tree-like accesses; this could mean modifying an existing data structure or designing one from scratch.

The second step is an initialization procedure that converts an instance of this plaintext data structure into its oblivious counterpart as follows. The client converts plaintext data structure nodes into ODS nodes by replacing all plaintext pointers with *ODS pointers*. (These depend on the underlying ORAM scheme. For example, when using Path ORAM, an ODS pointer is a pair $\mathsf{ptr} = (\mathsf{bid}, \mathsf{lf})$ consisting of a block identifier and a leaf.) Afterwards, the client encrypts and outsources all ODS nodes to the (untrusted) server, while retaining only the root's ODS pointer. Definition 1 ensures that the client does not need to store other pointers.

Subsequently, when executing an operation on the data structure, the client runs a special *start* procedure, then uses an *access* procedure to obliviously perform all the memory accesses required by the operation, and then runs a *finalize* procedure. To access a certain node, the client follows pointers from the root to the node; throughout, the client updates positions and data as required by the data structure operation.

- **Initialization:** $\mathsf{ODS.Init}^{\mathcal{S}}(m, [\mathsf{node}_i]_1^n, i_{\mathsf{rt}}) \to (\mathsf{st}, \mathsf{ptr}_{\mathsf{rt}})$**.** On input a maximum number of nodes m, a list of data structure nodes $[\mathsf{node}_i]_1^n$, and an index i_{rt} for the "root" of the data structure in this list, ODS.Init converts the nodes in $[\mathsf{node}_i]_1^n$ to ODS nodes, initializes the server and outputs initial client state st and an ODS pointer $\mathsf{ptr}_{\mathsf{rt}}$ for the root.
- **Start:** $\mathsf{ODS.Start}(\texttt{mut}\ \mathsf{st}, \mathsf{ptr}_{\mathsf{rt}})$**.** On input the current client state st and the root's pointer $\mathsf{ptr}_{\mathsf{rt}}$, ODS.Start updates the state to use rt for future invocations of ODS.Access.
- **Access:** $\mathsf{ODS.Access}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{op}) \to \mathsf{res}$**.** On input the current client state st, and operation type op, ODS.Access outputs the operation result res (and updates the state st). There are four operation types.
  - **Read:** $\mathsf{op} = \texttt{read}(\mathsf{ptr})$ **and** $\mathsf{res} = \mathsf{node}$**.** Takes as input a pointer and outputs the node at the pointer.
  - **Insert** $\mathsf{op} = \texttt{ins}(\mathsf{node})$ **and** $\mathsf{res} = \mathsf{ptr}$**.** Takes as input a node to insert and outputs a pointer to it.
  - **Delete:** $\mathsf{op} = \texttt{del}(\mathsf{ptr})$ **and** $\mathsf{res} = \perp$**.** Takes as input a pointer to a node to delete and outputs $\perp$.
  - **Write:** $\mathsf{op} = \texttt{write}(\mathsf{node}, \mathsf{ptr})$ **and** $\mathsf{res} = \perp$**.** Takes as input a node to write and a pointer to it, and outputs $\perp$.
- **Finalize:** $\mathsf{ODS.Finalize}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{node}, \mathsf{bound}) \to \mathsf{ptr}_{\mathsf{rt}}$**.** On input current client state st, the (possibly updated) data structure root node, and an upper bound bound on the number of ORAM.ReadBlock operations to be performed, ODS.Finalize invokes ORAM.ReadBlock on dummy inputs enough times to make the total number of reads equal to bound, and outputs an updated root pointer $\mathsf{ptr}_{\mathsf{rt}}$.

We define security of ODS schemes in Appendix C.

IV. Oblivious sorted multimaps

Our *oblivious sorted multimap* (OSM) enables a client to outsource a certain type of key-value map to an untrusted server so that the map remains encrypted and, yet, the client can still perform search, insert, and delete operations with small cost (in latency and bandwidth) without revealing which key-value pairs of the map were accessed. This notion extends previous notions such as *oblivious maps* [55, 74].

The following sub-sections are organized as follows: in Section IV-A we define sorted multimaps, the data structure supported by OSM; in Section IV-B we define OSM schemes; in Section IV-C we informally describe our construction of an OSM scheme. We provide more details in Appendix A.

A. Sorted multimaps

A sorted multimap Map is a data structure that maps a key $\mathsf{k} \in \{0,1\}^{\ell_{\mathsf{k}}}$ to a (possibly empty) list of sorted and distinct values $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ with $\mathsf{v}_i \in \{0,1\}^{\ell_{\mathsf{v}}}$, denoted $\mathsf{Map}[\mathsf{k}]$. It supports the following operations.

- **Size:** $\mathsf{Map.Size}(\mathsf{k}) \to n$**.** On input a key k, Map.Size outputs the number of values in the list $\mathsf{Map}[\mathsf{k}]$.
- **Find:** $\mathsf{Map.Find}(\mathsf{k}, i, j) \to (\mathsf{v}_i, \ldots, \mathsf{v}_j)$**.** On input a key k, start index i, and end index j (no less than i), Map.Find outputs the values between locations i and j (included) in the list $\mathsf{Map}[\mathsf{k}]$; any requested location beyond the end of the list $\mathsf{Map}[\mathsf{k}]$ is answered with the value $\mathsf{v} := \perp$. (In particular, the answer always consists of $j - i + 1$ values.)
- **Insert:** $\mathsf{Map.Insert}(\mathsf{k}, \mathsf{v}) \to \perp$**.** On input a key k and value v, Map.Insert adds v to the list $\mathsf{Map}[\mathsf{k}]$ (if not already present), keeping its values sorted.
- **Delete:** $\mathsf{Map.Delete}(\mathsf{k}, \mathsf{v}) \to b$**.** On input a key k and value v, Map.Delete removes v from the list $\mathsf{Map}[\mathsf{k}]$, and outputs 1 if v was present and 0 if not.

B. Definition of OSM schemes

An OSM scheme is a tuple $\mathsf{OSM} := (\mathsf{Init}, \mathsf{Find}, \mathsf{Insert}, \mathsf{Delete}, \mathcal{S})$ that contains algorithms for two parties, the *OSM client* and the *OSM server*. The OSM client uses Init to initialize the scheme with a given sorted multimap; subsequently, he may use Find to retrieve sublists associated with a given key, as well as use Insert and Delete to modify such lists. All of these algorithms require interaction with the OSM server, which runs the interactive algorithm $\mathcal{S}$. We represent this interaction by treating $\mathcal{S}$ as an oracle.

- **Initialization:** $\mathsf{OSM.Init}^{\mathcal{S}}(\mathsf{Map}) \to \mathsf{st}$**.** On input a sorted multimap Map, OSM.Init interacts with $\mathcal{S}$ in order to store at $\mathcal{S}$ an "encryption" of Map, and then outputs a local state st (to be stored by the OSM client).

The semantics of the following operations are the same as the corresponding Map operations, where Map is the sorted multimap stored encrypted at $\mathcal{S}$ by OSM.Init.

- **Size:** $\mathsf{OSM.Size}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{k}) \to n$**.** On input local state st and key k, OSM.Size interacts with the server $\mathcal{S}$ and then outputs an integer n.
- **Find:** $\mathsf{OSM.Find}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{k}, i, j) \to (\mathsf{v}_i, \ldots, \mathsf{v}_j)$**.** On input local state st, key k, start index i, and end index j (no less than i), OSM.Find interacts with $\mathcal{S}$ and then outputs a list of $j - i + 1$ values $(\mathsf{v}_i, \ldots, \mathsf{v}_j)$.
- **Insert:** $\mathsf{OSM.Insert}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v})$**.** On input local state st, key k, and value v, OSM.Insert interacts with $\mathcal{S}$.
- **Delete:** $\mathsf{OSM.Delete}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v}) \to b$**.** On input local state st, key k, and value v, OSM.Delete interacts with $\mathcal{S}$ and then outputs a bit b.

Correctness. Correctness of an OSM scheme is defined via two experiments. In the real world experiment, the adversary has access to an oracle $\mathbf{C}_{\mathsf{Real}}$ that implements the OSM client. In the ideal world experiment, the adversary has access to an oracle $\mathbf{C}_{\mathsf{Ideal}}$ that implements a (plaintext) sorted multimap. Both oracles expose to the adversary the same interface of possible queries (`Init`, `Size`, `Find`, `Insert`, `Delete`). See Fig. 3 for details on how $\mathbf{C}_{\mathsf{Ideal}}$ and $\mathbf{C}_{\mathsf{Real}}$ generate their answers. An OSM scheme is *correct* if no efficient adversary $\mathcal{A}$ can distinguish between these oracles, i.e., $\mathcal{A}^{\mathbf{C}_{\mathsf{Real}}}$ and $\mathcal{A}^{\mathbf{C}_{\mathsf{Ideal}}}$ are computationally indistinguishable as distributions.

	Correctness		Security	
	$\mathbf{C}_{\mathsf{Real}}$	$\mathbf{C}_{\mathsf{Ideal}}$	$\mathbf{S}_{\mathsf{Real}}$	$\mathbf{S}_{\mathsf{Ideal}}$
$\mathtt{Init}(m, \mathsf{Map})$	store st $\leftarrow \mathsf{OSM.Init}^{\mathcal{S}}(m, \mathsf{Map})$	store Map	store st $\leftarrow \mathsf{OSM.Init}^{\mathcal{S}}(\mathsf{Map})$	store st $\leftarrow \mathsf{Sim.Init}^{\mathcal{S}}(m, \ell_{\mathsf{k}}, \ell_{\mathsf{v}})$
$\mathtt{Size}(\mathsf{k})$	$\mathsf{OSM.Size}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}) \to n$	$\mathsf{Map.Size}(\mathsf{k}) \to n$	$\mathsf{OSM.Size}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}) \to n$	$\mathsf{Sim.Size}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st})$
$\mathtt{Find}(\mathsf{k}, i, j)$	$\mathsf{OSM.Find}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, i, j) \to \vec{\mathsf{v}}$	$\mathsf{Map.Find}(\mathsf{k}, i, j) \to \vec{\mathsf{v}}$	$\mathsf{OSM.Find}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, i, j) \to \vec{\mathsf{v}}$	$\mathsf{Sim.Find}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, j - i + 1)$
$\mathtt{Insert}(\mathsf{k}, \mathsf{v})$	$\mathsf{OSM.Insert}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v})$	$\mathsf{Map.Insert}(\mathsf{k}, \mathsf{v}) \to \bot$	$\mathsf{OSM.Insert}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v})$	$\mathsf{Sim.Insert}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st})$
$\mathtt{Delete}(\mathsf{k}, \mathsf{v})$	$\mathsf{OSM.Delete}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v}) \to b$	$\mathsf{Map.Delete}(\mathsf{k}, \mathsf{v}) \to b$	$\mathsf{OSM.Delete}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v}) \to b$	$\mathsf{Sim.Delete}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st})$

Figure 3: Correctness and security oracles for an OSM scheme. Oracle outputs are highlighted.

Security. Security of an OSM scheme is defined via two experiments. In the real experiment, the adversary has access to an oracle $\mathbf{S}_{\mathsf{Real}}$ that runs the OSM client and outputs nothing. In the ideal experiment, the adversary has access to an oracle $\mathbf{S}_{\mathsf{Ideal}}$ that runs a simulator Sim that receives only certain subsets of the inputs. Both oracles expose to the adversary the same interface of queries (`Init`, `Size`, `Find`, `Insert`, `Delete`), and the adversary gets to observe *all communication of the oracles with the server* $\mathcal{S}$ and *all server state*. See Fig. 3 for details on how $\mathbf{S}_{\mathsf{Real}}$ and $\mathbf{S}_{\mathsf{Ideal}}$ generate their answers. An OSM scheme is *secure* if no efficient adversary $\mathcal{A}$ can distinguish between these oracles, i.e., $\mathcal{A}^{\mathbf{S}_{\mathsf{Real}}}$ and $\mathcal{A}^{\mathbf{S}_{\mathsf{Ideal}}}$ are computationally indistinguishable as distributions.

C. Construction of an OSM scheme

First, we explain why oblivious maps from prior works [55, 74] are not suitable for realizing OSM schemes; then, we informally describe our construction of an OSM scheme. In Appendix A we provide the detailed construction, including pseudocode and proofs of correctness and security.

Insufficiency of oblivious maps. Suppose that, given a sorted multimap that associates keys k to sorted lists $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$, we construct a (standard) map by associating new keys $(\mathsf{k}, 1), \ldots, (\mathsf{k}, n)$ to the values $\mathsf{v}_1, \ldots, \mathsf{v}_n$ respectively. Searching is simple: to find the values of a key k at indices $i, \ldots, j$, we fetch from the map the values associated to $(\mathsf{k}, i), \ldots, (\mathsf{k}, j)$. However, if we want to insert, for a key k, a new value v' that is smaller than all other values in the list of k, we need to shift every key "to the right": (k, t) must become $(\mathsf{k}, t+1)$ for every $t \in \{1, \ldots, n\}$. This entails $\Omega(n)$ oblivious accesses, which is expensive; even worse, doing so leaks the size of the list. Hiding this leakage would require padding to the worst-case size of a list, thereby making this idea even more expensive. (Indeed, some keys might have very large lists, e.g., proportional to the number of documents in a database!) Our approach below sidesteps these issues by implicitly constructing a (sub-)tree over each key's list, ensuring that insertions have logarithmic complexity and reveal only the total number of key-value pairs in the map (and thus do not reveal the list's size).

Our OSM construction. We directly construct an OSM scheme in the *oblivious data structure* (ODS) framework of [74], summarized in Section III-B. This involves two steps: (i) construct a *plaintext* data structure having *tree-like* memory accesses, and (ii) replace its memory accesses with oblivious counterparts defined by the ODS framework. Recall (from Definition 1) that an access pattern is tree-like if every data structure operation starts from a distinguished root node, and the graph arising from following pointers during its execution forms a *tree* (there is a unique path from the root node to all other nodes). Below we describe at a high level how to complete the first of the two steps (the more interesting one).

A tree-like sorted multimap. We extend *AVL search trees* to store multiple values for the same key, and borrow techniques from *order statistic trees* to efficiently retrieve the i-th value of a given key.

An AVL search tree is a balanced binary search tree that implements a simple map from keys to values; it supports searches, inserts, and deletes in worst-case logarithmic time (via its worst-case logarithmic height). Each node in the tree stores a key and a value; each key appears in at most one node. (In particular, when inserting a key-value pair (k, v), if a node with key k already exists, its value is simply overwritten to v.)

In contrast, we need a sorted multimap, which maps a key to a *sorted list* of values. We still consider nodes that store a key and a value (and other information described below), but now allow multiple nodes to share the same key. Some operations almost directly follow from AVL trees: (a) when inserting a key-value pair (k, v) not already in the tree, we insert a new new node for (k, v) via AVL tree insertion while treating the pair (k, v) as a key; (b) deletions mirror insertions; (c) search can be modified to retrieve the list of values associated with a key (instead of just a single value). However, the foregoing modifications fall short of enabling retrieval of arbitrary sublists of the list corresponding to a key (without retrieving the full list), as needed in a sorted multimap. To do this, we incorporate techniques from order statistic trees, as we discuss next.

In an order statistic binary search tree, each node also stores the number of nodes in each (left and right) child subtree. This information can be used to efficiently find the node with the i-th smallest key and to augment AVL insertions and deletions to maintain this information. (See [15, Chapter 14].)

We modify this approach to obtain AVL-based sorted multimaps in which one can efficiently find a key's i-th value, and thus also any sublist of values. Informally, a node with key k stores the number of nodes *that also have key* k in each child subtree (rather than all nodes, potentially with different keys, in those subtrees); see Fig. 4. Straightforward modifications to the insertion and deletion procedures ensure that this information is maintained across operations. It is not hard to verify that the resulting data structure has tree-like accesses, as required.

Correctness of the foregoing approach is reducible to the correctness of AVL search trees and order statistic trees (our

modifications are minor). Insertions and deletions take time $O(\log(n))$, while finding the i-th through j-th values for a key requires time $O(\log(n) + j - i)$.

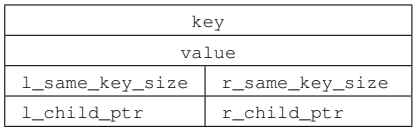

Figure 4: Information stored in a node in our OSM construction.

Below we summarize operations for the sorted multimap (with tree-like accesses) outlined above. Due to space reasons, we omit a description of deletions; analogously to inserts, these can be achieved via suitable modifications of AVL tree deletion.

- $\mathsf{Size}(\mathsf{k})$**:** Depth-first search for k until the first k-node, which stores the total number of k-nodes in the tree, and return this number. (Compare: in order statistic trees, the root stores the number of nodes in the tree.)
- $\mathsf{Find}(\mathsf{k}, i, j)$**:** Find paths to the i-th and j-th k-nodes and fetch all k-nodes in the subtree bounded by these.
  1) *Find path to s-th* k*-node:* Find the s-th smallest node in the order statistic subtree consisting only of k-nodes. That is, run a depth-first search for k as follows. When visiting a k'-node with $\mathsf{k}' \neq \mathsf{k}$, recursively search the left subtree (if $\mathsf{k} < \mathsf{k}'$) or right subtree (if $\mathsf{k} > \mathsf{k}'$). When visiting a k-node, letting l be the number of k-nodes in its left subtree: if $s < l$, recursively search the left subtree; if $s = l$, output the path to the current node; if $s > l$, set $s := s - (l + 1)$ and recursively search the right subtree.
  2) *Retrieve nodes:* Find the node at which the paths to the i-th and j-th k-nodes diverge and run a breadth-first search from this node by considering only k-nodes. That is, add to the BFS queue only nodes that are no less than the i-th k-node and no greater than the j-th k-node (to compare two nodes, first compare their keys and, if equal, compare their values). Return the resulting set of k-nodes.
- $\mathsf{Insert}(\mathsf{k}, \mathsf{v})$**:** Search for the node where insertion must occur; if this node already exists, then we are done; otherwise, create a new node and make it the appropriate child of the previously-visited node. Then retrace the path from the inserted node back to the root, rebalancing as needed. The rebalancing procedure is a modification of order statistic AVL trees that ensures that the size information of visited k-nodes is correctly updated (at each step of retracing, we store the number of k-nodes seen thus far and, if the node at the current step is a k-node, then we rebalance and update the size of the appropriate child subtree).

We remark that our OSM construction above (coupled with padding as discussed below) already provides a search index that does not leak access patterns *without relying* on hardware enclaves: the client stores the OSM state locally and interacts with the remote server over the network for each OSM operation. Leveraging hardware enclaves will enable better performance and support for multiple users.

V. Doubly-oblivious primitives

We describe how to design *client algorithms* for Path ORAM, ODS, and OSM that are *themselves* oblivious. We refer to the resulting cryptographic primitives as *doubly-oblivious* because not only are the client's accesses to the server's memory oblivious, but also the client's accesses to its own local memory are oblivious. This requirement arises when running the ORAM client inside a hardware enclave at the server because, as discussed in Section II-A, the enclave does *not* hide access patterns to this encrypted memory. Path ORAM (Section III-A), ODS (Section III-B), and OSM schemes (Section IV) already guarantee that the client's accesses to external memory are oblivious, but in current constructions the client's accesses to internal memory are not oblivious.

One can trivially make accesses to internal memory oblivious by replacing each such access with a linear scan. However, such an approach yields expensive solutions, and the challenge lies in designing alternatives that, ideally, are almost as efficient as the original client algorithm.

In the next few sub-sections we explain how we design *efficient* data-oblivious *client* algorithms for Path ORAM (Section V-A), ODS (Section V-B), and OSM schemes (Section V-C). Table I summarizes the costs of the (standard) client algorithms and the data-oblivious variants that we use. Our experiments demonstrate that the overheads that arise from double obliviousness are small (see Sections VI-A and VI-B).

A. Doubly-oblivious RAM

We now provide intuition for our construction of *path doubly-oblivious RAM* (Path DORAM). We begin by describing our construction of DORAM.ReadBlock and DORAM.Evict, and then describe our DORAM.Init algorithm. Pseudocode for these algorithms is provided in Appendix D.

In the rest of this section, we denote the number of buckets at the server by N, the capacity of each bucket (in blocks) by C, and the block size (in bits) by B. We use a linear-time function $\mathsf{ObliSwap}(b, x, y)$ that obviously swaps x and y if $b = 1$. Our algorithms also invoke Batcher's oblivious odd-even merge sort, which has time complexity $O(n \log^2 n)$ to sort n items.

Naive approach. A straightforward approach to achieve double-obliviousness is to replace suitable sub-routines (e.g., a binary search) with linear scans. Namely, ReadBlock fetches the required path, inserts all blocks on that path into its stash, and returns the requested block by linearly scanning the stash; Evict constructs buckets for the path of a leaf lf as follows.

> Initialize `mut` $\mathsf{inserted} = 0$. Then for each block bl in the stash Stash:
> For each bucket bu on the path of lf (ordered from leaf upwards):
> 1) Let $\mathsf{is\_ancestor} = 1$ if bu is on the path to bl's leaf.
> 2) For each $i \in \{1, \ldots, C\}$:
> a) Let $\mathsf{cond} = \mathsf{bu}[i].\mathsf{is\_dummy} \wedge \mathsf{is\_ancestor} \wedge \neg\, \mathsf{inserted}$.
> b) $\mathsf{ObliSwap}(\mathsf{cond}, \mathsf{bl}, \mathsf{bu}[i])$.
> c) Set $\mathsf{inserted} := \mathsf{cond} \vee \mathsf{inserted}$.

Namely, for each block bl in Stash, Evict scans the list of buckets to be written back, checks if bl can go in one of these, and obliviously writes it to that bucket if so. When evicting n paths after n ReadBlock calls, the stash contains roughly $S = nC \log(N)$ blocks, and so this naive Evict procedure has time complexity $O(B \cdot S \cdot nC \log N) = O(n^2 BC^2 \log^2 N)$.

Scheme	Algorithm			Client type	
				Standard	Doubly-oblivious
Path ORAM	Init	T_{i_1}	$=$	$O(CN \log CN)$	$O(CN \log^2(CN) \log N)$
	ReadBlock	T_{r}	$=$	$O(C \log N \log S)$	$O(C \log N + \mathsf{ebl})$
	Evict	T_{e}	$=$	$O(BS \log N \log(\mathsf{ibu}))$	$O(\mathsf{ebl}^2 \log N + BS \log^2 BS)$
ODS	Init	T_{i_2}	$=$	$O(CN \log(CN)) + T_{\mathsf{i}_1}$	
	Start	T_{s}	$=$	$O(1)$	
	Access	T_{a}	$=$ $\begin{cases} \notin \text{cache} \\ \in \text{cache} \end{cases}$	$O(\log(\mathsf{ca})) + T_{\mathsf{r}}$ $O(\log(\mathsf{ca}))$	$O(\mathsf{ca}) + T_{\mathsf{r}}$ $O(\mathsf{ca})$
	Finalize	$T_{\mathsf{f}}(n)$	$=$	$O(\mathsf{ca}) + T_{\mathsf{e}}$ (with $\mathsf{ebl} = n$)	
OSM	Init			$O(CN \log(CN)) + T_{\mathsf{i}_2}$	
	Find(m)			$(2h+m) \cdot T_{\mathsf{a}} + T_{\mathsf{f}}(2h+m)$	
	Size			$h \cdot T_{\mathsf{a}} + T_{\mathsf{f}}(h)$	
	Insert			$(h+1) \cdot T_{\mathsf{a}} + T_{\mathsf{f}}(h+1)$	

B	block size (in bits)
N	server size (in buckets)
C	bucket size (in blocks)
ebl	size of ExplicitBlocks (in blocks)
ibu	size of ImplicitBuckets (in buckets)
S	$= \mathsf{ebl} + C \cdot \mathsf{ibu}$, stash size (in blocks)
ca	size of ODS cache ($\leq$ ebl)
h	$= 1.44 \log(CN)$, worst-case height of AVL tree with CN nodes

Table I: Comparison of standard and doubly-oblivious client algorithms for Path ORAM, ODS, and OSM. Whenever a client algorithm invokes a subroutine, the running time of the subroutine is for the corresponding client type. For example, DODS invokes Path DORAM algorithms.

Saving a multiplicative factor of C. We improve upon naive eviction by splitting eviction into two steps, saving a multiplicative factor of C. In the first step, we *assign* blocks to buckets; in the second step, we *write* blocks to buckets.

- *Block assignment.* We initialize a linear-scan "bucket fullness" map BuFu from bucket nodes (i.e., identifiers of the bucket's location in the ORAM tree, not the full bucket) to the number of blocks in those buckets (i.e., the fullness of the bucket) so that entries in BuFu are sorted by their nodes from leaf to root. Then, for each block bl in Stash, we scan the list of buckets to be written back (in order from leaf to root), and update BuFu as follows. If bl should be written to a bucket bu, then we increment BuFu[bu.node], and set bl.node := bu.node. Otherwise, we perform dummy operations. This step has time complexity $O(S \cdot n \log N) \approx O(n^2 C \log^2 N)$.
- *Bucket construction.* We obliviously sort Stash to group together blocks with the same node, and construct buckets out of these. To hide how many blocks are assigned to buckets, we pad Stash with dummy blocks. Any unassigned blocks are re-added to Stash. This step has time complexity $O(B \cdot S \log^2(S)) \approx O((nBC \log N) \log^2(nBC \log N))$.

The overall complexity of this eviction procedure, which we call $\mathsf{Evict}_{\mathsf{s}}$, is $O(n^2 C \log^2 N + nBC \log N \log^2(nBC \log N)) \approx O(nC \log N(n \log N + B \log^2(nBC \log N)))$.

Processing only requested blocks. We further improve on the above via the following insight. Even though a user invokes ReadBlock to request only one block, $O(\log N)$ additional blocks are implicitly fetched and added to the stash, which means that Evict has to process these additional blocks when constructing buckets. Our new eviction procedure $\mathsf{Evict}_{\mathsf{f}}$ gains in time complexity by separately processing explicitly requested blocks (henceforth *explicit blocks*) and implicitly fetched blocks (henceforth *implicit blocks*), as follows.

In more detail, we modify ReadBlock to scan the list of fetched buckets, (obliviously) remove the block of interest (say bl), and replace bl with a dummy block. It then adds bl to a list ExplicitBlocks of previously requested explicit blocks, and adds the updated list of fetched buckets to a list of previously requested buckets ImplicitBuckets. For eviction, note that blocks in (buckets in) ImplicitBuckets are already bucketed correctly: they are already in buckets on the path to their leaves. Hence, we can skip re-bucketing these blocks, and can focus on re-bucketing only the blocks in ExplicitBlocks. Concretely, $\mathsf{Evict}_{\mathsf{f}}$, like $\mathsf{Evict}_{\mathsf{s}}$, proceeds in two phases:

- *Block assignment.* We use ImplicitBuckets to compute a pre-populated bucket fullness map BuFu. Then, for every block in ExplicitBlocks, we update BuFu as in $\mathsf{Evict}_{\mathsf{s}}$ and assign each block to a bucket. This step has time complexity $O(|\mathsf{ExplicitBlocks}| \cdot n \log N)$.
- *Bucket construction.* We proceed as in $\mathsf{Evict}_{\mathsf{s}}$.[1]

The overall time complexity of $\mathsf{Evict}_{\mathsf{f}}$ is thus $O(nC \log N(n/C + B \log^2(nBC \log N)))$, which saves a factor of $\frac{\log N}{C}$ compared to $\mathsf{Evict}_{\mathsf{s}}$.

However, this efficiency gain comes at the expense of a lower eviction rate; $\mathsf{Evict}_{\mathsf{f}}$ evicts fewer blocks than $\mathsf{Evict}_{\mathsf{s}}$. This is because $\mathsf{Evict}_{\mathsf{f}}$ only re-assigns explicit blocks, and does not shuffle implicit blocks. Furthermore, each such explicit block can only be assigned to a slot vacated by a previously fetched explicit block. Together, these constraints reduce the rate of stash eviction. As a countermeasure, our final construction of DORAM.Evict invokes $\mathsf{Evict}_{\mathsf{f}}$ in the common case (for speed), but invokes $\mathsf{Evict}_{\mathsf{s}}$ at fixed intervals (to empty out the stash). Empirical evidence from our experiments, suggests that this interval can be a fixed constant as small as 3.

Initialization. There is a naive doubly-oblivious initialization strategy: given the initial list of n blocks $[\mathsf{bl}_i]_1^n$, individually insert every block into the stash, and then use DORAM.Evict to evict each block from the stash. However this method requires time $O(nCN)$. When the server is at capacity ($CN \approx n$), this grows quadratically with n, and is too large for the database sizes that we consider. We address this problem by designing a new doubly-oblivious initialization strategy that has time complexity $O(CN \log^3 N)$, which enables us to efficiently handle databases with tens of millions of records.

Our algorithm proceeds layer by layer in the tree. Within each layer, it proceeds similarly to Evict: it first assigns blocks to buckets, and then obliviously sorts these blocks to group them

[1] In our implementation, instead of sorting, we write blocks to buckets via linear scans (as in the naive approach). While this is asymptotically worse, for our use cases this method is concretely faster.

into buckets. In more detail, for a given tree layer, DORAM.Init first obliviously sorts $[\mathsf{bl}_i]_1^n$ by the blocks' tree nodes (initially, just each block's assigned leaf). Next, it scans the list to compute the fullness of each bucket, and assigns each block to a bucket according to this fullness. Finally, it constructs buckets by obliviously sorting $[\mathsf{bl}_i]_1^n$ so that blocks with the same tree nodes are together (as before, we pad with enough dummy blocks before to hide the number of bucketed blocks). To proceed to the next layer, it sets the nodes of unassigned blocks to be the parent of their current nodes. Since there are $\log(N)$ layers, and each layer requires two oblivious sorts and a linear scan, this algorithm has time complexity $O(CN\log^3(N))$.

Final construction. We now summarize our final construction of Path DORAM; for detailed pseudocode see Appendix D.

- **Initialization:** $\mathsf{ORAM.Init}^{\mathcal{S}}(m, [\mathsf{bl}_i]_1^n) \to \mathsf{st}$**.** Proceed layer-by-layer in the ORAM tree. In each layer, first assign blocks to buckets, and then obliviously sort these blocks to group them into buckets.
- **Read a block:** $\mathsf{DORAM.ReadBlock}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{bid}, \mathsf{lf}) \to \mathsf{bl}$**.** Fetch buckets on the path to lf. Scan this list to obliviously replace the block bl having identifier bid with a dummy block. Insert the modified buckets into ImplicitBuckets, and insert bl into ExplicitBlocks. Finally, output bl.
- **Eviction:** $\mathsf{DORAM.Evict}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, [\mathsf{lf}_i]_1^n)$**.** Given a integer t fixed at setup, store in st a counter $c \in \mathbb{Z}_t$. If $c = 0 \bmod t$, invoke $\mathsf{Evict_s}$; else, invoke $\mathsf{Evict_f}$. Increment c.

Note that we have not specified how to obliviously access the client's position map because this can be achieved by standard recursion techniques [64] or by using the ODS framework [74].

Stashless ORAM. The primary obstacle we faced in designing Path DORAM was creating a doubly-oblivious stash eviction procedure. To avoid this trouble, one might instead think to use a stateless ORAM scheme [28]. However, this idea does not help because all such schemes still require working space to store blocks *between reads and eviction*; the adjective "stateless" only describes *permanent* client storage. Obliviously accessing this working space is expensive when it is large, but Path ORAM only requires $\mathrm{polylog}(N)$ working space, compared to space n^c for $0 < c < 1$ for other schemes.

B. Doubly-oblivious data structures

We describe a framework for *doubly-oblivious data structures* (DODS). We modify the existing framework for singly-oblivious data structures (ODS, see Section III-B) to: (i) use Path DORAM (see prior sub-section), instead of merely Path ORAM, as a building block; and (ii) leverage other ideas for efficiency. Details follow.

The ODS client. We briefly recall the construction of the singly-oblivious data structure framework of [74]. The client realizes a data structure operation by running ODS.Start once, ODS.Access some number of times, and ODS.Finalize once; throughout, the client maintains a cache with fetched nodes. Whenever the client is queried on a node (via ODS.Access), it looks for the node in the cache and returns it if there; otherwise, the client performs an ORAM.ReadBlock operation to fetch the node from the server, adds it to the cache, and returns it. Since the number of ORAM.ReadBlock operations may be data dependent, ODS.Finalize pads this number to a data-independent (worst-case) number with dummy ORAM.ReadBlock operations, thereby ensuring that accesses to the (external) memory at the server are oblivious.

Naive approach. A naive approach to make the ODS client doubly-oblivious is to simply replace the underlying ORAM scheme with a DORAM scheme and replace the cache with an oblivious one. However, this does not suffice: whether the returned node is fetched from the cache or the server depends on the queried node, and an adversary observing accesses to internal memory can distinguish between the two cases, *even if accesses to external memory are oblivious and their number is data independent*. A straightforward fix is to *always* perform a (possibly dummy) DORAM.ReadBlock operation whenever DODS.Access is invoked, regardless of whether the queried node is cached or not. However, while doubly-oblivious, this approach harms efficiency since the ODS client now may perform unnecessary DORAM.ReadBlock operations.

Our approach. We avoid unnecessary dummy ReadBlock operations via the observation that, in certain cases, the adversary *can predict* if a node is fetched from the cache.

For example, in an AVL tree insertion, the rebalancing phase only visits nodes that have been previously visited, and so are in the cache. In our doubly-oblivious sorted multimap (see Section V-C), we design insertion so that rebalancing begins only after a fixed number of nodes have been accessed in the previous phase, so the adversary can predict when rebalancing begins, and thus also that the nodes accessed then are cached.

In such cases, we can forgo the dummy DORAM.ReadBlock operation and gain efficiency. When we are not in such a case (the information of whether a node is in the cache is not public), we fall back to the aforementioned simple approach (of always performing a dummy DORAM.ReadBlock).

Our framework for *doubly-oblivious data structures* (DODS) formalizes the foregoing ideas, most notably by exposing a richer interface that enables fine-grained control over memory accesses to internal memory. Below we summarize the interface and implementation of each algorithm of this framework.

- **Initialization:** $\mathsf{DODS.Init}^{\mathcal{S}}(m, [\mathsf{node}_i]_1^n, i_{\mathsf{rt}}) \to (\mathsf{st}, \mathsf{ptr}_{\mathsf{rt}})$**.** Equals ODS.Init, but calls DORAM.Init, not ORAM.Init.
- **Start:** $\mathsf{DODS.Start}(\mathtt{mut}\ \mathsf{st}, \mathsf{ptr}_{\mathsf{rt}})$**.** Equals to ODS.Start.
- **Access:** $\mathsf{DODS.Access}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{op}) \to \mathsf{res}$**.**
  Input now has the form "$\mathsf{op}(\mathsf{data}, \mathsf{dummy}, \mathsf{isCached})$". There are four cases:
  - $\mathsf{dummy} = 1, \mathsf{isCached} = ?$: Perform dummy ReadBlock.
  - $\mathsf{dummy} = 1, \mathsf{isCached} = 1$: Fetch dummy node from the cache without dummy ReadBlock.
  - $\mathsf{dummy} = 0, \mathsf{isCached} = 1$: Fetch actual node from cache.
  - $\mathsf{dummy} = 0, \mathsf{isCached} = ?$: Perform ReadBlock to fetch real (non-dummy) node. If queried node is already cached, perform dummy ReadBlock.

- **Finalize:** $\mathsf{DODS.Finalize}^{S}(\texttt{mut}\ \mathsf{st}, \mathsf{node}, \mathsf{bound}) \rightarrow \mathsf{ptr}_{\mathsf{rt}}$. Similar to ODS.Finalize, except that it *does not* perform additional dummy operations. Instead, it checks that the number of DORAM.ReadBlock operations thus far equals bound. Satisfying this condition is the responsibility of the data structure designer. (Compare: in ODS the designer only has to specify the bound; padding occurs automatically.)

C. Doubly-oblivious sorted multimaps

We construct doubly-oblivious sorted multimaps (DOSM). We modify our construction of singly-oblivious sorted multimaps (OSM, see Section V-B) to: (i) use DODS (see prior sub-section), instead of merely ODS, as a building block; and (ii) leverage the fine-grained interface of DODS for improved efficiency. Details follow.

Naive approach. A naive approach to make the OSM client doubly-oblivious is to simply replace the underlying ODS framework with the DODS framework. However, this does not suffice: the OSM client maintains internal state (outside the ODS framework) and its accesses to it are data dependent. For example, OSM.Insert uses a depth-first search to find the insertion location, and this search terminates as soon as the location is found, which depends on the key-value pair to insert. The adversary can learn some information about this pair because it can observe when this termination occurs (after this point all accesses correspond to cache accesses rather than external memory accesses).

Our construction. To eliminate such leakage, we identify data-dependent sub-procedures of our algorithms, and appropriately pad out the number of accesses made in these procedures to worst-case bounds that depend only on the number of key-value pairs in the map. For example, when an algorithm initiates a depth-first search, we ensure that the search terminates after accessing exactly $1.44\log(n)$ (real or dummy) nodes, which is the worst-case height of an AVL tree with n nodes.

Next, we design our algorithms so that that we can always predict whether or not a given dummy access needs to return a cached node. We can then take advantage of the fine-grained DODS interface to avoid unnecessary dummy operations. Below we summarize our doubly-oblivious construction (again omitting deletions for space reasons, as in Section IV-C).

- DOSM.Init: Equals OSM.Init, but calls DODS.Init instead of ODS.Init.
- DOSM.Size: Instead of halting the depth-first search when the first k-node is found, perform additional DODS.Access calls with input $\texttt{read}(\mathsf{dummy} = 1, \mathsf{isCached} = ?, \mathsf{k})$ to ensure that DORAM.ReadBlock is invoked $1.44\log(n)$ times in total (the worst-case height of an AVL tree with n nodes).
- DOSM.Insert: Modify the depth-first search used to find the insertion location so that DORAM.ReadBlock is invoked $1.44\log(n)$ times (as in DOSM.Size above). Also, in the retracing step, modify the rebalancing procedure to perform the same (real or dummy) operations regardless of the type of rebalancing required.
- DOSM.Find: Recall that OSM.Find has two steps: find paths to the i-th and j-th k-nodes (node_i and node_j from here on) and fetch all k-nodes in the subtree bounded by these. We describe how both steps can be made doubly-oblivious.
  1) *Find path to s-th* k*-node:* Modify the depth-first search so that DORAM.ReadBlock is invoked $1.44\log(n)$ times (as in DOSM.Size above).

  It is important to ensure that retrieving the path to node_j after retrieving the path to node_i does not reveal where the two paths diverge. This happens when the search retrieves common nodes from the cache and not the server, and is prevented by invoking DODS.Access with input $\texttt{read}(\mathsf{dummy} = 0, \mathsf{isCached} = ?, \mathsf{k})$ (this ensures that a ReadBlock is always performed).

  2) *Retrieve required nodes:* Find the node at which the paths to node_i and node_j diverge (as in OSM.Find), and then, instead of performing a simple breadth-first search from this node, run a modified breadth-first search that (a) uses an oblivious priority queue instead of a simple first-in-first-out queue, and (b) terminates after visiting $2 \cdot 1.44\log(n) + j - i$ nodes. Initialize this queue with (real and dummy) keys of the nodes on paths to node_i and node_j, in that order. When fetching the next node from the queue, add the key of the appropriate child to the queue with an "exploration priority" that decides when the node gets visited. We assign priorities so that nodes on the bounding paths are visited first, and k-nodes in the intersection afterwards.

VI. Evaluation and applications

Implementation. We realized singly- and doubly-oblivious versions of Oblix using $\sim 10\,\mathrm{K}$ lines of Rust code, split across libraries for singly- and doubly-oblivious Path ORAM, ODS, and OSM.

Evaluation. We evaluate Oblix via a set of benchmarks (Sections VI-A and VI-B) and via three applications: (i) private contact discovery for the Signal messaging service (Section VI-C); (ii) private retrieval of public keys in Key Transparency (Section VI-D); (iii) oblivious searchable encryption (Section VI-E). In each application, our results show that Oblix is competitive with, and sometimes also improves upon, alternate approaches with similar security guarantees. Overall, our work shows that ORAM-based techniques, often eschewed for their perceived large costs, can scale to large databases (tens of millions of records) and can be effectively applied to concrete problem domains.

We emphasize that this paper focuses on achieving low *latency*, and so our experiments focus on that. In many settings *throughput* is also important, and we leave to future work the problem of achieving high throughput as well. Our techniques ultimately leverage properties of Path ORAM, for which strong concurrency properties, exemplified in systems such as TaoStore [56], are known. We thus believe that improving throughput is an exciting, and potentially viable, future project.

Experimental setup. All experiments use a server with an Intel Xeon E3-1230 v5 CPU at $3.40\,\mathrm{GHz}$ with 8 logical cores, running Ubuntu 16.04. The CPU supports the Intel SGX v1

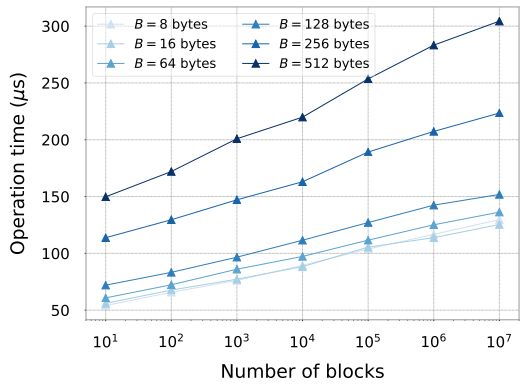

Figure 5: Latency of Path DORAM operations with an increasing number of initial blocks, across different block sizes B.

instruction set, and the total memory available to enclaves is limited to around $94\,\mathrm{MB}$. In experiments with Signal and Key Transparency, we initialize Oblix with the maximum number of key-value pairs that fit within memory, which is $64\,\mathrm{GB}$ in our testbed. We note that production servers are typically equipped with larger memory sizes; we therefore extrapolate the performance of Oblix for larger database sizes as well. Further, since the size of key-value pairs differs across applications, we configure the Path ORAM implementation underlying Oblix with a different block size per application. Finally, before each experiment, we warm up the ORAM stash via dummy requests in order to capture steady-state performance of Oblix.

A. Path DORAM microbenchmarks

We begin by evaluating the performance of our Path DORAM scheme (see Section V-A). Recall that during initialization, DORAM.Init is provided as input a maximum storage size m (in blocks), and a list of n initial blocks (with $n \leq m$). We evaluate the performance of a single operation (a ReadBlock followed by a Evict) in our DORAM scheme for $n = m \in \{10^1, \ldots, 10^7\}$ and for block sizes from 8 to 512 bytes. In Fig. 5, we report the average latency over 1000 operations; this latency is between tens to hundreds of microseconds.

Comparison with ZeroTrace. To put these numbers into perspective, we compare the performance of our scheme with that of ZeroTrace [57], which also implements a DORAM scheme within a hardware enclave. We provide a qualitative comparison between the two systems in Section VII; here, we focus on performance. The source code of ZeroTrace is not publicly available, and so we can only compare our results with the ones reported in the paper. However, both our testbeds use machines of similar capabilities.

Unlike Oblix, which uses the ODS framework to outsource the client's position map, ZeroTrace recursively stores the position map in smaller ORAMs. Each ZeroTrace ORAM operation thus requires recursive position map lookups. We estimate our DORAM scheme's performance in this setting by measuring the access times for each level of recursion and taking their sum. Our findings underscore the efficiency of our Path DORAM protocols compared to ZeroTrace: with 10^7 blocks and a block size of 8 bytes, an ORAM operation in Oblix takes $0.47\,\mathrm{ms}$ compared to ZeroTrace's $1.22\,\mathrm{ms}$ to $1.32\,\mathrm{ms}$ (based on the choice of the underlying ORAM scheme), representing a speedup of $\sim 2.5\times$. The gap widens further as the block size increases: for a block size of 512 bytes, Oblix takes $0.54\,\mathrm{ms}$ and is $4.5\times$ to $6.5\times$ faster than ZeroTrace.

B. DOSM microbenchmarks

We evaluate the latency of searches and inserts in our doubly-oblivious OSM scheme (see Section V-C). Our experiments show that latency is a few milliseconds, even when the database contains millions of key-value pairs.

Searches. The cost of a search query depends on (i) the total number of key-value pairs, and (ii) the number of values requested for the queried key. We experimentally measure latency as a function of these parameters, and report the average latency across 100 iterations. All experiments use keys and values of 8 bytes each, and use an underlying Path ORAM implementation with a block size of 160 bytes.

- *Increasing the number of key-value pairs.* We initialize an OSM scheme with up to 2^{24} key-value pairs. We then issue search queries for random keys, requesting a single value per key. Fig. 6 shows that search time is logarithmic in the number of key-value pairs. Moreover, even with 2^{24} key-value pairs, search time remains low at $4.4\,\mathrm{ms}$.
- *Increasing the number of requested values.* We initialize an OSM scheme with 2^{14} keys each mapped to 2^{10} values, for a total of 2^{24} key-value pairs. We then issue queries for random keys, fetching an increasingly-large interval of values. Fig. 7 shows that search time is linear in the size of the interval. (Fetching a single value merely requires the client to fetch a single path in the search tree, as opposed to two paths in the general case of fetching intervals of values.)
- *Increasing the number of values per key.* We initialize an OSM scheme with 2^i keys each mapped to 2^{24-i} values, for a total of 2^{24} key-value pairs. We then issue queries for random keys, fetching an interval of 10 values. Our experiments confirm that search time does not depend on the number of values per key: across different choices of i, the latency is steady at $12.7\,\mathrm{ms}$.

Inserts. We initialize an OSM scheme with up to 2^{24} key-value pairs, and then measure the cost of inserting a random key-value pair. Fig. 8 shows that insert time is logarithmic in the number of key-value pairs in the database. Moreover, even with 2^{24} key-value pairs, insert time remains low at $5.4\,\mathrm{ms}$.

C. Private contact discovery in Signal

Signal [2] is a popular messaging service that offers end-to-end message encryption. When a user downloads the Signal application on a phone, the application communicates with Signal servers to determine which contacts on the user's phone use Signal; similarly, when the user adds new contacts to the phone, the application must determine which of these use Signal. This process is known as *contact discovery*. The importance to ensure its privacy (Signal servers do not learn the contact list in the user's phone) has already been documented [44].

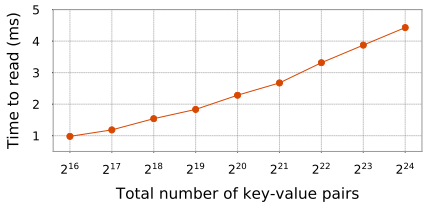

Figure 6: Search time is logarithmic in the number of key-value pairs.

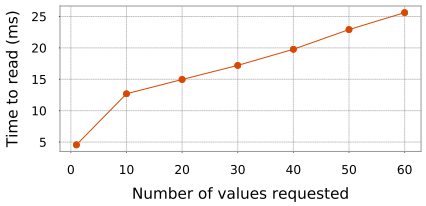

Figure 7: Search time is linear in the size of the requested interval.

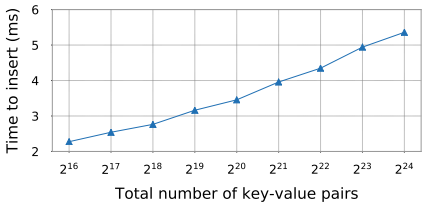

Figure 8: Insert time is logarithmic in the number of key-value pairs

Signal's approach. Signal makes contact discovery private via a method based on Intel SGX [44, 3], where the user sends a list of *encrypted* contacts and Signal servers compare these, within the hardware enclave, against the database of all Signal users. In order to prevent leakage through accesses to internal memory, the enclave first converts the list into an oblivious hash table, and then iterates over all Signal users, looking up each one in the hash table. Overall, if the user sends a list with m contacts and Signal has N users, the latency is $O(m^2 + N)$; note that the latency is linear in the number of all Signal users.

Our approach. We describe how to use Oblix to achieve private contact discovery with latency $O(m \log N)$; in particular, we do not perform a linear scan of all Signal users. This is an asymptotic improvement because $N \gg m$ (Signal has millions of users but any user typically has no more than several hundred contacts on a phone). Our experiments below show that these asymptotic gains yield efficiency gains in practice.

We use Oblix to construct, and then maintain, an oblivious index over all Signal users. When a user submits a list of contacts, the hardware enclave iterates over the contacts in the list, looking up each one in the oblivious index. As a result, latency is linear in the number contacts in the list (m), but only logarithmic in the number of all Signal users (N).

Experimental comparison. We consider databases of up to $N = 128\,\mathrm{M}$ users. Each user is identified by a phone number represented as an 8-byte integer (as in Signal's implementation); we thus initialized the index with 8-byte keys mapped to null values. We set the Path ORAM block size to 160 bytes.

We compare the performance of Signal's approach and our approach by issuing contact discovery requests with lists of different sizes ($m = 1, 10, 100, 1000$), and measure the latency to process the request at the server. We report the average time across 100 iterations per request.

Fig. 9 compares costs of Signal's approach and our approach. The cost in Signal's approach comes from: (i) converting the submitted list into an oblivious hash table, and then (ii) performing all the lookups. As the total number N of users grows, the latter dominates and the total cost increases linearly with N. When $N = 128\,\mathrm{M}$, the latency is $950 - 830\,\mathrm{ms}$.

Fixing the number m of contacts submitted by the user, the cost in our approach grows logarithmically in N, and so eventually becomes lower than the cost in Signal's approach, as N grows; the crossover point depends on m. E.g., fixing $m = 100$, if $N = 88\,\mathrm{M}$ then both approaches take $\sim 579\,\mathrm{ms}$; if $N = 128\,\mathrm{M}$ then Signal's approach degrades to $835\,\mathrm{ms}$ while our approach only takes $591\,\mathrm{ms}$ (an improvement of $\sim 30\%$).

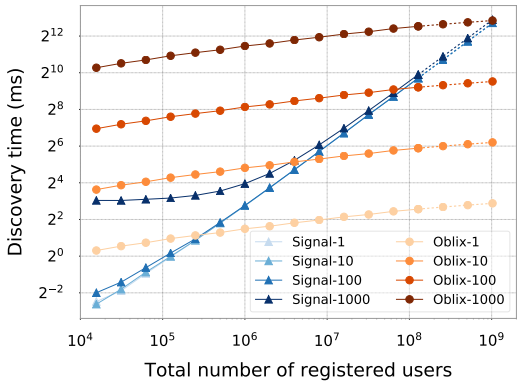

Figure 9: Cost of private contact discovery in Signal vs. Oblix *with an increasing number of registered users, for address books of sizes 1 to 1000. Dotted segments are extrapolations. Both axes scale logarithmically.*

Fig. 9 also extrapolates the cost of both approaches for databases larger than 128M users. With 1 billion users ($N = 10^9$), if $m = 1000$ then Signal's approach and our approach have similar costs (7.4 s and 7.6 s respectively); but if $m = 100$, then our approach is $\sim 9\times$ faster (0.74 s vs. Signal's 6.7 s).

Fig. 9 further highlights the benefit of our approach for *incremental* (as opposed to *initial*) contact discovery, where a user inserts new contacts into the phone and the Signal application must discover which of these are Signal users. While for initial contact discovery $m = 100$ and $m = 1000$ are representative values, for incremental contact discovery smaller values such as $m = 1$ and $m = 10$ are more appropriate. For these, our approach is up to two orders of magnitude faster. For example, when $m = 1$ and $N = 128\,\mathrm{M}$, our approach is $\sim 140\times$ faster ($5.9\,\mathrm{ms}$ vs. Signal's $832\,\mathrm{ms}$).

D. Anonymizing Google's Key Transparency

Google's Key Transparency (KT) [1, 46] is a scheme for ensuring integrity of key lookups: users can safely fetch other users' public keys from an untrustworthy key server. To achieve this, the service maintains a *Merkle prefix tree* over all user keys and gossips the root hash among the users; up to 2^d keys can be supported if the tree height is d ($d = 256$ in Google's implementation). When a user requests a public key, the service returns a *proof of integrity* that consists of the siblings of all the nodes in the path from the root to the leaf containing the public key. However, KT does not provide anonymity: when the server answers a request, it knows the identity of the user whose key it returns. We describe how to use Oblix to anonymize KT, with an order-of-magnitude improvement in cost compared to a baseline approach with the same level of security.

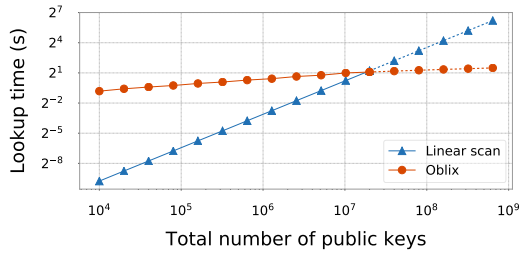

Figure 10: Cost of anonymous lookup in Key Transparency in baseline approach vs. our approach. Dotted segments are extrapolations. Both axes scale logarithmically.

Baseline approach. A simple baseline approach, similar in spirit to Signal's approach for private contact discovery (see Section VI-C), is a lookup that obliviously scans the whole Merkle tree, within the hardware enclave. Namely, we first initialize an empty array with d buckets; each bucket has a real slot and a dummy slot. We then iterate over all nodes in the Merkle tree as follows: given a node at depth j, if the node is part of the proof we write its value to the real slot in the j-th bucket; otherwise, we write it to the dummy slot. (Writing to either slot can itself be made oblivious.) After iterating over all nodes, the array will contain the proof for the desired public key. Overall, this approach has latency $O(N)$, where N is the total number of keys in the Merkle tree.

Our approach. We use Oblix to achieve anonymity with lookup latency $O(d \log N)$, a significant asymptotic gain over the baseline approach. The idea is simple: we store all Merkle tree nodes in an oblivious index in which keys are node identifiers and each key is mapped to a hash. As in the plaintext case, lookup consists of retrieving $O(d)$ nodes from the tree.

Experimental comparison. We consider databases of up to $N = 20\,\mathrm{M}$ public keys. We use 256-bit ECDSA public keys and use SHA-512/256 hashes to build a Merkle tree over the keys, in line with Google's implementation of KT. This results in a Path ORAM block size of 256 bytes. We compare the performance of the baseline approach and our approach by issuing 100 lookup requests and reporting their average latency.

Fig. 10 compares the baseline approach and our approach. The cost of the baseline approach is linear in N (number of public keys), while that of our approach is logarithmic in N. For small N, the baseline approach has lower cost; for $N = 20\,\mathrm{M}$, both approaches have comparable costs (2.1 s with Oblix vs. 2.3 s for the baseline); as N increases further, our approach has significantly lower cost. For example, for $N = 40\,\mathrm{M}$ our approach is $2\times$ faster (2.3 s vs. 4.6 s) and for $N = 320\,\mathrm{M}$ our approach is $\sim 14\times$ faster (2.6 s vs. 37 s). These latencies are on the order of seconds, and thus impact user experience.

E. Oblivious searchable encryption

Searchable encryption (SE) [62] enables a client to outsource encrypted data to an untrusted server, while still being able to search this remote data with small cost (in latency and bandwidth). Several works [11, 12, 17, 36] extend this functionality to support inserts and deletes to the data.

Below we first informally describe how to use Oblix to obtain an efficient SE scheme that supports *oblivious* searches, inserts, and deletes while further enabling the client to hide result sizes. We then evaluate our scheme's performance on real data. (For a formal definition, construction, and proofs for our SE scheme, see the full version.)

Our SE scheme. The plaintext data structure underlying our SE scheme is a *scored inverted index* (SII). A SII maps a key k to a (potentially empty) *list* of score-value pairs $[(s_i, v_i)]_1^n := \mathsf{SII}[k]$ that is sorted in descending order according to the scores s_i. The SII is parameterized by an integer r that dictates the "return size" of searches, as we now explain. The data structure supports search, insert and delete operations. A SE scheme $\mathsf{SE} := (\mathsf{Init}, \mathsf{Insert}, \mathsf{Delete}, \mathsf{Find}, \mathsf{Update}, \mathcal{S})$ allows a client to outsource storage of a SII to an untrusted server while still securely preserving search, insert, and delete functionality.

- **Initialization:** $\mathsf{SE.Init}^{\mathcal{S}}(m, \mathsf{SII}) \to \mathsf{st}$**.** On input a maximum number m of key-value pairs, and a scored inverted index SII, convert SII into a sorted multimap Map, and invoke $\mathsf{OSM.Init}(m, \mathsf{Map})$ to get OSM state $\mathsf{st}_{\mathsf{OSM}}$. This initializes the server $\mathcal{S}$. Output the initial client state $\mathsf{st} := \mathsf{st}_{\mathsf{OSM}}$.
- **Find:** $\mathsf{SE.Find}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, k, \omega) \to [(s_i, v_i)]_1^r$**.** On input client state st, keyword k, and search offset ω, first compute indices $i := \omega r$ and $j := (\omega + 1)r$ and then output $\mathsf{OSM.Find}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, k, i, j)$. The output equals $\mathsf{SII}[k][\omega r, \ldots, (\omega + 1)r]$.
- **Insert:** $\mathsf{SE.Insert}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [(k_i, s_i)]_1^n, v) \to \perp$**.** On input client state st, key-score list $[(k_i, s_i)]_1^n$, and value v, add (s_i, v) to $\mathsf{SII}[k_i]$ (if not present) for every i by invoking $\mathsf{OSM.Insert}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, k_i, (s_i, v))$.
- **Delete:** $\mathsf{SE.Delete}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [(k_i, s_i)]_1^n, v) \to \vec{b}$**.** On input value v and key-score list $[(k_i, s_i)]_1^n$, remove (s_i, v) from $\mathsf{SII}[k_i]$ (if present) for every i by invoking $\mathsf{OSM.Delete}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, k_i, (s_i, v))$, and output a boolean vector indicating whether the i-th removal was successful.

Evaluation on Enron dataset. We evaluate the latency of Oblix on the entire Enron email dataset [19], consisting of $\sim 528\,\mathrm{K}$ emails. We extracted keywords from this dataset by first stemming the words using standard stemming techniques, and then removing 675 stopwords. We next filtered out any words that contained non-alphabetic characters, or were ≥ 20 or ≤ 3 characters long. This gave us a total of $\sim 259\,\mathrm{K}$ keywords, which we used to create an inverted index having $\sim 38\,\mathrm{M}$ key-value pairs. We initialize the underlying Path ORAM implementation with a block size of 200 bytes. We then measure the cost of searches and inserts in the index and report the average of 100 iterations.

- **Search.** We search for the ten highest-ranking results for the keyword appearing in the largest number of documents ($\sim 145\,\mathrm{K}$). We observe that on average, the search takes 20.1 ms. For larger (or smaller) intervals, the time increases (or decreases) proportionately.
- **Insert.** We construct a new document consisting of the 100 most popular keywords. We then assign this document an unused document identifier, and populate the inverted index

with each of its constituent keywords. We observe that on average, the total time for inserting the 100 key-value pairs into the index is $\sim 775\,\mathrm{ms}$, or $7.75\,\mathrm{ms}$ per keyword.

VII. Related work

There is a rich literature on encrypted search indices and oblivious algorithms. We focus on works most relevant to us: doubly-oblivious ORAM, systems that combine obliviousness and hardware enclaves, and schemes for encrypted search.

A. Doubly-oblivious RAM

Most prior work [18, 21, 25, 38, 67, 72, 73, 78] on doubly-oblivious RAM focuses on using ORAM for secure multi-party computation (MPC) in the RAM model. These works focus on challenges arising from the interactive and communication-intensive nature of MPC. For example, one line of work [25, 38, 72, 73] expresses asymptotically efficient Tree ORAM algorithms as circuits with small size. Another line of work [18, 78] reduces online protocol costs by considering different ORAM paradigms that are asymptotically worse, but offer better concrete performance in the MPC setting. The trade-offs made by these works (such as optimizing for circuit size, or using asymptotically worse protocols) are not always effective in our setting of plain execution, where accessing memory is more expensive than performing computations.

B. Obliviousness on hardware enclaves

General-purpose programs. Several works [60, 61, 68] modify enclaved programs to endow them with *page-level* obliviousness. Such techniques can be composed with ours to obtain oblivious programs for functionalities beyond search.

ORAM. ZeroTrace [57] uses a doubly-oblivious Path ORAM client (corresponding to the naive client outlined in Section V-A) in an SGX enclave to get an oblivious memory controller. They use this to implement oblivious data structures. However, unlike Oblix's highly-optimized doubly-oblivious data structures, their data structures incur linear overhead per access. They also do not implement an efficient doubly-oblivious initialization algorithm, precluding applications like private contact discovery or private public-key retrieval.

ObliDB [20] uses Path ORAM and SGX enclaves to construct an oblivious database, but some of their techniques do not seem to be doubly-oblivious.

Works such as GhostRider [41], Tiny ORAM [22], or Shroud [43] propose combining ORAM techniques with custom trusted hardware. These systems use specialized hardware, whereas our construction utilizes widely available hardware enclaves. Furthermore, they provide poor efficiency (slow insertion and deletion when hiding size information) and security guarantees (result sizes leak) in the context of search.

Private information retrieval and private set intersection. Prior works attempt to use ORAM on trusted hardware of different kinds to achieve PIR [7, 70, 76, 77], but do not achieve scalable implementations. Tamrakar et al. [66] propose a protocol that utilizes hardware enclaves to achieve private set intersection. While their implemented system is quite performant, it is specialized for membership testing, and cannot support richer applications like anonymous Key Transparency or oblivious searchable encryption.

C. Search-specific schemes

Oblivious schemes. TWORAM [24] uses garbled RAM techniques to support oblivious search. Naveed [50] proposes the idea of hiding access patterns by storing an inverted index in the oblivious map of [74]. However, neither work supports inserts/deletes, neither hides result sizes, and neither provides a system design or implementation. Even if implemented, these schemes would suffer from the overhead of classical ORAM protocols (as discussed in Section I). Moataz and Blass [48] achieve substring search using ORAM techniques; one could use their techniques to extend our work to substring search.

Chan et al. [13] propose hiding result sizes via a new *differential obliviousness* technique, but their security guarantees are incomparable to ours. Asharov et al. [5] construct an ORAM scheme with good locality but weaker obliviousness guarantees, and use this to construct an oblivious SE scheme that does not hide result sizes. Neither scheme considers doubly-oblivious client algorithms, and neither provides an implementation.

Non-oblivious schemes. Fuhry et al. [23] use a enclave-based BTree-based search index to realize a searchable encryption scheme, but do not hide access patterns nor result sizes.

VIII. Acknowledgements

We thank Fariborz Assaderaghi, Alicia da Conceicao, Marc Joye, Sami Nassar, Ho Wai Wong-Lam, and other colleagues from NXP Semiconductors for valuable feedback and discussions, Assaf Araki and Intel for supplying the Intel SGX Cluster, Jethro Beekman for help with his Rust SGX SDK, and our shepherd Marina Blanton and the anonymous reviewers for valuable feedback that greatly improved this paper. This work was supported by NXP Semiconductors, the UC Berkeley Center for Long-Term Cybersecurity, Intel/NSF CPS-Security grants #1505773 and #20153754, as well as gifts to the RISELab from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] Google's Key Transparency. https://github.com/google/keytransparency.
[2] Signal. https://signal.org.
[3] Signal's Contact Discovery Service. https://github.com/whispersystems/ContactDiscoveryService/, 2017.
[4] M. A. Abdelraheem, T. Andersson, and C. Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. ePrint 2017/024, 2017.
[5] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi. Oblivious computation with data locality. ePrint 2017/772, 2017. http://eprint.iacr.org/2017/772.
[6] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov. Anonymous RAM. In *ESORICS '16*.
[7] S. Bakiras and K. F. Nikolopoulos. Adjusting the trade-off between privacy guarantees and computational cost in secure hardware PIR. In *SDM '11*.
[8] R. Bost. $\Sigma o \varphi o \varsigma$: Forward secure searchable encryption. In *CCS '16*.
[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT '17*.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS '15*.

[11] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS '14*.

[12] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO '13*.

[13] T.-H. H. Chan, K.-M. Chung, B. Maggs, and E. Shi. Foundations of differentially oblivious algorithms. ePrint 2017/1033, 2017. https://eprint.iacr.org/2017/1033.

[14] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *AsiaCCS '17*.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[16] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security '16*.

[17] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS '06*.

[18] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *CCS '17*.

[19] Enron email dataset. https://www.cs.cmu.edu/~./enron/.

[20] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. ArXiv, Report 1710.00458, 2017. http://arxiv.org/abs/1710.00458.

[21] S. Faber, S. Jarecki, S. Kentros, and B. Wei. Three-Party ORAM for secure computation. In *ASIACRYPT '15*.

[22] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. A low-latency, low-area hardware oblivious RAM controller. In *FCCM '15*.

[23] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi. HardIDX: Practical and secure index with SGX. In *DBSec '17*.

[24] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO '16*.

[25] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS '13*.

[26] M. Giaruad, A. Anzala-Yamajako, O. Bernard, and P. Lafourcase. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *ICETE '17*.

[27] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[28] M. T. Goorish, M. Michael, O. Ohrimenko, and T. Roberto. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA '12*.

[29] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *EUROSEC '17*.

[30] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *CCS '16*.

[31] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security '17*.

[32] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *ATC '2017*.

[33] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song. ShadowCrypt: Encrypted web applications for everyone. In *CCS '14*.

[34] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS '12*.

[35] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. McKeen. Intel software guard extensions: EPID provisioning and attestation services.

[36] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS '12*.

[37] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS '16*.

[38] M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *ASIACRYPT '14*.

[39] K. Kurosawa. Garbled searchable symmetric encryption. In *FC '14*.

[40] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A mimicry privacy shield - a system's approach to data privacy on public cloud. In *USENIX Security '14*.

[41] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS '15*.

[42] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 2014.

[43] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST '13*.

[44] Marlinspike, Moxie. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/, 2017.

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*.

[46] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security '15*.

[47] R. C. Merkle. A certified digital signature. In *CRYPTO '89*.

[48] T. Moataz and E.-O. Blass. Oblivious substring search with updates. ePrint 2015/722, 2015. http://eprint.iacr.org/2015/722.

[49] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES '17*.

[50] M. Naveed. The fallacy of composition of oblivious RAM and searchable encryption. ePrint 2015/668, 2015.

[51] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *SP '14*.

[52] W. Ogata, K. Koiwa, A. Kanaoka, and S. Matsuo. Toward practical searchable symmetric encryption. In *IWSEC '13*.

[53] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS '16*.

[54] Relevance scores: Understanding and customizing. https://docs.marklogic.com/guide/search-dev/relevance.

[55] D. S. Roche, A. J. A. Aviv, and S. G. Choi. A practical oblivious map data structure with secure deletion and history independence. In *SP '16*.

[56] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In *SP '16*.

[57] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotrace : Oblivious memory primitives from Intel SGX. ePrint 2017/549, 2017. http://eprint.iacr.org/2017/549.

[58] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to conceal cache attacks. In *DIMVA '17*.

[59] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS '17*.

[60] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *AsiaCCS '16*.

[61] R. Sinha, S. K. Rajamani, and S. A. Seshia. A compiler and verifier for page access oblivious computation. In *ESEC/FSE '17*.

[62] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00*.

[63] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS '14*.

[64] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS '13*.

[65] R. Strackx and F. Piessens. Ariadne: a minimal approach to state continuity. In *USENIX Security '16*.

[66] S. Tamrakar, J. Liu, A. Paverd, J. Ekberg, B. Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. In *AsiaCCS '17*.

[67] T. Toft. Secure data structures based on multi-party computation. In *PODC '11*.

[68] S. Tople and P. Saxena. On the trade-offs in oblivious execution techniques. In *DIMVA '17*.

[69] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security '17*.

[70] P. Wang, H. Wang, and J. Pieprzyk. Secure coprocessor-based private information retrieval without periodical preprocessing. In *AISC '10*.

[71] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky Cauldron on the Dark Land: Understanding memory side-channel hazards in SGX. In *CCS '17*.

[72] X. S. Wang, T. Chan, and E. Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *CCS '15*.

[73] X. S. Wang, Y. Huang, T. Chan, A. Shelat, and E. Shi. SCORAM: oblivious RAM for secure computation. In *CCS '14*.
[74] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS '14*.
[75] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP '15*.
[76] Y. Yang, X. Ding, R. H. Deng, and F. Bao. An efficient PIR construction using trusted hardware. In *ISW '08*.
[77] X. Yu, C. Fletcher, L. Ren, M. van Dijk, and S. Devadas. Efficient private information retrieval using secure hardware. MIT Tech Report 509, 2013. http://csg.csail.mit.edu/pubs/memos/Memo-509/memo509.pdf.
[78] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting Square-Root ORAM: efficient random access in multi-party computation. In *SP '16*.
[79] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security '16*.

APPENDIX A
CONSTRUCTION OF AN OSM SCHEME

We now provide details for our construction of an OSM scheme from Section IV-C. We first provide detailed pseudocode (Fig. 11) for our construction, and then provide correctness and security proofs for the same.

Theorem 1. *The OSM scheme from Section IV-C is correct as per the security definition in Section IV-B.*

Proof. The oblivious sorted multimap scheme is correct assuming the correctness of the plaintext sorted multimap and the oblivious data structures framework of [74] are correct. It is easy to verify the correctness of the plaintext sorted multimap, since it is a small modification to order statistic trees and AVL trees. The sorted multimap also satisfies Definition 1, and hence can be used with the ODS framework. □

Theorem 2. *The OSM scheme from Section IV-C is secure as per the security definition in Section IV-B.*

Proof. We construct a simulator Sim (Fig. 15) that uses the ODS simulator $\mathsf{Sim}_{\mathsf{ODS}}$ as a black box. The view of an adversary interacting with the simulator oracle $\mathbf{S}_{\mathsf{Ideal}}$ instantiated with Sim is indistinguishable from an adversary interacting with $\mathbf{S}_{\mathsf{Real}}$ because the simulator pads the number of access to the appropriate amount, and so the adversary sees the same number of read and write memory accesses to the server regardless of the query input.

$\mathsf{Sim.Init}^{\mathcal{S}}(m, \ell_{\mathsf{k}}, \ell_{\mathsf{v}})$: 1) $l := \mathsf{calc\_node\_size}(\ell_{\mathsf{k}}, \ell_{\mathsf{v}})$. 2) $\mathsf{st}_{\mathsf{ODS}} \leftarrow \mathsf{Sim}_{\mathsf{ODS}}.\mathsf{Init}^{\mathcal{S}}(m, l)$. 3) Store $\mathsf{st} := (\ell_{\mathsf{k}}, \ell_{\mathsf{v}}, \mathsf{st}_{\mathsf{ODS}})$.	$\mathsf{Sim.Size}^{\mathcal{S}}()$: 1) Let $b := 1.44 \log(m)$. 2) $\mathsf{Sim}_{\mathsf{ODS}}.\mathsf{Access}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}_{\mathsf{ODS}}, b)$.
$\mathsf{Sim.Insert}^{\mathcal{S}}()$: 1) Let $b := 1.44 \log(m) + 1$. 2) $\mathsf{Sim}_{\mathsf{ODS}}.\mathsf{Access}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}_{\mathsf{ODS}}, b)$.	$\mathsf{Sim.Find}^{\mathcal{S}}(r)$: 1) Let $b := 2 \times 1.44 \log(m) + r$. 2) $\mathsf{Sim}_{\mathsf{ODS}}.\mathsf{Access}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}_{\mathsf{ODS}}, b)$.

Figure 15: Simulator for our OSM construction.

□

APPENDIX B
SECURITY OF PATH ORAM

Security for Path ORAM is defined via two experiments: one in which an adversary interacts with an oracle $\mathbf{S}_{\mathsf{Real}}$ acting as a proxy to the scheme, and another in which the adversary interacts with an oracle $\mathbf{S}_{\mathsf{Ideal}}$ that acts as a proxy to a simulator Sim that *only gets some of the inputs*. Both oracles expose to the adversary the same interface (see Fig. 16): the adversary can make an `Init` query that specifies the initial blocks (and a maximum number of blocks), and then can make any number of `Read` or `Evict` queries, with the restriction that the input to any `Evict` query is the list of *all* leaves fetched since the previous such call. The adversary may observe accesses to the server made by the oracles as a result of these invocations. Path ORAM is *secure* if the adversary cannot distinguish between the two experiments.

APPENDIX C
SECURITY OF AN ODS SCHEME

Security of an ODS scheme defined via two experiments: one in which an adversary interacts with an oracle $\mathbf{S}_{\mathsf{Real}}$

$\mathsf{OSM.Init}^{\mathcal{S}}(m, \mathsf{Map})$: 1) Let $\mathsf{Tree} = \mathsf{Map.New}()$; 2) For $(\mathsf{k}, \mathsf{v}) \in \mathsf{Map}$: $\mathsf{Tree.Insert}(\mathsf{k}, \mathsf{v})$. 3) Let $\mathsf{rt} = \mathsf{Tree.root}()$. 4) Let $i = 1$. 5) If $\mathsf{rt} \neq \perp$, for each node in Tree: a) If $\mathsf{node} = \mathsf{rt}$: $i_{\mathsf{rt}} = i$. b) $i := i + 1$. 6) Let $(\mathsf{ptr}_{\mathsf{rt}}, \mathsf{st}_{\mathsf{ODS}}) := \mathsf{ODS.Init}(m, \mathsf{Tree}, i_{\mathsf{rt}})$. 7) Output $\mathsf{st} = (\mathsf{ptr}_{\mathsf{rt}}, \mathsf{rt}, \mathsf{st}_{\mathsf{ODS}})$.	$\mathsf{OSM.Size}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k})$: 1) Let $\mathsf{rootKey} := \mathsf{st.rt.key}$. 2) Let $\mathsf{ptr} := \mathsf{st.ptr}_{\mathsf{rt}}$. 3) $\mathsf{ODS.Start}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st.st}_{\mathsf{ODS}}, \mathsf{st.ptr}_{\mathsf{rt}})$ 4) Let $\mathsf{size} := 0$. 5) While $\mathsf{rootKey} \neq \perp$: a) Let $\mathsf{curNode} \leftarrow \mathsf{ODS.Access}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st.st}_{\mathsf{ODS}}, \mathtt{read}(\mathsf{ptr}))$. b) If $\mathsf{rootKey} = \mathsf{k}$: set $\mathsf{size} := \mathsf{curNode.size}()$; break. c) Else if $\mathsf{rootKey} < \mathsf{k}$: i) $\mathsf{rootKey} := \mathsf{curNode.leftKey}()$. ii) $\mathsf{ptr} := \mathsf{curNode.lChild}()$. d) Else: i) $\mathsf{rootKey} := \mathsf{curNode.rightKey}()$. ii) $\mathsf{ptr} := \mathsf{curNode.rChild}()$. 6) Let $\mathsf{bound} := 1.44 \cdot \log(\mathsf{osmClient.treeSize})$. 7) $\mathsf{st.ptr}_{\mathsf{rt}} := \mathsf{ODS.Finalize}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st.st}_{\mathsf{ODS}}, \mathsf{st.rt}, \mathsf{bound})$. 8) Output size.
$\mathsf{OSM.Insert}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v})$: 1) Let $\mathsf{ptr}_{\mathsf{rt}} := \mathsf{st.ptr}_{\mathsf{rt}}$. 2) Let $\mathsf{st}_{\mathsf{ODS}} := \mathsf{st.st}_{\mathsf{ODS}}$. 3) Let $(\mathsf{rt}', \dots) \leftarrow \mathsf{OSM.InsHelper}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v}, \mathsf{ptr}_{\mathsf{rt}})$ (Fig. 13). 4) Let $\mathsf{pad} := 1.44 \cdot \log(\mathsf{osmClient.treeSize}) + 1$. 5) $\mathsf{st.ptr}_{\mathsf{rt}} := \mathsf{ODS.Finalize}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}_{\mathsf{ODS}}, \mathsf{rt}', \mathsf{pad})$.	$\mathsf{OSM.Find}^{\mathcal{S}}(\mathsf{st}, \mathsf{k}, i, j)$: 1) Let $\mathsf{ptr}_{\mathsf{rt}} := \mathsf{st.ptr}_{\mathsf{rt}}$. 2) Let $\mathsf{st}_{\mathsf{ODS}} := \mathsf{st.st}_{\mathsf{ODS}}$. 3) Let $(\mathsf{rt}', \vec{\mathsf{v}}) \leftarrow \mathsf{OSM.FindHelper}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}, \mathsf{k}, \mathsf{v}, \mathsf{ptr}_{\mathsf{rt}})$ (Fig. 12). 4) Let $\mathsf{pad} := 1.44 \cdot \log(\mathsf{osmClient.treeSize}) + 1$. 5) $\mathsf{st.ptr}_{\mathsf{rt}} := \mathsf{ODS.Finalize}^{\mathcal{S}}(\mathtt{mut}\ \mathsf{st}_{\mathsf{ODS}}, \mathsf{rt}', \mathsf{pad})$. 6) Output $\vec{\mathsf{v}}$.

Figure 11: Construction of a sorted multimap.

OSM.FindHelper$^{\mathcal{S}}$(mut st, k, i, j)
1) Let st_{ODS} := st.st_{ODS}.
2) if rootKey $\neq \perp$:
  a) **Find path to i-th k-node**:
    lower $\leftarrow$ OSM.GetAtIndex$^{\mathcal{S}}$(mut st, k, i).
  b) **Find path to j-th k-node**:
    upper $\leftarrow$ OSM.GetAtIndex$^{\mathcal{S}}$(mut st, k, j).
  c) Find **node with index** i: first := last(lower).
  d) Find **node with index** j: last := last(upper).
    Compute the node at the intersection of the two paths:
  e) Let intersection := $\perp$.
  f) For $i \in \{0, \ldots, \min(\text{lower.len}, \text{upper.len})\}$:
    if lower[i] = upper[i]: set intersection := lower[i].

    Find nodes that lie between the two paths:
  g) If intersection $\neq \perp$:
    i) Initialize empty queue workQueue.
    ii) Initialize empty list of matching nodes results.
    iii) workQueue.push(intersection.ptr).
    iv) While workQueue is not empty:
      A) Let ptr := workQueue.pop().
      B) curNode := ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, read(ptr)).
      C) If curNode < first and curNode < last:
        workQueue.push(curNode.rChild).
      D) Else if curNode > first and curNode > last:
        workQueue.push(curNode.lChild).
      E) Else:
        workQueue.push(curNode.rChild);
        workQueue.push(curNode.lChild).
    v) If (first $\leq$ curNode $\leq$ last) and (curNode.key = k):
      result.push(curNode).
  h) Return results.
3) Else, return empty list.

Figure 12

OSM.InsHelper$^{\mathcal{S}}$(mut st, k, v, ptr)
1) Let st_{ODS} := st.st_{ODS}.
2) if ptr $\neq \perp$:
  a) let curNode := ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, read(ptr)).
  b) If k = curNode.key and v = curNode.value: return.
  c) Else if k < curNode.key:
    i) Let
    (child, k_c, size, keySize) = OSM.InsHelper$^{\mathcal{S}}$(mut st, k, v, curNode.lChild).
    ii) If curNode.key = k_c: curNode.lSize := size.
    iii) If curNode.key = k: curNode.lSize := size; keySize := curNode.size.
    iv) curNode.lChild := ptr_c.
    v) ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, write(ptr, curNode)).
    vi) Return (Balance$^{\mathcal{S}}$(mut st, curNode.key, ptr), keySize).
  d) Else:
    i) Let
    (child, k_c, size, keySize) = OSM.InsHelper$^{\mathcal{S}}$(mut st, k, v, curNode.rChild).
    ii) If curNode.key = k_c: curNode.rSize := size.
    iii) If curNode.key = k: curNode.rSize := size; keySize := curNode.size.
    iv) curNode.rChild := ptr_c.
    v) ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, write(ptr, curNode)).
    vi) Return (Balance$^{\mathcal{S}}$(mut st, curNode.key, ptr), keySize).
3) Else:
  a) **Construct new node**
    $$\text{node} := \begin{pmatrix} \text{Key k} & \text{Value v} \\ \text{lChild} = \perp & \text{rChild} = \perp \\ \text{lSize} = 0 & \text{rSize} = 0 \\ \text{leftHeight} = 0 & \text{rightHeight} = 0 \end{pmatrix}.$$
  b) st.treeSize = st.treeSize + 1.
  c) Let $\text{ptr}_c \leftarrow$ ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, ins(node)).
  d) (node.key, ptr_c, node.size(), node.size())

Figure 13

OSM.GetAtIndex$^{\mathcal{S}}$(mut st, k, i)
1) Let st_{ODS} := st.st_{ODS}.
2) Let curKey := st.rootKey.
3) Let ptr := st.ptr_{rt}.
4) Initialize empty list path.
5) While curKey $\neq \perp$:
  a) Let curNode := ODS.Access$^{\mathcal{S}}$(mut st_{ODS}, read(ptr)).
  b) **If the current node is a k-node**, i.e. k = curKey:
    i) If i = curNode.lSize():
      A) path.Insert(curNode)
      B) Break out of loop.
    ii) Else if i < curNode.lSize():
      A) curKey := curNode.leftKey().
      B) ptr := curNode.lChild().
    iii) Else:
      A) curKey := curNode.rightKey().
      B) ptr := curNode.rChild().
      C) $i := i -$ curNode.lSize() + 1.
    Ignore non-k-nodes for indexing purposes.
  c) Else if k < curKey:
    i) curKey := curNode.leftKey().
    ii) ptr := curNode.lChild().
  d) Else:
    i) curKey := curNode.rightKey().
    ii) ptr := curNode.rChild().
  e) path.Insert(curNode)
6) Return path.

Figure 14

implementing the ODS scheme, and another in which he interacts with an oracle $\mathbf{S}_{\text{Ideal}}$ that acts as a proxy to a simulator Sim that *only gets some of the inputs*. Both oracles expose to the adversary the same interface (see Fig. 17): the adversary first invokes the oracle on `Init`, and then makes any number

	Security	
	$\mathbf{S}_{\text{Real}}$	$\mathbf{S}_{\text{Ideal}}$
$\texttt{Init}(m, [\text{bl}_i]_1^n)$	store st $\leftarrow$ ORAM.Init$^{\mathcal{S}}(m, [\text{bl}_i]_1^n)$	store st $\leftarrow$ Sim.Init$^{\mathcal{S}}(m, \lvert\text{bl}_1\rvert)$
Read(bid)	ORAM.ReadBlock$^{\mathcal{S}}$(mut st, bid)	Sim.ReadBlock$^{\mathcal{S}}$(mut st)
$\texttt{Evict}([\text{lf}_i]_1^n)$	ORAM.Evict$^{\mathcal{S}}$(mut st, $[\text{lf}_i]_1^n$)	Sim.Evict$^{\mathcal{S}}$(mut st, n)

Figure 16: Real and ideal oracles for Path ORAM.

of `Access` calls to the oracle. The adversary is allowed to observe the server accesses made by the oracles as a result of these invocations. An ODS scheme is *secure* if the adversary cannot distinguish between the two experiments.

Appendix D
Construction of Path DORAM

We expand upon our description of Path DORAM in Section V-A by providing pseudocode for our construction in Fig. 18.

	Security	
	$\mathbf{S}_{\mathsf{Real}}$	$\mathbf{S}_{\mathsf{Ideal}}$
$\texttt{Init}\begin{pmatrix}\mathsf{DS}\\ [\mathsf{DSop}_j]_1^s\\ m\\ [\mathsf{node}_i]_1^n\\ i_{\mathsf{rt}}\end{pmatrix}$	1) Check that sequentially executing operations in $[\mathsf{DSop}_j]_1^s$ results in nodes $[\mathsf{node}_i]_1^n$ with root at index i_{rt}. 2) Store $(\mathsf{st}, \mathsf{ptr}_{\mathsf{rt}}) \leftarrow \mathsf{ODS.Init}^{\mathcal{S}}(m, [\mathsf{node}_i]_1^n, i_{\mathsf{rt}})$.	Store $\mathsf{st} \leftarrow \mathsf{Sim.Init}^{\mathcal{S}}(m, \lvert\mathsf{Node}_1\rvert)$.
$\texttt{Access}(\mathsf{DSop})$	1) Start ODS: $\mathsf{ODS.Start}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{ptr}_{\mathsf{rt}})$. 2) Invoke the data structure DS on DSop, replacing plaintext pointer accesses with corresponding ODS pointer accesses. 3) Let the current root node be node. 4) Store $\mathsf{ptr}'_{\mathsf{rt}} \leftarrow \mathsf{ODS.Finalize}(\texttt{mut}\ \mathsf{st}, \mathsf{node}, \mathsf{bound})$.	$\mathsf{Sim.Access}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{bound})$.

Figure 17: Real and ideal oracles for ODS.

$\mathsf{DORAM.ReadBlock}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, \mathsf{bid}, \mathsf{lf}) \to \mathsf{bl}$
1) Fetch from $\mathcal{S}$ the list of buckets Bu that are on the path to lf.
2) Let mut ans be a dummy block.
3) For each bucket $\mathsf{bu} \in \mathsf{Bu}$:
   a) For each $i \in \{1, \ldots, C\}$:
      i) Let $\mathsf{cond} := (\mathsf{bu}[i].\mathsf{bid} = \mathsf{bid})$.
      ii) $\mathsf{OblSwap}(\mathsf{cond}, \mathsf{ans}, \mathsf{bu}[i])$
   b) Insert bu into st.ImplicitBuckets.
4) Insert ans into st.ExplicitBlocks and output ans.

$\mathsf{DORAM.Init}^{\mathcal{S}}(m, [\mathsf{bl}_i]_1^n)$
1) Let $\mathsf{treeSize} := \mathsf{ComputeTreeSize}(m, C)$.
2) Let **layer size (in buckets)** $s := (\mathsf{treeSize} + 1)/2$.
3) Initialize empty list of buckets buckets.
4) Initialize block list $\mathsf{blocks} := [\mathsf{bl}_i]_1^n$.
5) For each layer in the ORAM tree:
   a) Let $k := \mathsf{blocks.len}()$.
   b) Let D be a list of sC dummy blocks.
   c) For the i-th chunk of C blocks in D, set the node of each block in this chunk to be the i-th tree node in the current layer.
   d) Append D to blocks (so that $\mathsf{blocks.len}() = k + sC$).
   e) Annotate each block in blocks with a boolean flag indicating whether or not it is dummy.
   f) **Obliviously prepare blocks for bucketing:** obliviously sort blocks so that all same-node blocks are grouped together, and within every such group, dummy blocks are sorted to the end of the group. The groups are sorted in ascending node order.
   g) Let $\mathsf{ctr} := 0$.
   h) Let $\mathsf{cur\_node} := \bot$.
   i) For each bl in blocks, **try to assign it to a bucket in current layer:**
      i) Let $b := \mathsf{bl.node} = \mathsf{cur\_node}$.
      ii) $\mathsf{ctr} := b \cdot (\mathsf{ctr} + 1)$.
      iii) $\mathsf{cur\_node} := \mathsf{bl.node}$.
      iv) $\mathsf{bl.in\_bucket} := \mathsf{ctr} < C$;
   j) **Obliviously collect all bucketed blocks together:** obliviously sort blocks so that blocks with in_bucket = false are last, and the remaining blocks are sorted in ascending order of their node.
   k) Construct s buckets from the first sC blocks in blocks, and append these to buckets.
   l) **Remove bucketed blocks:** $\mathsf{blocks} := \mathsf{blocks}[sC \ldots sC + k]$.
   m) For each remaining block bl in blocks, **update its assigned node:** $\mathsf{bl.node} := \mathsf{bl.node.parent}()$.
   n) **Update layer size:** $s := s/2$.
6) Create ExplicitBlocks from the remaining blocks in blocks.
7) Encrypt and upload each bucket in buckets to $\mathcal{S}$.
8) Output $\mathsf{st} := (\mathsf{ImplicitBuckets} = \bot, \mathsf{ExplicitBlocks})$.

$\mathsf{DORAM.Evict}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [\mathsf{lf}_i]_1^n)$
1) If $\mathsf{st.NumWrites} = 0 \pmod t$: $\mathsf{DORAM.Evict_s}^{\mathcal{S}}(\mathsf{st}, [\mathsf{lf}_i]_1^n)$.
2) Else: $\mathsf{DORAM.Evict_f}^{\mathcal{S}}(\mathsf{st}, [\mathsf{lf}_i]_1^n)$.
3) Set $\mathsf{st.NumWrites} := \mathsf{st.NumWrites} + 1$.

$\mathsf{DORAM.Evict_s}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [\mathsf{lf}_i]_1^n)$
1) Initialize $\mathsf{blocks} := \mathsf{st.ExplicitBlocks}$.
2) Append blocks (in each bucket) in st.ImplicitBuckets to blocks.
3) Let Nodes be the list of nodes comprising paths to $[\mathsf{lf}_i]_1^n$.
4) Initialize bucket fullness map BuFu so that for each $\mathsf{node} \in \mathsf{Nodes}$, $\mathsf{BuFu}[\mathsf{node}] = 0$.
5) **Assign blocks to buckets:** for each block bl in blocks:
   a) Let $\mathsf{assigned\_node} := \bot$ and let $\mathsf{assigned\_flag} := 0$.
   b) For each node node in path to bl.lf:
      i) Let $\mathsf{is\_free} := (\mathsf{BuFu}[\mathsf{node}] \neq \bot) \ \wedge \ (\mathsf{BuFu}[\mathsf{node}] < C)$.
      ii) Let $\mathsf{cond} := \mathsf{is\_free} \ \wedge \ \neg\, \mathsf{assigned\_flag}$.
      iii) Increment $\mathsf{BuFu}[\mathsf{node}]$ by cond.
      iv) $\mathsf{OblSwap}(\mathsf{cond}, \mathsf{assigned\_node}, \mathsf{node})$.
      v) $\mathsf{OblSwap}(\mathsf{cond}, \mathsf{assigned\_flag}, 1)$.
   c) Set $\mathsf{bl.node} := \mathsf{assigned\_node}$.
6) **Append dummy blocks:** Append $\lvert\mathsf{ImplicitBuckets}\rvert \cdot C$ dummy blocks (having node $\bot$) to blocks.
7) **Construct buckets:** obliviously sort blocks by bl.node, sorting blocks with node $\bot$ to the end.
8) From the first $\lvert\mathsf{Nodes}\rvert \cdot C$ elements of blocks, construct the buckets to be written back. Truncate the remainder of blocks at at maximum stash size, and insert these blocks into st.ExplicitBlocks.

$\mathsf{DORAM.Evict_f}^{\mathcal{S}}(\texttt{mut}\ \mathsf{st}, [\mathsf{lf}_i]_1^n)$
1) Let Nodes be the list of nodes comprising paths to $[\mathsf{lf}_i]_1^n$.
2) Initialize bucket fullness map BuFu so that for each $\mathsf{node} \in \mathsf{Nodes}$, $\mathsf{BuFu}[\mathsf{node}] = 0$.
3) **Assign blocks to buckets:** for each block bl in st.ExplicitBlocks:
   a) Let $\mathsf{assigned\_node} := \bot$ and let $\mathsf{assigned\_flag} := 0$.
   b) For each node node in path to bl.lf:
      i) Let $\mathsf{is\_free} := \mathsf{BuFu}[\mathsf{node}] \neq \bot \ \wedge \ \mathsf{BuFu}[\mathsf{node}] < C$.
      ii) Let $\mathsf{cond} := \mathsf{is\_free} \ \wedge \ (\mathsf{assigned\_flag} = 0)$.
      iii) Increment $\mathsf{BuFu}[\mathsf{node}]$ by cond.
      iv) $\mathsf{OblSwap}(\mathsf{cond}, \mathsf{assigned\_node}, \mathsf{node})$.
      v) $\mathsf{OblSwap}(\mathsf{cond}, \mathsf{assigned\_flag}, 1)$.
   c) Set $\mathsf{bl.node} := \mathsf{assigned\_node}$.
4) **Insert blocks into buckets:**
5) Initialize empty list blocks.
6) Append all blocks in ExplicitBlocks to blocks.
7) For each bucket $\mathsf{bu} \in \mathsf{st.ImplicitBuckets}$:
   a) For each block bl in bu:
      i) Set $\mathsf{bl.node} := \mathsf{bu.node}$.
      ii) Insert bl into blocks.
8) Construct buckets from blocks by oblivious sorting as in $\mathsf{Evict_s}$.
9) Write back constructed buckets to $\mathcal{S}$, and insert any remaining blocks into st.ExplicitBlocks.

Figure 18: Algorithms for Path doubly-oblivious RAM (for bucket size C).