New Constructions for Forward and Backward Private Symmetric Searchable Encryption

Javad Ghareh Chamani
jgc@cse.ust.hk
Hong Kong University of Science and Technology
& Sharif University of Technology

Dimitrios Papadopoulos
dipapado@cse.ust.hk
Hong Kong University of Science and Technology

Charalampos Papamanthou
cpap@umd.edu
University of Maryland

Rasool Jalili
jalili@sharif.edu
Sharif University of Technology

ABSTRACT

We study the problem of dynamic symmetric searchable encryption. In that setting, it is crucial to minimize the information revealed to the server as a result of update operations (insertions and deletions). Two relevant privacy properties have been defined in that context: *forward* and *backward* privacy. The first makes it hard for the server to link an update operation with previous queries and has been extensively studied in the literature. The second limits what the server can learn about entries that were deleted from the database, from queries that happen after the deletion. Backward privacy was formally studied only recently (Bost et al., CCS 2017) in a work that introduced a formal definition with three variable types of leakage (Type-I to Type-III ordered from most to least secure), as well as the only existing schemes that satisfy this property. In this work, we introduce three novel constructions that improve previous results in multiple ways. The first scheme achieves Type-II backward privacy and our experimental evaluation shows it has $145 - 253\times$ faster search computation times than previous constructions with the same leakage. Surprisingly, it is faster even than schemes with Type-III leakage which makes it the most efficient implementation of a forward and backward private scheme so far. The second one has search time that is asymptotically within a polylogarithmic multiplicative factor of the theoretical optimal (i.e., the result size of a search), and it achieves the strongest level of backward privacy (Type-I). All previous Type-I constructions require time that is at least linear in the total number of updates for the requested keywords, even the (arbitrarily many) previously deleted ones. Our final scheme improves upon the second one by reducing the number of roundtrips for a search at the cost of extra leakage (Type-III).

KEYWORDS

Searchable Encryption; Forward/Backward Privacy; Cloud Databases

ACM Reference Format:
Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243833

1 INTRODUCTION

Storing an encrypted database at a remote server, while retaining the ability to access and dynamically maintaining it, is fundamental for modern computing. Consider for example a client that owns a database, outsources it to an untrusted server and subsequently issues search queries of the form *"retrieve all documents that contain keyword w"*. Ideally, the server should not only learn nothing about the content of the documents (which can be achieved by traditional encryption schemes), but also no additional meta-information, e.g., how many times w was searched for, frequency of keywords in the database, etc. This "perfect" level of privacy is theoretically achievable by strong encryption techniques such as fully-homomorphic encryption [21], whose large performance overhead however limits adoption in practice.

Symmetric searchable encryption (SSE), originally proposed by Song et al. [41], provides a way for accessing this encrypted database efficiently by slightly relaxing the privacy requirements. Concretely, SSE reveals some information to the server during query execution, known as *leakage*. This leakage typically includes the *search pattern* that reveals which search queries refer to the same keyword w as well as the *access pattern* that reveals which files are returned for a query.

Dynamic symmetric searchable encryption and its leakage. The first works on SSE focused on static datasets and it was not until 2009 when SSE schemes that support updates on the database in a principled manner (also known as dynamic SSE schemes) were first introduced [13, 27, 28]. Dynamic SSE, however, introduces additional privacy concerns due to the added functionality. For example, *(a)* adding a file f to the database might reveal that f contains keywords that were searched before, or *(b)* searching for a keyword w might reveal which files from the past (that have been removed from the database now) contained w.

Forward and backward privacy. Schemes that avoid the leakage associated with case *(a)* above are called *forward private* and were first introduced by Chang and Mitzenmacher [13] and subsequently

Table 1: Comparison of existing forward and backward private dynamic SSE schemes. N is the total number of (document, keyword) pairs, $|W|$ is the number of distinct keywords, and $|D|$ is total number of documents. For keyword w, a_w is the total number of updates, n_w is the number of files currently containing w, and d_w is the number of deleted entries for w. RT is the number of roundtrips for search. BP represents the achieved backward privacy type. $\tilde{O}$ notation hides polylogarithmic factors.

Scheme	Computation		Communication			Client Storage	BP
	Search	Update	Search	Update	Search RT		
Moneta [7]	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^2 N)$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^3 N)$	3	$O(1)$	I
Fides [7]	$O(a_w)$	$O(1)$	$O(a_w)$	$O(1)$	2	$O(\|W\|log\|D\|)$	II
Diana$_{del}$ [7]	$O(a_w)$	$O(\log a_w)$	$O(n_w + d_w \log a_w)$	$O(1)$	2	$O(\|W\|log\|D\|)$	III
Janus [7]	$O(n_w d_w)$	$O(1)$	$O(n_w)$	$O(1)$	1	$O(\|W\|log\|D\|)$	III
Mitra [Sec. 3]	$O(a_w)$	$O(1)$	$O(a_w)$	$O(1)$	2	$O(\|W\|log\|D\|)$	II
Orion [Sec. 4]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(1)$	I
Horus [Sec. 4.3]	$O(n_w \log d_w \log N)$	$O(\log^2 N)$	$O(n_w \log d_w \log N)$	$O(\log^2 N)$	$O(\log d_w)$	$O(\|W\|log\|D\|)$	III

refined in [5, 19, 20, 30, 42, 43]. Forward privacy has become an essential property for dynamic SSE schemes, especially in light of recent file-injection attacks [47] that become particularly effective when the SSE scheme is not forward-private. Schemes that try to limit the leakage from case *(b)* are called *backward private* and have been studied far less in the literature. Other than an informal mention in [43] and the "folklore" construction from ORAM techniques, no formal definition or construction that achieves this property existed for a long time. Very recently, Bost et al. [7] introduced a formal definition for backward privacy with three different types of leakage ordered from most to least secure.

Type-I leakage (Backward Privacy with Insertion Pattern): Type-I schemes reveal, during a search for w, the number and type of previous updates associated with w, the identifiers of files containing w currently in the database, and when each such file was inserted.

Type-II leakage (Backward Privacy with Update Pattern): In addition to the information contained in Type-I leakage, Type-II schemes also reveal when all updates related to w took place.

Type-III leakage (Weak Backward Privacy): Finally, Type-III schemes also reveal exactly which deletion update canceled which previous addition (e.g., the deletion that took place during the tenth operation canceled the addition from the fifth operation). Note that all three types satisfy the basic property of hiding the actual identifiers of files that contained w but were deleted prior to w's search.

The schemes of Bost et al. [7]. Bost et al. [7] provided four backward-private constructions that achieve different privacy/efficiency trade-offs. The first one is Fides which is a Type-II construction. At a high level, it uses two deployments of the forward private SSE scheme of [5] to store update entries of the form (w, id, op) where w is a keyword, id is a file identifier, and $op = add/del$. Additions are stored in the first deployment and deletions in the second one. For searching, the user queries both deployments, retrieves all entries, removes the deleted ones locally, and requests the files with the remaining identifiers from the server. In that way, the server cannot tell which identifiers were deleted.

Diana$_{del}$ and Janus are Type-III schemes that rely on puncturable cryptographic primitives to achieve better results, by increasing the amount of information leaked. Diana$_{del}$ uses a puncturable pseudorandom function [4, 8, 29] to achieve better concrete performance than Fides. Janus uses puncturable encryption [22] which allows search queries to be executed with a single round of interaction. Finally, Bost et al, presented Moneta, the only existing Type-I scheme so far. However, its construction is based on the recent ORAM scheme of [20] which uses garbled circuits to avoid interaction. This somewhat limits its potential for adoption in practice, due to the concrete communication overhead, and it serves mostly as a theoretical result for the feasibility of Type-I schemes.

1.1 Our Results

In this work, we present three SSE schemes with forward and backward privacy. Our schemes improve the results of [7] in several ways. A comparison can be seen in Table 1.

Fast Type-II backward privacy. Our first scheme, Mitra (Section 3), offers backward-privacy Type-II. Asymptotically, it achieves the same performance as Fides, however, due to the use of symmetric encryption our experimental evaluation indicates that it has 145 – 253× better computation time for searches and 86 – 232× for updates. Surprisingly, Mitra has better overall performance than Diana $_{del}$ and Janus which only achieve Type-III backward privacy, which makes Mitra the most efficient existing forward and backward private SSE. We believe the combination of its low leakage level, practical performance, and simplicity of design, make it a great candidate for adoption in practice.

Optimizing the search time. As can be seen in Table 1, all existing schemes (including Mitra) impose search time of $\Omega(a_w)$ where a_w is the total number of updates related to w (clearly $n_w d_w > a_w$ where n_w is the number of documents containing w *currently* and d_w is the number of previous deletions for w). This in practice can be very far from the optimal cost which is $O(n_w)$. Inspired by this, we explore whether backward-private SSE schemes that have *optimal* ($O(n_w)$) or *quasi-optimal* ($O(n_w \cdot \text{polylog}(N))$) search time exist. Crucially, not even Moneta, the construction from [7] that relies on ORAM achieves this property. Furthermore, even when examining schemes that are only forward-private, the only known quasi-optimal construction is from Stefanov et al. [43].

We answer the above question in the affirmative by providing two schemes, Orion (Section 4) and Horus (Section 4.3), which have quasi-optimal search time. Orion achieves the strongest level of backward privacy, Type-I. Asymptotically, it requires $O(\log N)$ rounds of interaction and the search process takes $O(n_w \log^2 N)$ steps. However, these asymptotics hold even when no deletions have occurred (i.e., $n_w = a_w$) which motivated us to develop our last scheme HORUS, that improves the efficiency of ORION achieving better search performance and reduced rounds of interaction. The number of roundtrips for a search of w is only $O(\log d_w)$. In particular, if no (or a constant number of) such deletions have taken place HORUS requires $O(1)$ roundtrips to retrieve the file identifiers. On the other hand, HORUS is only backward private Type-III which, however, may still be sufficient in many applications.

Section 5 contains our experimental evaluation and a comparison with the performance of the constructions of [7].

Overview of techniques. Our first scheme uses an approach for maintaining an encrypted index that has been extensively used in the literature of dynamic SSE (e.g., [19, 31, 42]). In particular, MITRA has similarities with the recent construction of Etemad et al. [19] that is only forward-private. Triplets of the form keyword/document/operation (w, id, op) are stored encrypted in a map dictionary, using pseudorandomly generated locations, based on the counter of updates upd_{cnt} for w which is maintained locally. Note that $op = add/del$, i.e., even deletions are "stored" in this manner. Every update accesses locations that appear random, as far as the server can tell. To retrieve the files for w, the client simply regenerates the pseudorandom positions for the different counter values $1, \ldots, upd_{cnt}$ for w. In this way, the server does not learn the identifiers of deleted entries.

ORION again maintains a map data structure where each entry (w, id) is looked up using as key the pair (w, upd_{cnt}). Every time a new entry (w, id) is inserted upd_{cnt} is incremented. When an entry (w, id) is deleted, the entry corresponding to the maximum upd_{cnt} value for w is "swapped" to the position corresponding to upd_{cnt} (see Figure 2). In this manner, at all times, the correct result for a search for w can be retrieved by looking up *certain positions*: $(w, 1), \ldots, (w, n_w)$. Performing this swapping during updates requires being able to lookup the position (w, upd_{cnt}) while only knowing (w, id). This is the opposite direction than the one offered by the map, therefore we need to use a second map that supports this type of mapping. ORION achieves forward privacy and backward privacy Type-I, by using two *oblivious map (OMAP)* data structures [46] which hide the actual accessed memory locations from the server. Naively implemented, this approach would require $\Omega(n_w)$ rounds of interaction for search, one for retrieving each entry (w, i), in order to maintain privacy. However, a close observation of the OMAP instantiation of Wang et al. [46] reveals that it can support "batch" query execution with the same number of roundtrips as for the case of a single one, i.e., $O(\log N)$.

HORUS reduces the obligatory amount of interaction by replacing one of the oblivious maps of ORION (the one that is only used during searches to retrieve results) with a non-recursive (one-level) *Path-ORAM* structure [44]. Normally, this approach would require $O(N)$ storage at the client in order to maintain the position map for the ORAM which invalidates outsourcing the database in the first place. We replace the randomly generated ORAM positions with ones generated with a pseudorandom function (PRF), thus the client only needs to store the PRF key. In order to achieve forward and backward privacy, we need to ensure that the same input is never consumed by the PRF more than once throughout the execution of updates. This is done by introducing an additional access counter for each upd_{cnt} that measures the number of times the content of the location corresponding to (w, upd_{cnt}) was edited by an update (acc_{cnt}). This solves the privacy issues, but it introduces a new problem: The client needs to know the correct acc_{cnt} for each $upd_{cnt} = 1, \ldots, n_w$ during a search for w. We solve this issue by having the client perform n_w *binary searches* executed "in parallel" in order to identify the n_w correct acc_{cnt} values. We show that with HORUS the maximum value of acc_{cnt} is $O(d_w)$ therefore the number of necessary roundtrips is reduced to $O(\log d_w)$. However, this distribution of previously accessed positions reveals (during search operations) to the server how many times acc_{cnt} was incremented for different upd_{cnt} values, which in turn leaks information about which previous addition was canceled by each deletion.

1.2 Related Work

SSE was first introduced by Song et al. [41] who proposed a linear-time search construction. Curtmola et al. [15] proposed the modern security definition of SSE that also introduced a sublinear-time construction. Chase and Kamara [14] introduced the broader notion of structured encryption to model encryption schemes that allow for controlled disclosure of some predicate of the data; searchable encryption is a specific type of structured encryption.

The first schemes to explicitly support efficient updates in the database were by Kamara et al. [28] and Kamara and Papamanthou [27], with the latter reducing the amount of leakage of the first one. However, neither of them had forward privacy, which was first introduced as a notion in [13]. Since then, efficient dynamic SSE schemes with forward privacy have been studied extensively and there are numerous works that proposed improved constructions [5, 9, 19, 20, 23, 30, 31, 35, 43]. Stefanov et al. [43] were the first to introduce backward privacy to capture leakage related to deleted entries without, however, providing a definition or a construction. The first (and only, to the best of our knowledge) work that focused on backward privacy and defined the notion as well as offered backward private schemes was the recent work of Bost et al [7]. Other work on SSE has focused on more expressive queries [10, 14, 16, 25, 26, 32], multiuser settings [37, 38], constructions that perform well for data on-disk [2, 11, 17, 33], and combining SSE with ORAM [20, 35].

2 CRYPTOGRAPHIC BACKGROUND

We introduce here the necessary notations and definitions that will be used in the paper. We denote by $\lambda \in \mathbb{N}$ a security parameter. PPT stands for probabilistic polynomial-time. By $v(\lambda)$ we denote a negligible function in λ. In a two-party protocol P execution between a client and a server, the notation $P(x; y)$ means that x is the client's input and y is the server's input.

Assume a collection of D documents with identifiers $id_1, \ldots, id_D$, each of which contains textual keywords from a given alphabet Λ. We consider the database $\mathbf{DB}$ that consists of pairs of file identifiers and keywords, such that pair $(id_i, w) \in \mathbf{DB}$ if and only if the file with identifier id_i contains keyword w. Let W denote the set of all keywords that appear in $\mathbf{DB}$, $|W|$ the number of distinct keywords, N the number of document/keyword pairs (i.e., $N = |\mathbf{DB}|$), and $\mathbf{DB}(w)$ denote the set of documents that contain keyword w.

Pseudorandom functions. Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $G : \{0,1\}^\lambda \times \{0,1\}^\ell \rightarrow \{0,1\}^{\ell'}$ be a pseudorandom function (PRF) family. $G_K(x)$ denotes $G(K, x)$. G is a secure PRF family if for all PPT adversaries Adv, $|\Pr[K \leftarrow Gen(1^\lambda); \mathsf{Adv}^{G_K(\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)}(1^\lambda) = 1]| \leq v(\lambda)$, where $R : \{0,1\}^\ell \rightarrow \{0,1\}^{\ell'}$ is a truly random function.

Searchable encryption. A *dynamic symmetric searchable encryption scheme (SSE)* $\Sigma = (Setup, Search, Update)$ consists of algorithm *Setup*, and protocols *Search*, *Update* between a client and a server:

- $Setup(\lambda)$ is an algorithm that on input the security parameter outputs (K, σ, EDB) where K is a secret key for the client, σ is the client's local state, and EDB is an (empty) encrypted database that is sent to the server. In the following we sometimes use the notation $Setup(\lambda, N)$ to refer to a setup process that takes a parameter N for the maximum number of entries.
- $Search(K, q, \sigma; EDB)$ is a protocol for searching the database. In this paper, we only consider search queries for a single keyword i.e., $q = w \in \Lambda^*$. The client's output is $DB(w)$ (empty if $w \notin W$). The protocol may also modify K, σ and EDB.
- $Update(K, op, in, \sigma; EDB)$ is a protocol for inserting an entry to or removing an entry from the database. Operation op can be *add* or *del*, input *in* consists of a file identifier *id* and a keyword w. The protocol may modify K, σ and EDB.

In the above, we followed the definition of [5, 7] with minor modifications. Given the above API, on input the data collection the client can run *Setup*, followed by N calls to *Update* to "populate" *EDB*. Other works (e.g., [19]) follow a different but functionally equivalent approach that offers a single build operation for this. Similarly, some existing works [19, 30] have *Update* take as input an entire file for addition, or the file identifier for deletion and the protocol adds/removes all the relevant keywords to/from the database. Again, this is functionally equivalent as this process can be decomposed to multiple calls of the above *Update* protocol. Finally, we implicitly assume that after receiving the indexes $\mathbf{DB}(w)$, the client performs an additional operation to retrieve the actual files; we omit this step from when describing our constructions.

Informally, an SSE is correct if the returned result is correct for every query (for a formal definition, we refer readers to [9]). The confidentiality of an SSE scheme is parametrized by a leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ which captures the information that is revealed to an adversarial server throughout the protocol execution. $\mathcal{L}^{Stp}$ corresponds to leakage during setup, $\mathcal{L}^{Srch}$ to leakage during a search operation, and $\mathcal{L}^{Updt}$ to leakage during updates. Informally, a secure SSE scheme with leakage $\mathcal{L}$ should reveal nothing about the database $\mathbf{DB}$ other than this leakage.

This is formally captured by a standard real/ideal experiment with two games $\mathsf{Real}^{\mathsf{SSE}}$, $\mathsf{Ideal}^{\mathsf{SSE}}$ presented in Figure 6 in Appendix A, following the definition of [43].

Definition 2.1 ([44]). An SSE scheme Σ is adaptively-secure with respect to leakage function $\mathcal{L}$, *iff* for any PPT adversary Adv issuing polynomial number of queries q, there exists a stateful PPT simulator Sim = $(SimInit, SimSearch, SimUpdate)$ such that $|\Pr[\mathsf{Real}^{SSE}_{\mathsf{Adv}}(\lambda, q) = 1] - \Pr[\mathsf{Ideal}^{SSE}_{\mathsf{Adv},\mathsf{Sim},\mathcal{L}}(\lambda, q) = 1]| \leq v(\lambda)$.

Forward and backward privacy. Forward and backward privacy are two SSE properties that aim to control what information is leaked by dynamic schemes in relation to updates that take place. Informally, a scheme is *forward private* if it is not possible to relate an update that takes place to previous operations, at the time during which it takes place. This is particularly useful in practice, e.g., to hide whether an addition is about a new keyword or a pre-existing one (which may have been previously searched for).

Definition 2.2. An $\mathcal{L}-adaptively-secure$ SSE scheme that supports addition/deletion of a single keyword is forward private *iff* the update leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'^{Updt}(op, id)$ where $\mathcal{L}'$ is a stateless function, op is insertion or deletion, and id is a file identifier.

Backward privacy aims to limit the information that the server can learn when executing a search for a keyword w for which some entries have previously been deleted. Ideally, the SSE scheme should reveal nothing to the adversary about these previously deleted entries, and at least not the file identifiers of the deleted entries [43]. A formal definition was given in [7] for three different types of backward privacy with varying leakage patterns, from Type-I which reveals the least information to Type-III which reveals the most. Before we give the final definition, we provide some additional functions that will be necessary, following the notation of [7].

Consider a list Q that has one entry for each query executed. The entry for a search is of the form (u, w) where u is the query timestamp and w is the searched keyword. That of an update is $(u, op, (w, id))$ where $op = add/del$ and id is the modified file. For a keyword w, let $\mathbf{TimeDB}(w)$ be a function that returns the list of all timestamp/file-identifier pairs of keyword w that have been added to $\mathbf{DB}$ and have not been subsequently deleted.

$$\mathbf{TimeDB}(w) = \{(u, id) \mid (u, add, (w, id)) \in Q \text{ and } \forall u', (u', del, (w, id)) \notin Q\}$$

$\mathbf{Updates}(w)$ is a function that returns the timestamp of all insertion and deletion operations for w in Q. Formally, $\mathbf{Updates}(w) = \{u | (u, add, (w, id)) \in Q \text{ or } (u, del, (w, id)) \in Q\}$. Finally, $\mathbf{DelHist}(w)$ is a function that returns the history of deleted entries by giving all (insertion timestamp, deletion timestamp) pairs to the adversary. Most importantly, it reveals explicitly which deletion corresponds to which addition.

$$\mathbf{DelHist}(w) = \{(u^{add}, u^{del}) \mid \exists\, id : (u^{add}, add, (w, id)) \in Q \text{ and } (u^{del}, del, (w, id)) \in Q\}$$

As can be seen, the above functions' leakage is progressively increasing. We are now ready to formally define backward privacy with different types of leakage.

Definition 2.3 ([7]). An $\mathcal{L}$-adaptively-secure SSE scheme has backward privacy:

Type-I (BP-I): *iff* $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathbf{TimeDB}(w), a_w)$.

Type-II (BP-II): *iff* $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathbf{TimeDB}(w), \mathbf{Updates}(w))$.

Type-III (BP-III): *iff* $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, w)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathbf{TimeDB}(w), \mathbf{DelHist}(w))$.

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

Note that the above definition assumes schemes leak the documents that currently contain w in order to account for the leakage from actually retrieving the files. Namely, **TimeDB**(w) function explicitly reveals the indexes of returned documents.

Oblivious Maps. A data structure $\mathcal{D}$ is a collection of data supporting certain types of operations such as insertions, deletions, or lookups. Each type of operation receives corresponding arguments (e.g., the key of the value to lookup). An *oblivious data structure* is a privacy-preserving version of $\mathcal{D}$ that aims to hide the type and content of a sequence of operations performed on the data structure. Intuitively, for any two possible sequences of k operations, their resulting access patterns (i.e., the sequence of memory addresses accessed will executing the operations) must be indistinguishable. (See [46] for a formal definition).

Our Orion and Horus constructions use as a building block an *oblivious map (OMAP)* that is a key/value oblivious data structure, implemented with an AVL tree as per the instantiation of [46]. To make the map oblivious, it uses a non-recursive Path-ORAM [44] structure that stores the set of nodes. (We provide a more detailed description of Path-ORAM in Appendix B for the non-expert readers.) Each tree node contains the following information $node = ((id, data), pos, childrenPos)$ where $data$ is a value for the map, id is a key for the map, pos indicates the node's leaf number in Path-ORAM, and $childrenPos$ is an array which stores the Path-ORAM leaf numbers for the node's children.

OMAP offers three protocols $Setup$, $Find$, and $Insert$ for initializing the structure, retrieving the value for a given key, and inserting a key/value pair. We describe them in detail in Appendix C. At a high level, $Setup$ initializes a Path-ORAM T and stores an empty node for the root of the AVL tree at a randomly chosen position $rootID$. Subsequent $Find$, $Insert$ calls, traverse the AVL tree from the root in order to find/insert a matching node. Each node traversal requires a separate ORAM access. The ORAM position for a child node is stored at the parent. Finally, all accessed nodes are re-encrypted and mapped to fresh random positions before being stored again at T. For insertions, an AVL tree rebalancing process is executed, again via ORAM read/write accesses.

From the analysis of the AVL tree, each access retrieves $O(\log N)$ nodes for a tree of N nodes, each of which requires retrieving a $\log N$ path from T, thus the overall performance is $O(\log^2 N)$. Unlike standard non-recursive Path-ORAM, the client performing the accesses only needs to remember the leaf position of the root of the AVL tree (the ORAM stash of size $O(\log^2 N)$ can be downloaded with every access without affecting the asymptotic behavior) i.e., he needs $O(1)$ storage. An example operation of this OMAP is presented in Figure 1. In order to access node 7, first the root node 6 is retrieved (since the client remembers its ORAM position). Then by comparison with the children values 4, 9 the client decides to retrieve the latter, whose position he read from the root node. Then he repeats this process and picks the left child of node 9. Finally, the content of the accessed nodes 6, 9, 7 is updated from bottom-up with new ORAM positions randomly chosen, the nodes are re-encrypted, permuted and stored in the ORAM.

Since the searched key may be found at any layer, an operation can terminate early which may reveal information to the server. To avoid this, [46] pads the number of ORAM accesses with "dummy" operations that do not alter the tree structure but are indistinguishable from the server's point of view.

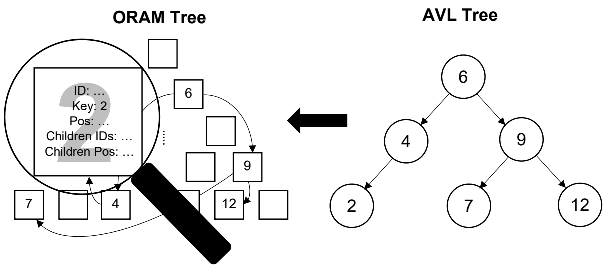


Figure 1: Sample AVL Tree and its Path-ORAM [46].

3 MITRA: A SIMPLE FORWARD AND BACKWARD PRIVATE SCHEME

In this section, we propose Mitra, our first backward and forward private scheme. Mitra follows a simple approach for storing encrypted records in a manner that leaks nothing to the server during updates (insertions and deletions), and only the time at which updates took place during searches. The construction uses a key-value dictionary that stores encrypted values of the form (id, op) where op is insertion or deletion, and id is the identifier of a specific file related to this operation. The keys (locations at which values are stored in the dictionary) are generated via a pseudorandom function in a way that guarantees the client can efficiently generate the set of all locations related to the specific keyword w for a given search operation. Compared to [19] which introduced a similar construction that is only forward private, the main difference of Mitra that makes it backward private is that, instead of sending to the server the key that allows him to generate the location and decrypt the entries for w, we send him the locations directly and the decryption happens locally at the client.

Setup. The setup algorithm (Algorithm 1) generates a secret key K on input the security parameter λ. The client initiates two empty maps (**DictW**,**FileCnt**). The first is sent to the server in order to store encrypted entries whereas **FileCnt** is stored locally.

Update. In the update procedure (Algorithm 2) the client receives keyword w, file identifier id, and corresponding operation $op = add/del$. For example input (add, w, id) means *"add an entry for keyword w in file id"*. The client also has access to the key K and local state **FileCnt** that stores for each distinct keyword w a counter that denotes how many updates have taken place in relation to w. First, the client checks whether **FileCnt**[w] has been initialized or not. In the latter case he sets the counter value of w to 0. In both cases, he increments the counter by 1 (lines 1-4). Next, the client runs the PRF G with key K twice, and computes $G_K(w, \textbf{FileCnt}[w]||0)$ and $G_K(w, \textbf{FileCnt}[w]||1)$. The first PRF output is used as the key $addr$ in which the encrypted value for $(id||op)$ will be stored at the server, whereas the second PRF output is XORed with the entry $(id||op)$ and the result becomes the encrypted value val which will be stored by the server (lines 5-6). The pair $(addr, val)$ is sent to the server who stores them as $\textbf{DictW}[addr] = val$ (line 7).

Search. Finally, we describe the search process (Algorithm 3). While searching for all files containing keyword w, the client first looks up the counter **FileCnt**[w] which is the total number of updates

Algorithm 1 Mitra *Setup*(λ)

```
1: $K \leftarrow Gen(1^\lambda)$
2: **FileCnt**,**DictW** ← empty map
3: $\sigma \leftarrow$ **FileCnt**
4: $EDB \leftarrow$ **DictW**
5: Send $EDB$ to the server
```

Algorithm 2 Mitra *Update*($K, op, (w, id), \sigma; EDB$)

```
Client:
1: if FileCnt[$w$] is NULL then
2:     FileCnt[$w$] = 0
3: end if
4: FileCnt[$w$]++
5: $addr = G_K(w, \mathbf{FileCnt}[w]||0)$
6: $val = (id||op) \oplus G_K(w, \mathbf{FileCnt}[w]||1)$
7: Send $(addr, val)$ to the server
Server:
8: Set $\mathbf{DictW}[addr] = val$
```

Algorithm 3 Mitra *Search*($K, w, \sigma; EDB$)

```
Client:
1: TList = { }
2: for $i = 1$ to FileCnt[$w$] do
3:     $T_i = G_K(w, i||0)$
4:     TList = TList $\cup \{T_i\}$
5: end for
6: Send TList to the server
Server:
7: $F_w = \{\}$
8: for $i = 1$ to TList.size do
9:     $F_w = F_w \cup \mathbf{DictW}[\text{TList}[i]]$
10: end for
11: Send $F_w$ to the client
Client:
12: $R_w = \{\}$
13: for $i = 1$ to $F_w.size$ do
14:     $(id||op) = F_w[i] \oplus G_K(w, i||1)$
15:     $R_w = R_w \cup (id||op)$
16: end for
17: Remove ids that have been deleted from $R_w$
18: return $R_w$
```

related to w. He then generates a list TList of all the locations at which the corresponding entries are stored in **DictW** at the server. This is done by evaluating the PRF G on input $G_K(w, i||0)$ for $i = 1, \ldots,$ **FileCnt**[w]. Note that these are the same locations that were computed during the previous updates for w, since G is a deterministic function. The list TList of locations is then sent to the server (lines 1-6). The server retrieves from **DictW** the values of all keys from TList and sends them back to the client (lines 7-11). Upon receiving these encrypted values, the client decrypts them by computing the PRF outputs $G_K(w, i||1)$ for $i = 1, \ldots,$ **FileCnt**[w] and XORing the i-th of them with the i-th encrypted value.

Security analysis. Our scheme achieves forward privacy and backward privacy Type-II. Forward privacy follows immediately since the two values $(addr, val)$ that the server observes during an update are indistinguishable from random, due to the pseudorandomness of G and the fact that during each update a different input is consumed by the PRF. Indeed, the server does not even learn the type of operation that is executed (addition/deletion), i.e., the update leakage is empty. For backward privacy, note that during a search for w the server will receive a number of PRF evaluations which he has seen previously during updates. This immediately reveals when each update operation for w took place. Beyond this, nothing else is revealed to the server; in particular the server does not learn which deletion cancels which addition. By the backward privacy definitions of Section 2, this leakage corresponds to BP-II.

We are now ready to state the following theorem regarding the security of Mitra (full proof is provided in Appendix D).

Theorem 3.1. *Assuming G is a secure PRF, Mitra is an adaptively-secure SSE scheme with* $\mathcal{L}^{Updt}(op, w, id) = \perp$ *and* $\mathcal{L}^{Srch}(w) = (TimeDB(w), Updates(w))$.

Efficiency of Mitra. The asymptotic performance of updates is clearly $O(1)$ for both client and server as they entail a constant number of operations. The same is true for the communication size as a single pair of values is transmitted. For search operations, Mitra requires $2 \cdot$ **FileCnt**[w] PRF evaluations and **FileCnt**[w] XOR operations from the client, and **FileCnt**[w] look-ups from the server. Recall that **FileCnt**[w] counts the total number of updates for w which, using our notation from Section 2, is denoted by a_w. Therefore, the overall asymptotic complexity of the search operation is $O(a_w)$ and the same is true for the communication (as the size of the communicated lists is a_w). The storage at the server after N updates have taken place is $O(N)$ since one entry is added to **DictW** for each of them. The permanent storage at the client is $O(|W| log|D|)$, where $|W|$ is the total number of keywords and $|D|$ is the total number of documents, since one counter is stored for each keyword. Finally, Mitra requires a single roundtrip to retrieve the file identifiers for w. If the actual files need to be retrieved, this would take one more round of interaction.

In terms of concrete performance, Mitra is extremely fast both for updates and searches. As we experimentally demonstrate in Section 5, it significantly outperforms Fides from [7] that achieves the same level of leakage (BP-II), both in terms of speed and communication bandwidth. In fact it is comparable, and often more performant than schemes that only achieve BP-III. Note that Mitra may be especially attractive for the server as no cryptographic operations take place there. This not only improves server performance (which may be very important in practice, e.g., in the case of a cloud server where resources are shared across tenants) but also makes the deployment of Mitra very easy.

Removing Deleted Entries. With Mitra the size of EDB grows with every update, including deletions. This is a common approach for building dynamic SSE schemes and has been extensively used in the literature as an easy method for handling deletions, e.g., [5, 7, 9, 19, 43]. If one wants to avoid this, one technique adopted by previous schemes is a periodic "clean-up" operation, executed during searches. Mitra can also be modified in a similar way. This is done by having the client remove deleted entries after a search, re-encrypting the remaining ones, and sending them back to the server. Using the encryption of Mitra, which is deterministic, this would produce the same ciphertexts, violating privacy. To avoid this, we maintain an extra counter map **SrcCnt** that is incremented after every search and is also given as input to the PRF. We call the resulting scheme Mitra*. Asymptotically it has the same performance as Mitra and the same backward privacy type. For completeness, its construction is described and proven secure in the extended version of our paper.

4 ORION: BACKWARD AND FORWARD PRIVATE SSE WITH QUASI-OPTIMAL SEARCH TIME

We now present Orion and Horus, two SSE schemes that are backward-private and whose search complexity is only proportional to the number of files n_w containing the searched keyword w, currently in the database. The first one achieves very strong backward privacy Type-I, i.e., it only reveals the number and type of updates related to a specific keyword. The second one achieves backward-privacy Type-III but it has reduced interaction, computation, and communication during search. We first describe a "straw-man" solution to highlight the difficulties in achieving schemes with (quasi-)optimal search time and non-trivial interaction. Consider the following definition.

Definition 4.1. We say that a dynamic symmetric searchable encryption scheme Σ has *optimal* (resp. *quasi-optimal*) search time, if the asymptotic complexity of *Search* is $O(n_w)$ (resp. $O(n_w \cdot poly\text{-}log(N))$). We say that Σ has *non-trivial interaction*, if it requires $o(n_w)$ rounds of interaction during *Search* (else we say Σ has *trivial* interaction).

We note here that none of the existing backward and forward private schemes achieves this property, not even the Moneta construction from [7] which utilizes ORAM to achieve strong backward privacy, as it has a search time of $\tilde{O}(a_w \log N + \log^3 N)$. Even if we restrict our attention to schemes that are only forward private, the only known construction with quasi-optimal search time is the work of Stefanov et al. [43] that achieves $O(n_w \log^3 N)$ via the use of an elaborate multi-layered data structure.

4.1 A Warm-up Solution

Inspired by the dictionary data structure of Mitra, consider a similar scheme where the location of each entry in the dictionary is now computed based on (w, id) (whereas in Mitra, it was computed from $(w, \mathbf{FileCnt}[w])$). The value of this entry will then be a "pointer" to the previous and next file identifiers related to w. The first time a file id for w is inserted, the previous and next file identifier values are null, and the client stores locally id as the latest file identifier for w. Then, when another file id' that contains w is inserted, the client adds to the dictionary a corresponding entry with id as the previous identifier and null as the next identifier. He also replaces id with id' in his local storage. Finally, he updates the previous entry for id, setting its next identifier to id'.

This allows the client to traverse the dictionary like a list. During search, he needs to remember only the latest file identifier for w. He accesses the corresponding dictionary entry, retrieves the previous identifier, and repeats the process until the previous identifier is null. Deletions can be handled in the standard manner for doubly-linked lists. The client first looks up the dictionary position for (w, id) (assuming he wants to remove the entry for keyword w in file id), retrieves the previous and next file identifiers id', id'', and then looks up the dictionary for $(w, id'), (w, id'')$. He then sets the next identifier at the entry for id' to id'' and the previous entry for id'' to id', "eliminating" the in-between entry for id.

Using an oblivious map. As described, the above approach reveals the accessed locations during updates and searches. E.g., if an entry is added for w, the location of the previous latest entry for w is revealed, trivially violating forward privacy. Locations accessed during a search can also be related to previously accessed ones during updates, e.g., leaking in this manner information related to deleted entries, which violates backward privacy. To avoid this leakage, the entire dictionary data structure can be instantiated with an *oblivious map* as defined in Section 2. In this manner, the sequence of locations accessed throughout the protocol is indistinguishable from random ones. Assuming the oblivious map implementation of [46], the resulting search time is $O(n_w \log^2 N)$. On the other hand, these accesses need to take place sequentially, as the next location is only revealed after accessing the previous one. This requires the assistance of the client, in order to decrypt each entry and compute the next location to be accessed, therefore it would take at least n_w rounds of interaction ($O(n_w \log N)$ using the scheme of [46]), resulting in a scheme with trivial interaction.

Challenge: Reducing interaction. Using the insights gained from our straw-man solution, oblivious data structures can readily lead to schemes with quasi-optimal search times. The remaining problem is how to reduce the interaction during search, as it may not be reasonable to accept in practice a number of roundtrips that grows with n_w. Looking under the hood of the oblivious map construction of [46], it builds a map using an AVL tree. The nodes of the tree are then stored in a non-recursive (one-level) Path-ORAM construction (see Appendix B). Each tree node contains not only its children's values but also their positions in the Path-ORAM, thus a map lookup is reduced to $O(\log N)$ ORAM accesses, one for every level of the AVL tree. Garg et al. [20] recently showed how to make Path-ORAM accesses non-interactive. With their approach, each AVL node would be fetched with a single interaction. However, they rely on garbled circuits [3] to avoid interaction, and each access consumes $O(\log N)$ garbled circuits that need to be replaced by freshly encrypted ones by the client before the next map access. Thus, even with that approach, performing the n_w map lookups necessary for our straw-man scheme would require $n_w \log N$ roundtrips, i.e., again resulting in a scheme with trivial communication.

4.2 Orion Construction

The basic idea behind Orion is to spend a little more time rearranging the entries during a delete operation, in order to facilitate subsequent searches. Similar to our straw-man solution, we will rely on oblivious maps to hide the accessed locations. We will use two of them: $OMAP_{upd}$ that is only accessed during update operations and $OMAP_{src}$ that is accessed during both updates and searches.

High-level overview. The client maintains a counter $updt_{cnt}$ for each keyword w that shows the number of files containing that

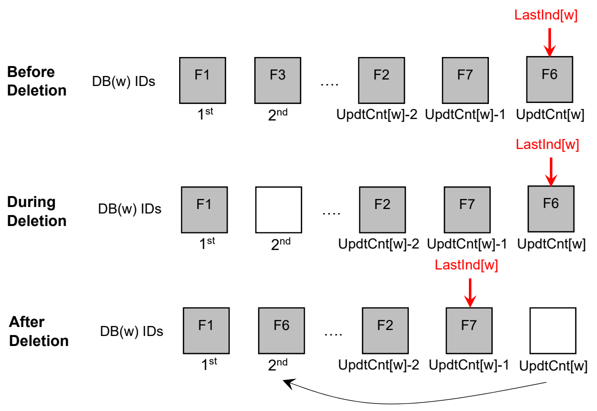


Figure 2: Sample deletion process for Orion.

Algorithm 4 Orion $Setup(\lambda, N)$

```
1: UpdtCnt, LastInd ← empty map
2: $(T, rootID) \leftarrow OMAP_{src}.Setup(1^\lambda, N)$
3: $(T', rootID') \leftarrow OMAP_{upd}.Setup(1^\lambda, N)$
4: $\sigma \leftarrow (rootID, rootID', \mathbf{UpdtCnt}, \mathbf{LastInd})$
5: $EDB \leftarrow (T, T')$
6: Send $EDB$ to the server
```

keyword *currently* in the database (initialized to 0, incremented after insertions, decremented after deletions). For an insertion, the client stores in $OMAP_{upd}$ a mapping from (w, id) to the corresponding $updt_{cnt}$ and a mapping from $(w, updt_{cnt})$ to id in the search $OMAP_{src}$. That is, the first oblivious map is accessed by file identifier (necessary for deleting specific entries) whereas the second one is accessed by $updt_{cnt}$. What allows Orion to handle searches more efficiently, is the way it handles deletions. For deleting the entry (w, id), the client first performs a look-up in $OMAP_{upd}$ to receive the corresponding update counter u. Then, he inserts to $OMAP_{src}$ an entry for (w, u) with value id', where id' is the identifier of the most recently inserted file for keyword w, i.e., the one retrieved by looking up (w, upd_{cnt}) from $OMAP_{src}$. Simply put, he swaps the deleted item with the right-most one in Figure 2. Finally, he looks up $(w, upd_{cnt} - 1)$ from $OMAP_{src}$ to retrieve the correct id in preparation for the next update query. What is particularly useful about this way of handling deletions is that it guarantees that, any given time, the n_w file identifiers corresponding to current documents containing w can be retrieved by looking up the entries $(w, 1) \ldots, (w, n_w)$.

The final missing piece to reduce the rounds of interaction is the observation that the oblivious map construction of [46] can handle "batch" queries without breaking the obliviousness property. A single query takes $O(\log^2 N)$ time and $O(\log N)$ rounds of interaction. Executing n_w queries in batch take $O(n_w \log^2 N)$ time, which yields quasi-optimal search time, and $O(\log N)$ rounds of interaction, which gives Orion non-trivial communication.

The procedures of Orion are presented in detail in Algorithms 4-6. Update and search are presented from the view of the client and blue lines correspond to oblivious map queries. Each such line corresponds to an interactive protocol, as described in Section 2.

Algorithm 5 Orion $Update(K, op, (w, id), \sigma; EDB)$

```
1: $mapKey = (w, id)$
2: $(rootID', updt_{cnt}) \leftarrow OMAP_{upd}.Find(mapKey, rootID')$
3: if $op = add$ then
4:     if $updt_{cnt}$ = NULL or $updt_{cnt} = -1$ then
5:         if UpdtCnt[$w$] is NULL then
6:             UpdtCnt[$w$] = 0
7:         end if
8:         UpdtCnt[$w$]++
9:         $data = ((w, id), \mathbf{UpdtCnt}[w])$
10:        $rootID' \leftarrow OMAP_{upd}.Insert(data, rootID')$
11:        $data = ((w, \mathbf{UpdtCnt}[w]), id)$
12:        $rootID \leftarrow OMAP_{src}.Insert(data, rootID)$
13:        LastInd[$w$] = $id$
14:    end if
15: else if $op = del$ then
16:    if $updt_{cnt} > 0$ then
17:        $data = (mapKey, -1)$
18:        $rootID' \leftarrow OMAP_{upd}.Insert(data, rootID')$
19:        UpdtCnt[$w$]--
20:        if UpdtCnt[$w$] > 0 then                ▷ There are entries for $w$
21:            if UpdtCnt[$w$] + 1 ≠ $updt_{cnt}$ then
22:                $data = ((w, \mathbf{LastInd}[w]), updt_{cnt})$
23:                $rootID' \leftarrow OMAP_{upd}.Insert(data, rootID')$
24:                $data = ((w, updt_{cnt}), \mathbf{LastInd}[w])$
25:                $rootID \leftarrow OMAP_{src}.Insert(data, rootID)$
26:            end if
27:            $key = (w, \mathbf{UpdtCnt}[w])$
28:            $(rootID, lastID) \leftarrow OMAP_{src}.Find(key, rootID)$
29:            LastInd[w] = $lastID$
30:        else
31:            LastInd[$w$] = 0
32:        end if
33:    end if
34: end if
35: Execute necessary dummy oblivious map accesses
```

Algorithm 6 Orion $Search(K, w, \sigma; EDB)$

```
1: $R = \{\}$
2: for $i = 1$ to UpdtCnt[$w$] do                ▷ Execute in batch
3:     $(rootID, id) \leftarrow OMAP_{src}.Find((w, i), rootID)$
4:     $R = R \cup \{id\}$
5: end for
6: return $R$
```

Setup. During setup, the client initializes two empty maps **UpdtCnt**, **LastInd**. The first stores the last $updt_{cnt}$ value of each keyword (corresponding to the number of files currently in the database containing the keyword) and the second stores the most recent file identifier inserted for each keyword. The client also sets up two oblivious maps. $OMAP_{src}$ maintains a mapping $(w, updt_{cnt}) \rightarrow id$, i.e., on input a keyword and an update counter it returns the corresponding file identifier. $OMAP_{upd}$ stores a mapping of $(w, id) \rightarrow updt_{cnt}$, i.e., on input a keyword and a file identifier it outputs the update counter of the corresponding entry (negative if the entry has been previously deleted). The encrypted database EDB

consists of the two oblivious maps, and the local state σ contains **UpdtCnt**, **LastInd**. The secret key K is (implicitly) set to the secret encryption keys of the oblivious maps.

Update. The client first makes an oblivious access to $OMAP_{upd}$ to retrieve the update counter for the pair (w, id) corresponding to the update. We then distinguish the case of addition and deletion. For addition, the client first sets the new value of **UpdtCnt**$[w]$ (lines 5-8) and then makes two oblivious accesses: (i) to $OMAP_{upd}$ to insert a mapping from (w, id) to the **UpdtCnt**$[w]$, (ii) to $OMAP_{src}$ to insert a mapping from $(w,$ **UpdtCnt**$[w])$ to id (lines 9-12). Finally, he sets **LastInd**$[w]$ to the newly added id.

For deletion operations, the client first updates the entry for (w, id) in $OMAP_{upd}$ to indicate the entry has been deleted (lines 17,18). He then decreases **UpdtCnt**$[w]$ by one to indicate that fewer files contain w now. If there are more files containing w (line 20), the client has to update the mappings which is done by two oblivious map accesses. The first (lines 22,23) is to $OMAP_{upd}$ in order to indicate that the previous latest entry for w is moved to the position that was vacated after the deletion. The second (lines 24,25) is matching modification for $OMAP_{src}$. Finally, the client fetches from $OMAP_{src}$ the identifier of the current latest insertion for w to update the entry **LastInd**$[w]$ (in preparation for future updates). In case there were no more entries for w, the client does not make these accesses and simply sets **LastInd**$[w] = 0$.

Eventually, the client performs a number of dummy OMAP accesses, if necessary, to hide data-dependent paths of the code. That is, for additions he guarantees to always make two calls to $OMAP_{upd}$ and one to $OMAP_{src}$ whereas for deletions the corresponding numbers are three and two.

Search. Due to the extra effort made during updates, the search operation is very simple for the client. He first retrieves the number of files currently containing w, that is, **UpdtCnt**$[w] = n_w$. Then he needs to issue n_w oblivious accesses to $OMAP_{upd}$. Importantly, these accesses can be executed in batch, without having to wait for the first in order to begin the next one. For example, with reference to Figure 1, to search for nodes 2, 7 the client could fetch root node 6, compare the children values to deduce he needs both left 4, and right child 9, retrieve them and repeat this process to get nodes 2, 7. After this process terminates, the entire accessed subtree needs to be remapped, re-encrypted, and stored in the ORAM.

To see how this is achieved, recall our description of the oblivious map from [46] instantiated with an AVL tree T, from Section 2. All accesses perform a tree traversal beginning from the root and entail $O(\log N)$ node retrievals from T each of which is performed via a non-recursive Path-ORAM structure. Assuming n_w such accesses, they all have the same first step: retrieving the root node of T. Then, the client can decrypt the root and retrieve the ORAM positions of the two children. In case of multiple queries, it is likely that he needs both of them. Therefore, for that level of the tree he makes two calls to ORAM. In general, for the i-th layer of T, the client may need to perform up to n_w ORAM accesses. Since the ORAM is non-recursive, these n_w ORAM accesses can be performed in batch. Following this process, since T has $O(\log N)$ levels, the total number of ORAM accesses are $O(n_w \log N)$ and they are performed in batch for each layer. Thus, the total rounds of interaction is $O(\log N)$. After this step, the client performs the remapping process for all $O(n_w \log N)$ retrieved nodes, and updates the ORAM structure, in the same way as described in Section 2 for the case of a single access. To ensure no leakage beyond the number n_w, we take two additional measures. For the i-th layer of T the client performs $min\{n_w, 2^i\}$ accesses, independently of how many nodes from the next layer he actually needs (the remaining are dummy queries). Also, this process is always extended to the maximal possible height of an AVL tree of N elements ($\approx 1.45 \log N$).

Efficiency of Orion. The asymptotic performance of Orion is affected by that of the oblivious map of [46]. OMAP accesses take $O(\log^2 N)$ steps for a tree of N entries (here N is the total number of updates executed in the database) and require $O(\log N)$ rounds. For updates, since a constant number of such accesses is made, the above quantities give us the complexity of Orion. Regarding searches, recall that they entail n_w accesses executed in parallel and from our above analysis of the algorithm this can be done in $O(n_w \log^2 N)$ time and with $O(\log N)$ interactions. The communication complexity of updates algorithms is $O(\log^2 N)$ since each oblivious map access requires retrieving $O(\log N)$ Path-ORAM paths, each of length $\log N$. Likewise, the communication complexity of the search algorithm is $O(n_w \log^2 N)$.

Client storage. For simplicity of presentation, we assumed that the client stores **UpdtCnt** and **LastInd** locally. This implies a local storage of $O(|W| \log |D|)$ since each entry can be at most $\log |D|$ bits. This will typically be a few MB but in case this is unwanted, we can exploit the already existing oblivious maps to delegate this storage to the server. For example, we can store in $OMAP_{upd}$ special mappings $(w, 0) \rightarrow ($**UpdtCnt**$[w],$ **LastInd**$[w])$. Whenever the client needs to access these two maps, he can instead do it via an oblivious access to $OMAP_{upd}$. This would increase the number of roundtrips entailed and communication size, but it does not affect the asymptotic complexity of the scheme. Also, the oblivious map requires a stash of $O(\log^2 N)$ which is normally stored at the client. Instead, we can outsource it to the server and download it with every operation. With this two modifications the client storage of Orion becomes $O(1)$.

Server storage. On the server side, we stress that while the size of the two trees T, T' grows with each update operation (after N of them), their size becomes $O(N)$, the size of the corresponding Path-ORAM structures must already be bound to an upper bound of total updates. E.g., if the client wants the SSE to support up to 1 billion updates, he initializes each Path-ORAM with a tree structure of 32 levels. This means that the server will have to commit this much space initially. We do not believe this to be a serious limiting factor of our scheme due to the relative low cost and high availability of storage mediums. However, in settings where this may be an issue, there is an alternative that can be achieved via a simple amortization trick. The client initializes the Path-ORAM structures with trees of 1 layer. Afterwards, every 2^i updates for $i = 1, 2, \ldots$ he retrieves both oblivious map from the server and sets them up from scratch using Path-ORAM structures of $i + 1$ layers, i.e., with capacity 2^{i+1}, and fresh encryption keys. Since setting up an oblivious map with capacity 2^{i+1} takes $O(2^{i+1})$ time and this re-build operation takes place once every 2^i updates, the amortized cost per update is $O(1)$. Thus, the asymptotic behavior of Orion is not affected in this way; the same asymptotics hold but in an amortized manner.

Security analysis. The security analysis of ORION is rather straightforward due to the black-box use of oblivious maps. Throughout the execution of the protocol, the server only observes a sequence of Path-ORAM positions each chosen uniformly at random. Due to the padding with dummy queries discussed above, during updates the server sees a fixed number of such positions (depending on the type of update) which implies forward-privacy. During searches, the number of such locations that the server observes only reveals n_w. In particular, since after every update the accessed ORAM entries are remapped to freshly chosen random locations, it is impossible for the server to match the positions accessed during a search with specific update operations. This implies that ORION has minimal leakage, i.e., backward-privacy Type-I. We now state the following (full proof is provided in Appendix E).

THEOREM 4.2. *Assuming $OMAP_{src}$, $OMAP_{upd}$ are instantiated with the secure oblivious map of [46], ORION is an adaptively-secure SSE scheme with $\mathcal{L}^{Updt}(op, w, id) = op$ and $\mathcal{L}^{Srch}(w) = TimeDB(w)$.*

4.3 HORUS: More Efficient Searches

According to Definition 4.1, ORION has quasi-optimal search time $O(n_w \log^2 N)$ and non-trivial interaction. One downside, however, is that the search cost remains the same even when no deletions take place, i.e., it is actually $\Theta(n_w \log^2 N)$. Here, we present HORUS, a modified version of ORION, that achieves better asymptotic and concrete search time. Searches with HORUS take $O(n_w \log N \log d_w)$ in the worst case. Moreover, if no deletions related to w have taken place the search time becomes $O(n_w \log N)$. We describe here HORUS at a high-level (detailed description in Appendix F).

The main modification we make to ORION is that we replace $OMAP_{src}$ with simple non-recursive Path-ORAM structure which is again accessed by w and upd_{cnt}. In order to avoid having to store a position map of size N at the client, we instead generate it using a secure PRF, using a similar trick to [20]. However, this introduces a security issue. Since the PRF is a deterministic function, it will generate the same output when accessing the same (w, upd_{cnt}). This situation may arise multiple times through the protocol execution. E.g., assume the pairs $(w, 5), (w, 11), (w, 4)$ have been inserted in that chronological order. In this case, $(w, 5)$ corresponds to $upd_{cnt} = 1$, $(w, 11)$ to $upd_{cnt} = 2$, and $(w, 4)$ to $upd_{cnt} = 3$. To remove $(w, 11)$ afterwards, the client first retrieves its $upd_{cnt} = 2$ from $OMAP_{upd}$ and then evaluates the PRF on $(w, 2)$ to retrieve that ORAM path. This last operation violates forward-privacy as it trivially reveals this is the same ORAM location that was accessed during the addition process for $(w, 11)$.

HORUS avoids this by introducing a counter acc_{cnt} associated with each upd_{cnt} mapping to specific previous update location and it counts how many times this location has been accessed. In continuation of our previous example, the ORAM position for the three initial additions $(w, 5), (w, 11), (w, 4)$ for w would be computed as $PRF_K(w, 1, 1), PRF_K(w, 2, 1), PRF_K(w, 3, 1)$ where the first counter is upd_{cnt} and it indicates these are the three first updates for w, and the second counter is acc_{cnt}, indicating each of these upd_{cnt} values is used for the first time. These upd_{cnt}, acc_{cnt} are then stored in $OMAP_{upd}$ (whereas in ORION, $OMAP_{upd}$ stored only upd_{cnt}). When $(w, 11)$ is to be deleted, the client first retrieves the (upd_{cnt}, acc_{cnt}) pair for $(w, 11)$ from $OMAP_{upd}$. Then, he performs the same "swapping" trick as in ORION, inserting the entry corresponding to the previous latest update $(w, 4)$ to the OMAP position that corresponds to the upd_{cnt} of the removed entry but with acc_{cnt} incremented by one, i.e., $PRF_K(w, 2, 2)$. This ensures that, throughout the protocol execution, during update operations the same input is never consumed by the PRF more than once, which makes the distribution of ORAM positions observed by the server during updates indistinguishable from random.

However, this introduces an interesting challenge during search operations: How can the client know the correct acc_{cnt} value for all the n_w different values of upd_{cnt} counters of keyword w? One idea would be to retrieve them from $OMAP_{upd}$, by performing n_w accesses in batch, same as before. Clearly, this would require $\Theta(\log N)$ roundtrips, which is what we are trying to improve. Instead, the client engages into n_w *binary searches* executed in batch, where his goal is to identify the largest acc_{cnt} value that has been used for each upd_{cnt}, since these are the locations where the correct file identifiers for w are stored.

Concretely, to search for the n_w file identifiers associated with w, the client first evaluates $PRF_K(w, 1, 1), \ldots, PRF_K(w, n_w, 1)$ and sends the outputs indicating ORAM positions to the server, in batch. This step corresponds to "guessing" that the acc_{cnt} associated with each entry is equal to 1. After receiving the responses, he proceeds for the next binary search step by setting $acc_{cnt} = max_{acc}/2$, where max_{acc} is the maximum observed value of acc_{cnt} for keyword w, as established through the sequence of previous updates. For each upd_{cnt}, if the result is empty, (i.e., this acc_{cnt} has not been previously used) he tries again by moving "left" in the binary search, else by moving "right". The process ends when the largest used acc_{cnt} value has been found for all upd_{cnt}. The detailed procedure is somewhat more complicated to account for remapping the entries after each search, and it is described in Appendix F.

As we prove, the value max_{acc} is $O(d_w)$, i.e., it grows proportionally with the number of deletions for w, thus this process terminates after $O(\log d_w)$ rounds and entails $O(n_w \log d_w)$ ORAM accesses. Each such access returns a path of size $O(\log N)$ hence the search time and communication complexity is $O(n_w \log d_w \log N)$. In particular, note that if no deletions for w took place, $acc_{cnt} = 1$ for all entries and the process terminates after a single round. Retrieved entries are then re-mapped using a search counter that is incremented after every search, to hide future accesses.

On the downside, some of the ORAM positions accessed during searches were the same ones previously accessed during updates which introduces some leakage. In particular, simulating this transcript requires explicit knowledge of which addition is negated by each deletion (as this affects the acc_{cnt} of a particular location), which makes HORUS backward-private Type-III.

5 EXPERIMENTAL EVALUATION

We report on the performance of our proposed schemes and compare them with existing ones from [7]. We implemented MITRA, ORION, and HORUS in C++ using the Crypto++ [1] and OpenSSL [36] libraries for cryptographic operations. Specifically, we used AES-128/256 as the PRF. For the schemes of [7], we used the code released by the authors [6]. This includes DIANA, a scheme from [7] that

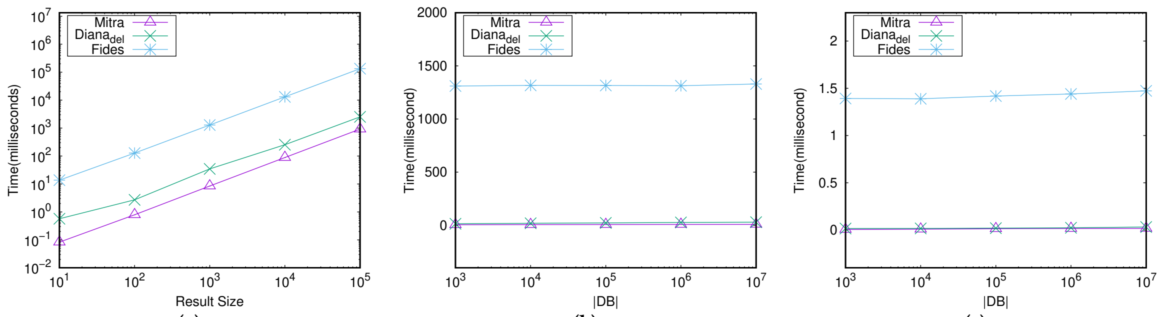


Figure 3: Computation time for: (a) search vs. variable result size for |DB| = 1M, (b) search vs. variable |DB| for result size 1K, (c) update vs. variable |DB|.

does not support deletions which we modified to implement Diana$_{del}$. The repository also includes Janus and the forward private scheme of [5] which we used as the back-end for building Fides, as described in [7]. All our implementations are publicly available [12].

For our experiments we used t2.xlarge AWS machines with four-core Intel Xeon E5-2676 v3 2.4GHz processor, running Ubuntu 14.04 LTS, with 16GB RAM, 100GB SSD (GP2) hard disk, and AES-NI enabled. All schemes were executed on a single machine while storing the database on RAM except for the WAN experiment which was run on two machines with 21ms roundtrip time (located in Germany and Ireland). We are interested in measuring executions time and communication size for search and update operations, as well as permanent client storage for the different schemes. We tested for variable database size $|\mathbf{DB}| = 10^3$–10^7 using synthetic records. For each database size, we set $|W|$ to one-hundredth of the database, and randomly generated the entries. Throughout the experiments, we considered variable result between 10–10^5 documents. Unless otherwise specified, after records were inserted we deleted at random 10% of the entries of the queried keyword to emulate the impact of deletions on performance. All experiments were repeated ×10 and the average is reported.

5.1 Performance comparison for Mitra

Our first set of experiments focuses on the performance of Mitra, and how it compares with Fides and Diana$_{del}$ from [7]. All three schemes, were implemented with the "cleanup" process enabled (that removes deleted entries after a search), as described in section 3 for Mitra and in [7] for the others.

Search. Figure 3 shows the search time when the result size (a) and the database size (b) change. As is clear from the graphs, the time growth is strongly linear for all schemes when the result size grows, whereas it remains visibly unaffected by changes in the database size. Throughout all the executions, Mitra is 145–253× faster than Fides, the only scheme with the same leakage type. This should come as no surprise as Fides relies on one-way trapdoor permutations (implemented with RSA) to achieve forward and backward privacy while Mitra relies on symmetric encryption. We believe the huge improvement in the search time is strong evidence for the practical performance of Mitra which has concretely very low overheads, e.g., for a database of 1M records and result size 100, it takes roughly 0.8ms for computation (most of which is spent at the client, with the server performing only lookups). To emphasize the performance of Mitra, observe that Diana$_{del}$ has worse search times, despite achieving only Type-III privacy!

Figure 4 shows the communication size when the result size (a) and the database size (b) change. Again, this grows linearly with result size and is unaffected by the database size. Comparing Fides and Mitra, we see that they have very similar communication sizes (the ratio of the former over the latter is 0.7–1.1). Despite the fact that with Fides the client only needs to initially send a single "token" (whereas with Mitra, he sends the entire TList), the search communication gap becomes smaller due to the server sending the encrypted indexes and the final cleanup process. In practice, both have low costs, e.g., Mitra transmits in total approximately $7KB$ for a database of $1M$ records and result size 100. Regarding the communication size for Diana$_{del}$, for the given configurations it is virtually the same as Mitra; for larger deletion percentages (not included above), its communication size becomes smaller (down to 3× smaller than that of Mitra).

Update. Figure 3(c) and Figure 4(c) show the update time and communication size respectively, for variable database sizes. The obvious conclusions from the figures are that: (i) the update time of the three schemes is almost unaffected by the size of the database, (ii) Fides is concretely much slower than the other two, due to performing an RSA private-key operation (instead of symmetric-key crypto), and (iii) all of them have excellent performance ($<$ 1.5ms for Fides, $\leq 32\mu s$ for the rest, and all of them $\leq$ 56bytes).

Client storage. All three schemes impose $|W| log|D|$ permanent storage at the client, in order to maintain the necessary update counter(s) for each keyword. The difference is that Mitra and Fides only requires one such counter per keyword, whereas Diana$_{del}$ requires two, as it stores insertions and deletions separately. Concretely, local storage for Mitra and Fides was roughly 156KB for 10K keywords (and twice that for Diana$_{del}$).

Comparison with Janus. In the above, we did not compare with Janus, the non-interactive scheme of [7], for two reasons. Firstly, it is only Type-III backward private. Secondly, its concrete performance is several times worse than these three schemes. For example, searching for a keyword with result size 1K in a database of 100K entries took 51 seconds assuming no previous deletions—this increases by roughly 7× with 10% deleted entries.

Experiments over WAN. In the above experiments, we executed the code for both the server and the client on the same machine and we reported computation times for searches and updates. To

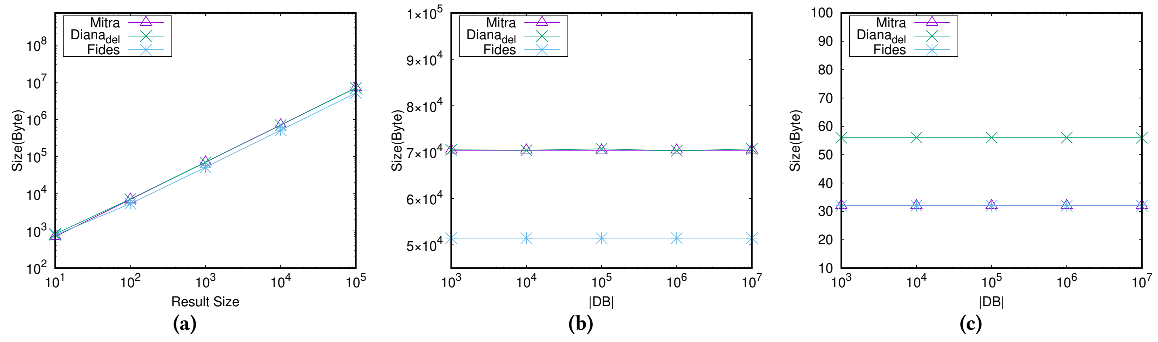


Figure 4: Communication size for: (a) search vs. variable result size for |DB| = 1M, (b) search vs. variable |DB| for result size 1K, (c) update vs. variable |DB|.

measure the end-to-end time for performing searches in a real-world setting, we ran Mitra and Fides in a WAN setting over two machines (server and client) with 21ms latency. Figure 5(a) shows the time breakdown for variable result sizes and $|\mathbf{DB}| = 1$M. Note that, in Fides the dominant overhead is due to computation whereas in Mitra due to communication (since $TList$ has to be transferred to the server). Despite this, in terms of total end-to-end time Mitra still outperforms Fides! According to our experiments, Mitra is $1.3 - 51\times$ faster than Fides depending on the result size.

As an insight for the practical performance of Mitra, consider the popular Enron email dataset which contains roughly 0.5M (keyword, email) pairs and 77K keywords. In this case, the performance of Mitra would be very similar to what we reported for $|\mathbf{DB}| = 1$M and 10K keywords. Over WAN with 21ms latency and result size 100, this would take 45ms for communication, 1.3ms for computation, and the client storage would be 156KB.

5.2 Performance of Orion & Horus

In order to measure the performance of Orion and Horus, we instantiated the oblivious map of [46] (using Path-ORAM with AES-256) and used it to implement the two schemes.

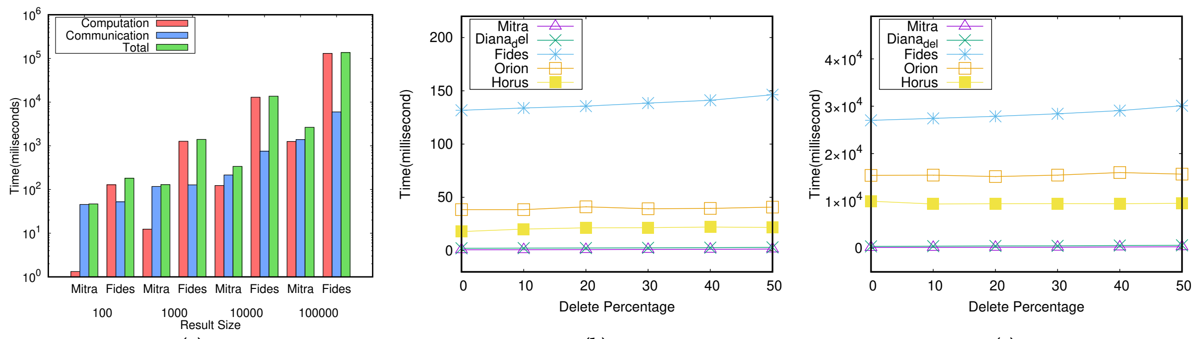
The search computation time for both of them is higher than that of Mitra while better than Fides. For example, for a database of size 100K and result size of 100, Orion took 38ms and Horus 17ms. For comparison, Fides (which has backward privacy Type-II) needs 131ms. Let us explain this in more detail. The search time of Fides grows strictly linearly with the result size, by performing a corresponding number of public key operations. On the other hand, the Orion client requests a number of ORAM paths from the server in order to "parse" the AVL tree. The number of paths grows with the result size but not necessarily linearly. E.g., assume the first two paths requested (corresponding to random positions) only differ at the last bit. In that case, the total number of ORAM buckets sent from the server would be just $\log N + 1$, instead of $2 \log N$. In general, the ORAM accesses requested from the client create a "subtree" of the Path-ORAM and only distinct buckets in this subtree need to be handled (and re-mapped).

Where Orion and Horus lack in performance is communication size. The sheer overhead of transmitting all the necessary ORAM blocks is concretely large. For a database of 1M and result size 100, Orion transmits 1.7MB and Horus 0.3MB—the same measurement for Fides is just 5.3KB. At the core of this issue is the "large number of buckets" problem when using ORAM for SSE. Given the very small size of the response to an SSE query (excluding actual files, in case they need to be retrieved) implies a very large overhead when encrypting them with ORAM due to the retrieval of multiple buckets per query. A very insightful discussion of the issue can be found in [34]. Simply put, in certain cases the client would be better off just encrypting the entire **DB** index with regular encryption and downloading it for every query. Still, recent research proposals [24, 39] explore the combination of oblivious structures with trusted hardware (which in our case would eliminate the need to transmit over the network), which opens the possibility of testing our schemes in that setting.

The effect of deletions. Despite the limitation identified above, one interesting result that we can deduce from our implementation is the effect of large volumes of deletions on the performance of different schemes. Recall that, schemes such as Orion and Horus that have quasi-optimal time should be essentially unaffected by the number of past deletions for a given result size—indeed that was our main motivation for exploring this direction. Figures 5(b)(c) show the required search computation time for all schemes for a database of size 100K and two cases: *(left)* "small" result size 100, and *(right)* "large" result size $20K$. In both cases, we vary the percentage of previous deletions between 0–50% while the result size remains fixed. (E.g., for 10% deletions with result size 100, we insert 111 records and delete 11 of them.) As shown in the figures, the overhead of Fides grows as the deletion percentage increases (the same is true for Mitra and Diana$_{del}$ but the effect is not readily visible due to scale). With Orion and Horus on the other hand, as long as the result size is fixed the number of previous deletions does not affect search times, which remain roughly constant.

6 CONCLUSION AND OPEN PROBLEMS

In this work, we introduced three new backward and forward private SSE schemes. Our constructions improve the state-of-the art ones in several aspects. Mitra is concretely the fastest existing such scheme, greatly outperforming existing ones, even those with higher leakage. Our two other constructions, Orion and Horus, are the first ones to have search time quasi-linear to n_w, i.e., the number of documents that contain keyword w *currently* in the database. All previous works that achieve backward-privacy have search time that is $\Omega(a_w)$, that is, at least linear in the total number of updates (including deletions) related to w; in practice a_w can be arbitrarily



(a) **(b)** **(c)**

Figure 5: (a) Computation, communication, and end-to-end time of WAN with 21ms latency for |DB| =1M and variable result size. Computation time for search vs.% of deletions for |DB| =100K and result size 100 (b) and 20K (c).

larger than n_w. ORION and HORUS both achieve this property but they offer different trade-offs between leakage and search performance. Our work leaves many open problems, such as investigating whether we can develop a scheme with quasi-linear search time and non-trivial communication without relying on ORAM (known to be possible for schemes that are only forward private). Another direction would be to devise a scheme with optimal search time (which seems hard for deletion-supporting constructions), or a non-interactive one with quasi-optimal search time. Finally, it would be interesting to revisit the backward privacy definitions of [7] and "evaluate" their leakage for real-world applications, e.g., in the light of possible deletion-specific attacks.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback. Research supported in part by HKUST award IGN16EG06, the National Science Foundation (NSF) through awards #1514261 and #1652259, and the National Institute for Standards and Technology.

REFERENCES

[1] Crypto++ Library 7.0. 2018. https://www.cryptopp.com/.
[2] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. 2016. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *STOC 2016*. 1101–1114.
[3] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *ACM CCS 2012*. 784–796.
[4] Dan Boneh and Brent Waters. 2013. Constrained Pseudorandom Functions and Their Applications. In *ASIACRYPT 2013*. 280–300.
[5] Raphael Bost. 2016. Σοφος: Forward Secure Searchable Encryption. In *ACM CCS 2016*. 1143–1154.
[6] Raphel Bost. 2017. OpenSSE Schemes. https://github.com/OpenSSE/opensse-schemes.
[7] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS 2017*. 1465–1482.
[8] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. 2014. Functional Signatures and Pseudorandom Functions. In *PKC 2014*. 501–519.
[9] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation.. In *NDSS*, Vol. 14. 23–26.
[10] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*. Springer, Berlin, Heidelberg, 353–373.
[11] David Cash and Stefano Tessaro. 2014. The locality of searchable symmetric encryption. In *EUROCRYPT 2014*. Springer, Berlin, Heidelberg, 351–368.
[12] Javad Ghareh Chamani. 2018. Implementation of Mitra, Orion, Horus, Fides, and DianaDel. https://github.com/jgharehchamani/SSE.
[13] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS 2005*. 442–455.
[14] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT 2010*. 577–594.
[15] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2016*. 79–88.
[16] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD 2016*. 185–198.
[17] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast Searchable Encryption With Tunable Locality. In *SIGMOD 2017, Chicago*. 1053–1067.
[18] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *TCC 2016-A*. 145–174.
[19] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient Dynamic Searchable Encryption with Forward Privacy. *PoPETs* 2018, 1 (2018), 5–20.
[20] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO 2016*. 563–592.
[21] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC 2009*. 169–178.
[22] Matthew D. Green and Ian Miers. 2015. Forward Secure Asynchronous Messaging from Puncturable Encryption. In *IEEE SP 2015*. 305–320.
[23] Florian Hahn and Florian Kerschbaum. 2014. Searchable encryption with secure and efficient updates. In *ACM CCS 2014*. 310–320.
[24] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. 2018. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *IACR Cryptology ePrint Archive* 2018 (2018), 247. http://eprint.iacr.org/2018/247
[25] Seny Kamara and Tarik Moataz. 2016. SQL on Structurally-Encrypted Databases. *IACR Cryptology ePrint Archive* 2016 (2016), 453. http://eprint.iacr.org/2016/453
[26] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *EUROCRYPT 2017*. 94–124.
[27] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and Dynamic Searchable Symmetric Encryption. In *FC 2013*. 258–274.
[28] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *ACM CCS 2012*. 965–976.
[29] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. 2013. Delegatable pseudorandom functions and applications. In *ACM CCS 2013*. 669–684.
[30] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates. In *ACM CCS 2017*. 1449–1463.
[31] Zheli Liu, Siyi Lv, Yu Wei, Jin Li, Joseph K. Liu, and Yang Xiang. 2017. FFSSE: Flexible Forward Secure Searchable Encryption with Efficient Performance. *IACR Cryptology ePrint Archive* 2017 (2017), 1105. http://eprint.iacr.org/2017/1105
[32] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *ACM CCS 2015*. 504–517.
[33] Ian Miers and Payman Mohassel. 2016. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. *IACR Cryptology ePrint Archive* 2016 (2016), 830.
[34] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptology ePrint Archive* 2015 (2015), 668. http://eprint.iacr.org/2015/668

[35] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. 2014. Dynamic searchable encryption via blind storage. In *IEEE SP 2014*. 639–654.
[36] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. 2003. https://www.openssl.org/.
[37] Cédric Van Rompay, Refik Molva, and Melek Önen. 2015. Multi-user Searchable Encryption in the Cloud. In *ISC 2015*. 299–316.
[38] Cédric Van Rompay, Refik Molva, and Melek Önen. 2017. A Leakage-Abuse Attack Against Multi-User Searchable Encryption. *PoPETs* 2017, 3 (2017), 168.
[39] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2017. ZeroTrace : Oblivious Memory Primitives from Intel SGX. *IACR Cryptology ePrint Archive* 2017 (2017), 549. http://eprint.iacr.org/2017/549
[40] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *ASIACRYPT 2011*. 197–214.
[41] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *IEEE SP 2000*. 44–55.
[42] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. 2018. Forward Private Searchable Symmetric Encryption with Optimized I/O Efficiency. *IEEE Transactions on Dependable and Secure Computing* (2018).
[43] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 14. 23–26.
[44] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013*. 299–310.
[45] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS 2015*. 850–861.
[46] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *ACM CCS 2014*. 215–226.
[47] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security 2016*. 707–720.

A GAME DEFINITIONS FOR SSE SECURITY

Figure 6 shows the execution of the $\mathsf{Real}^{\mathsf{SSE}}$ and $\mathsf{Ideal}^{\mathsf{SSE}}$ games for the SSE security definition 2.1.

```
$b \leftarrow \mathsf{Real}^{\mathsf{SSE}}_{\mathsf{Adv}}(\lambda, q)$:
1: $N \leftarrow \mathsf{Adv}(1^\lambda)$; $(K, \sigma_0, EDB_0) \leftarrow Initialize(1^\lambda, N)$;
2: for $k = 1$ to $q$ do
3:   $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$;
4:   if $type_k = search$ then
5:     $(\sigma_k, DB(w_k); EDB_k) \leftarrow Search(K, w_k, \sigma_{k-1}; EDB_{k-1})$
6:   else if $type_k = update$ then
7:     $(\sigma_k; EDB_k) \leftarrow Update(K, add/del, (id_k, w_k), \sigma_{k-1}; EDB_{k-1})$
8:   end if
9:   Let $t_k$ be the messages from client to server in the Access protocol above;
10: end for
11: $b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
12: return $b$;

$b \leftarrow \mathsf{Ideal}^{\mathsf{SSE}}_{\mathsf{Adv},Sim,\mathcal{L}}(\lambda, q)$:
1: $N \leftarrow \mathsf{Adv}(1^\lambda)$; $(st_{\mathcal{S}}, EDB_0) \leftarrow SimInit(1^\lambda, N)$;
2: for $k = 1$ to $q$ do
3:   $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$;
4:   if $type_k = search$ then
5:     $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow SimSearch(st_{\mathcal{S}}, \mathcal{L}^{Srch}(w_k); EDB_{k-1})$
6:   else if $type_k = update$ then
7:     $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow SimUpdate(st_{\mathcal{S}}, \mathcal{L}^{Updt}(w_k); EDB_{k-1})$
8:   end if
9: end for
10: $b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$;
11: return $b$;
```

Figure 6: Real and ideal experiments for the SSE scheme.

B TREE-BASED ORAM

Path-ORAM is an oblivious RAM introduced by Stefanov et al. [44]. It falls within the tree-based framework [40], and has since significantly improved [18, 45]. We describe here the construction as it is a building block for our schemes. We offer a high-level abstraction and refer interested readers to [44] for a more detailed description. In particular, we describe the non-recursive Path-ORAM for the case where the client stores the position map M. The tree-based ORAM's API can be abstracted as follows:

- $(\sigma; B) \leftarrow Initialize(1^\lambda, N)$: Given a security parameter λ and memory size N as input, it initializes a binary tree B with N leafs. Every node of B stores encryption of a dummy memory block with a null indicator to show that it has not been yet written. A position map M of size N is initialized. All nodes of the tree are encrypted using a semantically-secure encryption scheme, under key sk chosen by the client. Finally, a stash data structure S is initialized. The encrypted tree B is sent to the server, while $\sigma = (M, S, sk)$ are stored locally. In the following, $B(i)$ denotes the contents of the entire path from the i-th leaf of B to the root.
- $(\sigma, B(M[y]); B) \leftarrow Access(read, y, null, M, S, sk; B)$: To retrieve value with index y, the client searches stash S and if it is not found there it sends to the server the entry $M[y]$. The latter retrieves the path $B(M[y])$ and sends back all the encrypted nodes to the client who then decrypts them and extracts the entry that matches index y. He chooses a new random leaf from B for y and stores it at $M[y]$, He then repositions the retrieved nodes from along the path (freshly re-encrypted), together with the entries in S, in a way that "pushes" entries as deep as possible from root to leaf depending on their mapped positions in M. Any overflowing entries are stored in S. The new encrypted path is stored at the server who updates B.
- $(\sigma, B(M[y]); B) \leftarrow Access(write, y, val, M, S, sk; B)$: This is the process executed to set the entry for index y to val. It generally follows the same steps as *Read* but once the entry is identified it is updated to (y, val) before it is re-encrypted.

A note on our notation. Throughout the paper, we adopt the simplified notation $val \leftarrow Read(y, M[y])$ to abstract away the internals of the read operation. Let r denote the freshly chosen random position for y that will be stored to $M[y]$. Likewise, we adopt the notation $B(r) \leftarrow Write(y, val, r)$, to abstract away the internals of write operation. As described in Section 2, when using a non-recursive Path-ORAM to instantiate the oblivious map [46], there is no need to store M at the client. Also, the stash S can be dynamically downloaded with every access to reduce local storage. On the other hand, when using the above ORAM in Horus, we generate the values of M using a PRF function therefore again there is no need to store M.

Simulation of Path-ORAM accesses. As analyzed in [44], the transcript produced during the execution of a sequence of Path-ORAM accesses has a very simple structure. The messages for i-th access constitute $t_i = (M[y_i], B(M[y_i]))$, where $M[y_i]$ is chosen uniformly at random, and $B(M[y_i])$ contains contents of path, freshly re-encrypted with a semantically-secure scheme. This is indistinguishable from a transcript where the encrypted path is replaced with encryptions of all 0's. This, in turn, implies that the entire sequence of transcripts $t_1, \dots, t_q$ can be generated in a stateless manner, i.e., t_i can be simulated independently of t_{i-1}. Moreover, the simulation can run adaptively, without need to know q ahead of time. These two properties of Path-ORAM simulation are very important for proving the security of Orion and Horus.

C OBLIVIOUS MAP ROUTINES

OMAP offers the following API (see [46] for a detailed description).

- $(T, rootID) \leftarrow Setup(1^\lambda, N)$: Given a security parameter λ, and an upper bound N on the number of elements, it runs $T \leftarrow ORAM.Initialize(1^\lambda, N)$. It then creates an empty node as root of the AVL tree, stores it in T at randomly selected position $rootID$. The client sends T to the server and maintains locally the state output from the ORAM and $rootID$.
- $(rootID, data) \leftarrow Find(key, rootID)$: Given the search key key and the $rootID$, returns the corresponding value $data$. This is done by an interactive protocol that first fetches the tree *root*, compares the *id* with the retrieved one from the *root*, and subsequently retrieves either the left or the right child. This process repeats iteratively until the node for key has been found (or it is deduced it does not exist). Each node retrieval is done by running $ORAM.Access$, using the appropriate leaf position extracted from the parent node. At the end of this process, all the accessed nodes are re-encrypted and re-mapped to new random positions in T, the information is updated in the corresponding nodes and the entire accessed path is sent to the server. The client updates $rootID$ to the newly chosen position.
- $rootID \leftarrow Insert(key, val, rootID)$: Given a key-value pair key, val and $rootID$, it inserts this entry in the map. The process is similar to $Find$ with the important difference that a new node will be created for the entry. To maintain the schematics of the AVL tree, the process executes a *Balance* sub-protocol that re-balances the AVL tree by a sequence of ORAM read/writes. The server receives the updated tree T and the client the new position for the root $rootID$.

D PROOF OF THEOREM 3.1

Recall that in Mitra, the transcript the server observes consists of (initially empty) EDB, pairs $(addr, val)$ of entries to be added during updates, and address lists TList for entries to be retrieved during searches. The total length of the transcript is q. We now describe our simulator Sim. For setup, Sim simply outputs an empty map EDB and initializes an empty list I. During an update query, Sim computes $addr, val$ by sampling uniformly at random from the range of G. Let i be the timestamp of the update. Sim stores entry $I(j) = (addr, val)$. For all timestamps j that do not correspond to an update the entry $I(j)$ are null. During a search, Sim receives leakage functions $\mathbf{TimeDB}(w)$ and $\mathbf{Updates}(w)$. He then infers from $\mathbf{Updates}(w)$ the timestamps of previous updates related to the searched keyword, denoted by $J = (j_1, \dots, j_{a_w})$ and sends to server the addresses stored in $I(j_i)$ for $j = 1, \dots, a_w$. After receiving a result, it infers from $\mathbf{TimeDB}(w)$ the sets of documents that currently contain the searched keyword and sends it to the server.

We now prove the security of Mitra using Sim as follows.

Game-0 This is the Real^{SSE} game as defined in Appendix A.

Game-1 This is the same as **Game**-0, except that the values $G_K(w, \mathbf{FileCnt}[w]||b)$ for $b = 0, 1$ computed during an update are replaced by random values sampled uniformly at random from the range of G. A list I with q entries is maintained. If the i−th operation is an update the entry $I(j) = (addr, val, w, id)$ for $j = 1, \dots, q$ is storing the random values sampled together with the operation input (w, id), otherwise it stores null. During a search for keyword w, the game performs a scan of I to identify the entries that match w, sends to the server the corresponding $addr$ values, and receives a result. It then scans I again to deduce R_w the set of documents that currently hold w and sends R_w to the server. Since in **Game**-0 the PRF is never evaluated on the same input during updates, **Game**-1 is indistinguishable from **Game**-0 due to the security of the PRF.

Game-2 This is Ideal^{SSE} game as defined in Appendix A using simulator Sim described above. Clearly, the produced transcript follows the same distribution as the one produced during **Game**-1 since the leakage functions correspond to the same values that would be computed in that game, and the result of $id||op \oplus r$, where r is chosen uniformly at random is also distributed uniformly at random.

Regarding the correctness of Mitra, we note that unless G is a pseudorandom permutation (as is the case with our implementation), collisions may occur when computing $addr, val$ for different w, id pairs. This probability can be made arbitrarily small by increasing the range of G (see [19] for a simple trick to avoid this in practice).

E PROOF OF THEOREM 4.2

The transcript $(t_1, \dots, t_q)$ observed by the server during the Real^{SSE} game for Orion consists entirely of Path-ORAM positions and encrypted paths. Specifically, for the i-th operation, if this is an addition t_i consists of the necessary ORAM positions to retrieve one node from the AVL tree and two more to insert the new entries (and re-balance the tree). Due to padding, this always results in $\lceil 3 \cdot 3 \cdot 1.45 \log N \rceil$ ORAM positions and corresponding encrypted paths, assuming N updates. For a deletion, this number is $\lceil 5 \cdot 3 \cdot 1.45 \log N \rceil$ locations and paths.

If the i-th operation is a search with result size n_w, and assuming N updates have taken place, t_i consists of sets $t_{i,j}$ for $j = 1, \dots, \lceil 1.45 \log N \rceil$ each of which consists of a number of ORAM positions and encrypted paths. In particular, $|t_{i,j}| = min\{2^j, n_w\}$, that is, the number of ORAM positions and encrypted paths that consist each such set, and in turn, the structure of t_i can be determined just by knowing n_w. We now describe our simulator Sim:

- $SimInit(1^\lambda, N)$. It creates two trees T, T' with N leafs and populates each node with an encryptions of all 0's. It then sets $EDB_0 \leftarrow (T, T')$, sends it to the server, and sets $st_{\mathcal{S}}$ to null.
- $SimUpdate(st_{\mathcal{S}}, \mathcal{L}^{Updt}(w_k), EDB_{k-1})$. Recall that $\mathcal{L}^{Updt}(w_k) = op_k$. If $op_k = add$, then let $d = \lceil 3 \cdot 3 \cdot 1.45 \log N \rceil$, else $d = \lceil 5 \cdot 3 \cdot 1.45 \log N \rceil$. Sim generates d random positions and sends them to the server one after the other (receiving encrypted paths from EDB_{k-1} in-between). Finally, let P be the set of paths from T that correspond to the random positions generated for T, and

Algorithm 7 Horus $Setup(\lambda, N)$

```
1: $K \leftarrow Gen(1^\lambda)$
2: SrcCnt, UpdCnt, LastInd, Access, ReadyCnt$\leftarrow$ empty map
3: $T \leftarrow ORAM_{src}.Initialize(1^\lambda, A)$
                              $\triangleright$ $A$ is an empty memory array of size $N$
4: $(T', rootID) \leftarrow OMAP_{upd}.Setup(1^\lambda)$
5: $\sigma \leftarrow (rootID,$ SrcCnt, UpdCnt, LastInd, Access, ReadyCnt$)$
6: $EDB \leftarrow (T, T')$
7: Send $EDB$ to the server
```

P' be the corresponding set of paths from T'. Sim computes fresh encryptions of all 0's for these paths, and sends them to the server.

- $SimSearch(st_{\mathcal{S}}, \mathcal{L}^{Srch}(w_k), EDB_{k-1})$. Recall that $\mathcal{L}^{Srch}(w_k) =$ **TimeDB**(w). Sim Retrieves from this the number of entries currently in the database n_w. Then, for $j = 1, \ldots, \lceil 1.45 \log N \rceil$ it generates $min\{2^j, n_w\}$ random ORAM positions, which we denote by $t_{k,j}$. Sim sends them to the server $t_{k,1}, \ldots, t_{k,\lceil 1.45 \log N \rceil}$ one batch after the other (receiving encrypted paths from EDB_{k-1} in-between). Afterwards, let P be the set of paths from T that correspond to the random positions generated for T, and likewise P' for T'. Sim computes fresh encryptions of all 0's for these and sends them to the server. Finally, let R_W be the set of document identifiers corresponding to the searched keyword, as deduced from **TimeDB**(w). Sim sends R_w to the server.

Consider now the Ideal^{SSE} game with our simulator Sim. By our earlier analysis, the produced transcript is indistinguishable from the one produced during the Real^{SSE} game as the ORAM positions are generated in the same manner, the encryption scheme used for the Path-ORAM is semantically secure, an equal number of ORAM accesses is made, and the document identifiers are the same.

F HORUS CONSTRUCTION

Setup. The client generates five empty maps **SrcCnt**, **UpdCnt**, **LastInd**, **Access**, **ReadyCnt** that will be stored locally, as well as a PRF key K. Moreover, it initializes a non-recursive Path-ORAM $ORAM_{src}$ for an array of size N (where N is an upper bound on the size of the database), and an oblivious map $OMAP_{upd}$ and sends the resulting data structures to the server. The aim of $ORAM_{src}$ is to replace the oblivious map $OMAP_{src}$ in Orion. Inputs of the form $(w, upd_{cnt}, src_{cnt}, acc_{cnt})$ are mapped to one of the N leaves using the PRF G under key K. The counter upd_{cnt} is used to count update locations for w, as in Orion. Each counter acc_{cnt} corresponds to a specific location indicated by a upd_{cnt}, is initially set to 0, and incremented by 1 every time the location indicated by upd_{cnt} is accessed. Finally, src_{cnt} for keyword w counts how many times w has been searched, is initially 0 and incremented by 1 after every such search in order to map all relevant entries to new ORAM leafs.

The map $OMAP_{upd}$ works in a similar way as in Orion with one modification. It maps keys (b, w, x) to two types of values depending on the value of bit b. If $b = 0$, then x corresponds to a file identifier id and the result of the lookup will be a pair (upd_{cnt}, acc_{cnt}) or null. The entry for (w, id) can then be found in $ORAM_{src}$ at leaf $G_K(w, upd_{cnt}, src_{cnt}, acc_{cnt})$, where src_{cnt} is the current search count for w (already known by the client). If $b = 1$, then x corresponds to a counter upd_{cnt} and the result of the lookup is (id, acc_{cnt}), where acc_{cnt} corresponds to the number of times upd_{cnt} has been previously accessed, and id is the entry for w currently associated with location upd_{cnt}.

Update. For updates, we distinguish between the case of addition and deletion. For additions, the client first retrieves the correct pair (upd_{cnt}, acc_{cnt}) for the entry (w, id) to be added. He then ensures that this has not been added before or has been added and deleted (line 3). If the keyword is added for the first time or the location upd_{cnt} has not been accessed before, he initializes the values correctly (lines 4-12). He then retrieves from $OMAP_{upd}$ the correct acc_{cnt} for the next upd_{cnt} counter (which is where the new entry will be stored), increments it by one or initializes it, and updates the local storage (lines 13-22). Note that the id returned by the $OMAP_{upd}$ access is ignored at this point. He is then ready to store into $OMAP_{upd}$ the two entries necessary for this, one mapping (w, id) to the new (upd_{cnt}, acc_{cnt}) and one mapping w and the new upd_{cnt} to the new (id, acc_{cnt}) (lines 23-27). Then, he is ready to store the entry for (w, id) in the correct position in $ORAM_{src}$ and update the local state (lines 28-31). After storing the entry in ORAM, he ensures that the next search operation will be ready for execution, by setting (if necessary) a jump flag value at $acc_{cnt} = 1$ which shows the correct left boundary of the binary search for the current entry (lines 32-39).

For deletions, the client first retrieves the correct (upd_{cnt}, acc_{cnt}) counters for the entry (w, id) and proceeds only if the entry is in the database currently (line 3). He first "removes" the entry from $OMAP_{upd}$ by setting its upd_{cnt} to -1 and incrementing the acc_{cnt} for that location (lines 4-6). Then, the client decrements the **UpdCnt**$[w]$ value to correspond to the correct number of entries for w. If there are no remaining entries, he skips the rest of the process (line 8) and sets **LastId**$[w] = \perp$. Else, unless the entry that was deleted was the latest one for w (indicated by the fact that upd_{cnt} would be equal to the previous local entry of **UpdCnt**$[w]$), he needs to perform the "swapping" operation: the previous latest id needs to be written at the upd_{cnt} location of the deleted entry. This is done by writing an ORAM entry for $(w,$ **LastId**$(w))$ at location $G_K(w, upd_{cnt}, src_{cnt}, acc_{cnt} + 1)$ (lines 10-16). Moreover, corresponding entries are added at $ORAM_{upd}$ (lines 17-22).

The remaining part of the process is for updating the local state. Since the previous last entry has been moved, a new last entry must be prepared for future updates. This entails retrieving the (id, acc_{cnt}) pair for the decremented **UpdCnt**$[w]$ location (line 25), and updating the **LastInd**$[w]$ local entry (line 26).

In both cases, a necessary number or oblivious map and oblivious RAM dummy accesses is performed.

Search. The batch execution of ORAM read and write operations is described in Algorithms $BatchRead$ and $BatchWrite$. All the required information for these accesses is contained in $rInd$, $rposList$ for reads and $wInd$, $wposList$, $wval$ for writes.

Note that, according to the database manipulation during updates, the search query is expected to return **UpdCnt**$[w]$ identifiers corresponding to locations (w, upd_{cnt}) for $upd_{cnt} = 1, \ldots,$ **UpdCnt**$[w]$. The problem is that the client does not know the correct acc_{cnt} for each upd_{cnt}. However, he does know the maximum possible value for them is **Access**(w). Thus he can perform a binary search for each of them. The ORAM positions he searches for are

Algorithm 8 Horus $Update(K, add, (w, id), \sigma; EDB)$

```
1: $mapKey = (0, w, id)$
2: $(rootID, (upd_{cnt}, acc_{cnt})) \leftarrow OMAP_{upd}.Find(mapKey, rootID)$
3: if $upd_{cnt} = NULL$ or $upd_{cnt} = -1$ then
4:     if $\mathbf{UpdCnt}[w]$ is NULL then
5:         $\mathbf{UpdCnt}[w] = 0$
6:         $\mathbf{SrcCnt}[w] = 0$
7:         $\mathbf{ReadyCnt}[w] = 0$
8:     end if
9:     if $\mathbf{Access}[w]$ is NULL then
10:        $\mathbf{Access}[w] = 1$
11:    end if
12:    $\mathbf{UpdCnt}[w]++$
13:    $mapKey = (1, w, \mathbf{UpdCnt}[w])$
14:    $(rootID, (id, acc_{cnt})) = OMAP_{upd}.Find(mapKey, rootID)$
15:    if $acc_{cnt} = \perp$ then
16:        $acc_{cnt} = 1$
17:    else
18:        $acc_{cnt} = acc_{cnt} + 1$
19:        if $acc_{cnt} > \mathbf{Access}[w]$ then
20:            $\mathbf{Access}[w] = acc_{cnt}$
21:        end if
22:    end if
23:    $updVal = (\mathbf{UpdCnt}[w], acc_{cnt})$
24:    $data = ((0, w, id), updVal)$
25:    $rootID \leftarrow OMAP_{upd}.Insert(data, rootID)$
26:    $data = ((1, w, \mathbf{UpdCnt}[w]), (id, acc_{cnt}))$
27:    $rootID \leftarrow OMAP_{upd}.Insert(data, rootID)$
28:    $srcInd = w||\mathbf{UpdCnt}[w]||\mathbf{SrcCnt}[w]||acc_{cnt}$
29:    $newPos = G_K(w||\mathbf{UpdCnt}[w]||\mathbf{SrcCnt}[w]||acc_{cnt})$
30:    $T(newPos) \leftarrow ORAM_{src}.Write(srcInd, id, newPos)$
31:    $\mathbf{LastInd}[w] = id$
32:    if $\mathbf{ReadyCnt}[w] < \mathbf{UpdCnt}[w]$ then
33:        $\mathbf{ReadyCnt}[w]++$
34:        if $acc_{cnt} > 1$ then
35:            $srcInd = w||\mathbf{UpdCnt}[w]||\mathbf{SrcCnt}[w]||1$
36:            $newPos = G_K(w||\mathbf{UpdCnt}[w]||\mathbf{SrcCnt}[w]||1)$
37:            $T(newPos) \leftarrow ORAM_{src}.Write(srcInd,$
                   $\text{'@'}||acc_{cnt}, newPos)$
38:        end if
39:    end if
40: end if
41: Execute necessary dummy $OMAP_{upd}$ and $ORAM_{src}$ accesses
```

Algorithm 9 Horus $Update(K, del, (w, id), \sigma; EDB)$

```
1: $mapKey = (0, w, id)$
2: $(rootID, (upd_{cnt}, acc_{cnt})) \leftarrow OMAP_{upd}.Find(mapKey, rootID)$
3: if $upd_{cnt} > 0$ then
4:     $mapValue = (-1, acc_{cnt} + 1)$
5:     $data = (mapKey, mapValue)$
6:     $rootID \leftarrow OMAP_{upd}.Insert(data, rootID)$
7:     $\mathbf{UpdCnt}[w]$--
8:     if $\mathbf{UpdCnt}[w] > 0$ then
9:         if $\mathbf{UpdCnt}[w] + 1 \neq upd_{cnt}$ then
10:            $acc_{cnt}++$
11:            if $acc_{cnt} > \mathbf{Access}[w]$ then
12:                $\mathbf{Access}[w] = acc_{cnt}$
13:            end if
14:            $srcInd = w||upd_{cnt}||\mathbf{SrcCnt}[w]||acc_{cnt}$
15:            $newPos = G_K(w||upd_{cnt}||\mathbf{SrcCnt}[w]||acc_{cnt})$
16:            $T(newPos) \leftarrow ORAM_{src}.Write(srcInd,$
                   $\mathbf{LastInd}[w], newPos)$
17:            $mapKey = (0, w, \mathbf{LastInd}[w])$
18:            $mapVal = (upd_{cnt}, acc_{cnt})$
19:            $data = (mapKey, mapVal)$
20:            $rootID \leftarrow OMAP_{upd}.Insert(data, rootID)$
21:            $data = ((1, w, upd_{cnt}), (\mathbf{LastInd}[w], acc_{cnt}))$
22:            $rootID \leftarrow OMAP_{upd}.Insert(data, rootID)$
23:        end if
24:        $key = (1, w, \mathbf{UpdCnt}[w])$
25:        $(rootID, (id, acc_{cnt})) \leftarrow OMAP_{upd}.Find(key, rootID)$
26:        $\mathbf{LastInd}[w] = id$
27:    else
28:        $\mathbf{LastInd}[w] = \perp$
29:    end if
30: end if
31: Execute necessary dummy $OMAP_{upd}$ and $ORAM_{src}$ accesses
```

computed with the PRF G, initially testing for all $acc_{cnt} = 1$. The client may see two possible values in the position for $acc_{cnt} = 1$: If he receives an identifier, he sets the current value for the binary search, denoted by $curVal$, to the mean of the left and right boundaries (line 30 of Algorithm 10). Otherwise, he interprets the result as a jump flag value '@acc_{cnt}' (see below) which indicates the proper left boundary for the binary search for the current upd_{cnt}, and he updates $curVal$ accordingly (lines 26-28 of Algorithm 10). Then, he starts the binary search procedure in the batch mode (line 34 of Algorithm 10). After the binary search, the client remaps the retrieved entry to a new position using the PRF, by $\mathbf{SrcCnt}[w] + 1$, in order to disassociate the set of ORAM positions accessed during this search from future operations for w (line 9-12 of Algorithm 11). This repositioning may cause a problem in the next binary search, as acc_{cnt} for each $updt_{cnt}$ will not reset to 1 (as we do not want to update $OMAP_{upd}$). To avoid this, the client stores a jump flag value at $acc_{cnt} = 1$ if the current acc_{cnt} is greater than one (line 13-18 of Algorithm 11). Consequently, at the beginning of the next search operation for w the client will start the binary search from the value specified by the flag.

Another issue can be caused when an $updt_{cnt}$ position is deleted and after some search operations it is used again, due to intermediate insertions. E.g., suppose we have inserted four file identifiers for w, we execute a search ($src_{cnt} = 1$), and then remove one of them. During a subsequent search the src_{cnt} for the fourth upd_{cnt} position will not be increased. Assume that after a new search ($src_{cnt} = 2$) we insert another file for the same keyword. This will update the ORAM so that it contains new identifier with $acc_{cnt} = 2$ and $src_{cnt} = 2$ while there is not any entry for $acc_{cnt} = 1$ and $src_{cnt} = 2$, which prevents the next binary search from finding the correct acc_{cnt}. To solve this, we store the last number of $updt_{cnt}$s which were successfully used (and updated) by the last binary search in $\mathbf{ReadyCnt}$[w]. During insertions, the client compares

Algorithm 10 Horus $Search(K, w, \sigma; EDB)$

```
1:  $R = \{\}$
2:  $foundItem, left, right, curVal, lastID, lastAcc = []$
3:  $rInd, wInd, rposList, wposList, wval = []$
4:  for $i = 1$ to $\mathbf{UpdCnt}[w]$ do
5:      $left[i] = 1; foundItem[i] = false; curVal[i] = 1$
6:      $right[i] = \mathbf{Access}[w]$
7:  end for
8:  $foundAll = false$
9:  $firstRun = true$
10: while $foundAll = false$ do
11:     $foundAll = true$
12:     for $i = 1$ to $\mathbf{UpdCnt}[w]$ do
13:         if $foundItem[i] = false$ then
14:             $foundAll = false$
15:             $rInd[i] = w||i||\mathbf{SrcCnt}[w]||curVal[i]$
16:             $rposList[i] = G_K(w||i||\mathbf{SrcCnt}[w]||curVal[i])$
17:         end if
18:     end for
19:     if $foundAll = false$ then
20:         $ids \leftarrow ORAM_{src}.BatchRead(rInd, rposList)$
21:         for $i = 1$ to $\mathbf{UpdCnt}[w]$ do
22:             if $foundItem[i] = false$ then
23:                 $id \leftarrow ids[i]$
24:                 if $firstRun = true$ then
25:                     if $id[0] =$ '@' and $left[i] = 1$ then
26:                         $left[i] = extractValue(id)$
27:                         $curVal[i] = (left[i] + right[i])/2$
28:                         $lastAcc[i] = curVal[i]$
29:                     else
30:                         $curVal[i] = (left[i] + right[i])/2$
31:                     end if
32:                     Continue        ▷ Increment $i$ and restart loop
33:                 end if
34:                 BinarySearch$(id, i, \sigma)$
35:             end if
36:         end for
37:         $firstRun = false$
38:     end if
39: end while
40: $\mathbf{SrcCnt}[w]$++
41: $\mathbf{ReadyCnt}[w] = R.size$
42: $T(newPos) \leftarrow ORAM_{src}.BatchWrite(wInd, wval, wposList)$
43: return $R$
```

Algorithm 11 Horus $BinarySearch(id, i, \sigma; EDB)$

```
1:  if $id = \perp$ or $left[i] \geq right[i]$ then
2:      if $right[i] \leq left[i]$ then
3:          $foundItem[i] = true$
4:          if $lastID[i] = \perp$ then
5:              $lastID[i] = id$
6:              $lastAcc[i] = curVal[i]$
7:          end if
8:          $R = R \cup \{lastID[i]\}$
9:          $wInd.add(w||i||\mathbf{SrcCnt}[w] + 1||lastAcc[i])$
10:         $p = G_K(w||i||\mathbf{SrcCnt}[w] + 1||lastAcc[i])$
11:         $wposList.add(p)$
12:         $wval.add(lastID[i])$
13:         if $lastAcc[i] > 1$ then
14:             $wInd.add(w||i||\mathbf{SrcCnt}[w] + 1||1)$
15:             $p = G_K(w||i||\mathbf{SrcCnt}[w] + 1||1)$
16:             $wposList.add(p)$
17:             $wval.add('@'||lastAcc[i])$
18:         end if
19:     else
20:         if $curVal[i] = right[i]$ then
21:             $curVal[i] = \lfloor (left[i] + right[i])/2 \rfloor$
22:             $right[i] = curVal[i]$
23:         else
24:             $right[i] = curVal[i]$
25:             $curVal[i] = \lceil (left[i] + right[i])/2 \rceil$
26:         end if
27:     end if
28: else
29:     $lastID[i] = id$
30:     $lastAcc[i] = curVal[i]$
31:     if $curVal[i] = left[i]$ then
32:         $left[i] = curVal[i]$
33:         $curVal[i] = \lceil (left[i] + right[i])/2 \rceil$
34:     else
35:         $left[i] = curVal[i]$
36:         $curVal[i] = \lfloor (left[i] + right[i])/2 \rfloor$
37:     end if
38: end if
```

ReadyCnt[w] with **UpdtCnt**[w] to determine the necessity of a jump flag insertion at $acc_{cnt} = 1$ (if not set previously).

At the end of search procedure, the client increments **SrcCnt**[w], updates **ReadyCnt**[w], and executes batch write to store new encryptions on the server (lines 40-42 of Algorithm 10).

Efficiency of Horus. Following from the analysis of Orion and the fact that ORAM accesses for a non-recursive Path-ORAM take $O(\log N)$ time and communication, the update time and communication of Horus is again $O(\log^2 N)$. For searches, observe that the while-loop in line 10 of Algorithm 10 will run for $O(\log \mathbf{Access}[w])$ times from standard binary search analysis. Each loop entails two for-loops (lines 12 and 21 of Algorithm 10) that run for **UpdCnt**[w] steps, and one ORAM batch read (line 20 of Algorithm 10). Recall that $\mathbf{UpdCnt}[w] = n_w$, by definition of our construction. Moreover, our analysis in the extended version indicates that $\mathbf{Access}[w] \in O(d_w)$, i.e., it grows linearly with the number of deletions for w. Since each ORAM access takes $O(\log N)$ time, the overall search time complexity, as well as communication size, are $O(n_w \log d_w \log N)$. Assuming the $O(n_w)$ ORAM accesses in each repetition of the while-loop and the remapping ORAM accesses in line 42 of Algorithm 10 are executed in batch, the number of necessary roundtrips is $O(\log d_w)$. Finally, the client storage is $|W|log|D|$ for storing the keyword dictionaries. We cannot use the same trick as in Orion to store these in the OMAP, as that would impose $O(\log N)$ rounds of interaction for search.