TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption

Sanjam Garg[1], Payman Mohassel[2], and Charalampos Papamanthou[3]

[1] University of California, Berkeley
[2] Yahoo! Labs
[3] University of Maryland, College Park

Abstract. We present TWORAM, the first efficient round-optimal oblivious RAM (ORAM) scheme. TWORAM provides oblivious access of a memory index y in exactly two rounds: The client prepares an encrypted query encapsulating y and sends it to the server. The server accesses memory obliviously and returns encrypted information containing the desired value $\mathsf{M}[y]$. The cost of TWORAM is only a multiplicative factor of security parameter higher than the tree-based ORAM schemes such as the path ORAM of Stefanov et al. (CCS, 2013). TWORAM gives rise to interesting applications, and in particular to the first fully-secure searchable symmetric encryption scheme where search is sublinear and search pattern is not leaked—access pattern can also be concealed if we assume the documents are stored in the obliviously accessed memory M.

1 Introduction

Oblivious RAM (ORAM) is a cryptographic primitive for accessing a remote memory M of n entries in a way that memory accesses do not reveal anything about the accessed index $y \in \{1, \ldots, n\}$. Goldreich and Ostrovsky [13] were the first to show that ORAM can be built with $\mathsf{poly}(\log n)$ bandwidth overhead[4], and since then, there has been a fruitful line of research on substantially reducing this overhead [7,22,28,30], mainly motivated by the tree ORAM framework proposed by Shi et al. [25]. However, existing practical ORAM protocols are highly-interactive, requiring the client to perform a "download-decrypt-compute-encrypt-upload" operation several times (typically $O(\log n)$ rounds are involved). This can be a bottleneck for real-world applications where low latency is important.

In this paper we consider the problem of building an efficient round-optimal ORAM scheme. In particular, we propose TWORAM, an ORAM scheme enabling a client to obliviously access a memory location $\mathsf{M}[y]$ in two rounds, where the client sends an encrypted message to the server that encapsulates y, the server does oblivious computation, and finally sends a message back to the client, from which the client can retrieve the desired value $\mathsf{M}[y]$. In theory, this is already achievable using the recent work on garbled RAM by Lu and Ostrovsky [21] and its follow-up works [8,9,10]. In particular, by applying the garbled RAM constructions to simple RAM programs

[4] In this paper, *bandwidth overhead* is defined as the number of bits transferred between the client and the server during a single memory access.

for $\texttt{array\_read}(y)$ and $\texttt{array\_write}(y, val)$, one obtains round-optimal ORAM with only two rounds of interaction per access.

However, as we show in detail in the related work section, such approaches are impractical, leading to a bandwidth overhead that can go up to $O(\kappa^5 \log^5 n)$ [8], where κ is the security parameter and where the block size is $O(\log n)$. On the contrary, TWORAM's bandwidth overhead is always $O(\kappa \cdot p)$ where p is the bandwidth overhead of a tree ORAM scheme (e.g., for path ORAM [28], it is $p = \log^3 n / \log \log n$ for a block of size $O(\log n)$ bits). At the same time, TWORAM's protocols are very simple to describe and thus lend themselves to further optimizations and practical implementations.

1.1 Technical highlights and contributions

Our construction is inspired by the ideas from the recent, black-box garbled RAM work by Garg, Lu and Ostrovsky [8], applied to the specific case of tree ORAM algorithms [25]. In particular, our construction bypasses various inefficiencies involved in the original construction from [8], yielding much more practical protocols.

Our first step is to abstract away certain details of eviction-based tree ORAM algorithms, such as path ORAM [28], circuit ORAM [30] and Onion ORAM [7]. The non-recursive version of these algorithms works as follows: The memory M that must be accessed obliviously is stored as a sequence of L trees $T_1, T_2, \ldots, T_L$. The actual data of M are stored encrypted in the tree T_L, while the other trees store *position map* information (also encrypted). Only T_1 is stored on the client side. Roughly speaking, to access an index y in M, the client accesses T_1 and sends a path index p_2 to the server. The server then, successively accesses paths $p_2, p_3, \ldots, p_L$ in $T_2, T_3, \ldots, T_L$. However the paths are accessed adaptively: in order to learn p_i, one needs to first access p_{i-1} in T_{i-1}, and have all the information (also known as buckets) stored in its nodes decrypted. This is where existing approaches require interaction: decryption can only take place at the client side, which means all the information on the paths must be communicated back to the client.

TWORAM's core idea. In order to avoid the roundtrips described above, we do not use standard encryption. Instead, we *hardcode* the content of each bucket inside a *garbled circuit* [32]. In other words, after the trees $T_2, T_3, \ldots, T_L$ are produced, the client generates one garbled circuit per each internal node in each tree. The function of this garbled circuit is very simple: Informally, it takes as input an index x; loops through the blocks $A[i]$ contained in the current bucket until it finds $A[x]$, and returns the index $\pi = A[x]$ of the next path to be followed. Note that the index π is returned in form of a *garbled input* for the next garbled circuit, so that the execution can proceed by the server until T_L is reached, and the final desired value can be returned to the client (see Figure 3 for a more formal description).

This simplified description ignores some technical hurdles. Firstly, security of the underlying ORAM scheme requires that location where $A[x]$ is stored remain hidden. In particular, the garbled circuit which has the value $A[x]$ inside it must remain hidden. We resolve this issue as follows. For every bucket that the underlying ORAM needs to be touched, all the corresponding garbled circuits are executed in a specific order and

the value of interest is carried along the way and output only by the last circuit. We provide more details in our construction.

Secondly, the above approach only works well for a single memory access, since the garbled circuits cannot be reused and so we need to provide fresh ones in their place. Fortunately, as we show in the paper, only a logarithmic number of garbled circuits are touched for each access. These circuits can be downloaded by the client who decodes the hardcoded values, performs the eviction strategy locally (on plaintext data), and sends fresh garbled circuits back to the server. Note that if the client can maintain a small but permanent storage (as opposed a transient storage), this step does not increase the number of rounds from two to three, since sending the fresh garbled circuits to the server can be "piggybacked" onto the message the client prepares for the next memory access.

Finally, it is worth noting that eviction is the most computationally intensive operation of the underlying ORAM schemes. As pointed in [30], for typical secure computation techniques that internally use ORAM the size of the circuit performing the eviction turns out be the bottleneck. In contrast to this, in our construction only some parts of the underlying ORAM computation are performed using garbled circuits—specifically, ones that are essential for squishing rounds. This allows us to shift the load of eviction to the client, where it can be performed much more cheaply than on the server.

Finally, in order to ensure the desired efficiency we develop optimizations that help ensure that the sizes of the circuits garbled in our construction remain small.

Constant-round, fully-secure SSE. We apply TWORAM in Searchable Symmetric Encryption (SSE). An SSE scheme allows a client to outsource a database (which we define here as a set of document/keyword-set pair $\mathsf{DB} = (d_i, W_i)_{i=1}^{N}$) to an untrusted server in an encrypted format, where a search query for w should return all d_i where $w \in W_i$. A natural candidate for fully-secure SSE based on TWORAM is to store the database DB in a data structure such as a hash table with fast lookup based on w, and outsource the hash table memory using TWORAM. Now, each SSE search/add query can be implemented using a constant number of accesses in TWORAM. Intuitively, the security of TWORAM ensures that the SSE search pattern is not leaked to the server.

The main disadvantage of the above approach is that the number of TWORAM accesses for each keyword search is proportional to the number documents containing w (denoted by $\mathsf{DB}(w)$) which can be large. This would increase the round complexity by a multiplicative factor of $|\mathsf{DB}(w)|$ since the accesses in TWORAM cannot be parallelized.[5] In particular, each garbled circuit in the above-mentioned construction can only be used once and needs to be refreshed before the next access. This also means that, for each keyword search w, the bandwidth/computation overhead of TWORAM is multiplied by $|\mathsf{DB}(w)|$.

Our construction combines TWORAM and a non-recursive path ORAM (i.e., one whose position map of the first level is not outsourced) in a way that keyword searches only require a single access using TWORAM, and $|\mathsf{DB}(w)|$ parallel (and much more efficient) accesses to the non-recursive path ORAM. This yields a construction with 4 rounds of interaction. Furthermore, the size of the memory stored in TWORAM is only

[5] We note that it is also possible to extend our construction to enable multiple parallel accesses. However, this entails significant slowdown for our construction.

Table 1: Comparison with previous work. We use n to denote the number of memory entries and κ to denote the security parameter. From tree-based ORAMs, we compare with path ORAM [28], for which we have $p = O(\log^3 n)$ and $c = O(\log^2 n) \cdot \omega(1)$. For the comparison, the data block is taken to be $O(\log n)$ bits. Note that TWORAM adds a multiplicative security parameter to the path ORAM overhead, yet it reduces the rounds to three (with transient storage) and to two (with permanent storage).

approach	rounds per access	client storage (in bits)	bandwidth overhead per access (in bits)
hierarchical (e.g., [18])	$\log n$	$O(\log n)$ (transient)	$O(\log^3 n / \log\log n)$
tree-based (e.g., [28])	$\log n$	c (transient)	p
non-black-box GRAM [9,10,21]	2	$\mathsf{poly}(\log n, \kappa)$ (transient)	$\mathsf{poly}(\log n, \kappa, \lvert f \rvert)$
black-box GRAM [8]	2	$\geq \kappa^5 \cdot \log^5 n$ (transient)	$\geq \kappa^5 \cdot \log^5 n$
TWORAM	3	$c \cdot \kappa$ (transient)	$\kappa \cdot p$
TWORAM	2	$c \cdot \kappa$ (permanent)	$\kappa \cdot p$

proportional to total number of unique keywords which is significantly smaller than the number of keyword/document pairs. Finally, and most importantly, our construction *does not leak the search/access pattern*, by providing randomly generated tokens every time a search is performed.

The core idea is to store, using TWORAM, the memory for a hash table of pairs of the form $(w, (count_w, access_w))$, where w is a keyword, $count_w$ is the number of documents containing w and $access_w$ is the number of times w has been accessed so far. The keyword/document pairs $(w||i, d_i)$ (where d_i is the i-th document containing w) are then stored in a one-level path ORAM where their position in the path ORAM tree is determined on the fly and using a PRF F by computing $F_k(w||i, access_w)$, where k is a secret key held by the client. To search for a keyword w, we first access TWORAM to obtain $(count_w, access_w)$ (and increment $access_w$), and then generate all positions to look up in the path ORAM using the PRF F. These lookups can all be performed in parallel and updating the paths can be piggybacked to the next keyword search.

1.2 Related Work

Oblivious RAM. ORAM protocols in the literature can be categorized into *hierarchical* [14,15,18,20], motivated by the seminal work of Goldreich and Ostrovsky [13], and *tree-based* [7,22,28,30], motivated by the seminal work of Shi et al. [25]. *All these works* are interactive, requiring a polylogarithmic number of rounds for each data access. We note however, that, by picking the data block size to be very big (e.g., $\sqrt{n}$ bits), the number of rounds in tree-based ORAMs can be made constant, yet the bandwidth increases beyond polylogarithmic, so such a parameter selection is not interesting. In terms of improved round complexity, Williams and Sion [31] constructed a single-roundtrip ORAM scheme for accesses, yet their setup protocol involves a polylogarithmic number of rounds, as opposed to TWORAM, which can be setup with just a *single* message from the client to the server.

The only known approach to construct a round-optimal ORAM scheme is to use the recent work on garbled RAM [8,9,10,21] as a black box. However, such approaches

are impractical. For the non-black-box Garbled RAM approaches [21,10,9], the bandwidth overhead grows with $\mathsf{poly}(\log n, \kappa, |f|)$, where $|f|$ is the size of the circuit $|f|$ for computing the one-way function f. Hence this construction is only of theoretical interest. For the black-box Garbled RAM approach [8] the bandwidth overhead grows with $\mathsf{poly}(\log n, \kappa)$. The authors do not provide details on how large the involved polynomials are, which will depend on the choice of various parameters. According to our rough calculation, this value is worse than $\kappa^5 \cdot \log^5 n$. A detailed comparison of TWORAM and related work can be found in Table 1.

Searchable Encryption. Song et al. [26] presented the first SSE scheme showing the feasibility of performing searches on encrypted data. Since then, a large number of follow-up work have designed SSE for both static data [6,4,2] and dynamic data [3,11,17,16,29,27]. Unlike our construction, *all these approaches use deterministic tokens*, an therefore leak the search/access patterns.

The only approaches that have been proposed, that are constant-round and have randomized tokens (apart from constructing SSE through Garbled RAM) are the ones based on functional encryption [24]. However, such approaches incur linear search overhead.

2 Definitions and preliminaries

In this section, we recall definitions and describe building blocks we use in this paper. We use the notation $\langle C', S' \rangle \leftrightarrow \Pi \langle C, S \rangle$ to indicate that a protocol Π is executed between a client with input C and a server with input S. After the execution of the protocol the client receives C' and the server receives S'. For non-interactive protocols, we just use the left arrow notation ($\leftarrow$) instead.

2.1 Garbled Circuits

Garbled circuits were first constructed by Yao [32] (see Lindell and Pinkas [19] and Bellare et al. [1] for a detailed proof and further discussion). A circuit garbling scheme is a tuple of PPT algorithms $(\mathsf{GCircuit}, \mathsf{Eval})$, where $\mathsf{GCircuit}$ is the circuit garbling procedure and Eval the corresponding evaluation procedure. More formally:

- $(\tilde{C}, \mathsf{lab}) \leftrightarrow \mathsf{GCircuit}\,(1^\kappa, C)$: $\mathsf{GCircuit}$ takes as input a security parameter κ, and a circuit C. This procedure outputs a *garbled circuit* $\tilde{C}$ and lab, which is a set of input labels for each input wire of C.
- $y \leftrightarrow \mathsf{Eval}(\tilde{C}, \mathsf{lab}_x)$: Given a garbled circuit $\tilde{C}$ and a sequence of input labels lab_x (which we will call garbled inputs[6]), Eval outputs y.

Correctness. For correctness, we require that for any circuit C and an input x for C, we have that that $C(x) = \mathsf{Eval}(\tilde{C}, \mathsf{lab}_x)$, where $(\tilde{C}, \mathsf{lab}) \leftrightarrow \mathsf{GCircuit}\,(1^\kappa, C)$.

[6] Traditionally, a garbling scheme has encode algorithm the takes lab and x as input and outputs the garbled inputs for x. For simplicity we use lab_x to mean encoding input x.

Security. For security, we require that for any PPT adversary A, there is a PPT simulator Sim such that the following distributions are computationally indistinguishable:

- $\mathrm{Real}_{\mathsf{A}}(\kappa)$: A chooses a circuit C. The experiment runs $(\tilde{C}, \mathsf{lab}) \leftrightarrow \mathsf{GCircuit}\,(1^\kappa, C)$ and sends $\tilde{C}$ to A. A then outputs an input x. The experiment outputs $(\tilde{C}, \mathsf{lab}_x)$.
- $\mathrm{Ideal}_{\mathsf{A},\mathsf{Sim}}(\kappa)$: A chooses a circuit C. The experiment runs $(\tilde{C}, \sigma) \leftrightarrow \mathsf{Sim}(1^\kappa)$ and sends $\tilde{C}$ to A. A then outputs an input x. The experiment runs $\mathsf{lab}_x \leftrightarrow \mathsf{Sim}(1^\kappa, \sigma)$, and outputs $(\tilde{C}, \mathsf{lab}_x)$.

The above definition guarantees adaptive security, since the adversary gets to choose input x after seeing the garbled circuit $\tilde{C}$. We only know how to instantiate garbling schemes with adaptive security in the random oracle model. In the standard model, existing garbling schemes achieve a weaker static variant of the above definition where the adversary chooses both C and input x at the same time and before receiving $\tilde{C}$.

Concerning complexity, we note that if the cleartext circuit C has $|C|$ gates, the respective garbled circuit has size $O(|C|k)$. This is because every gate in the circuit is typically replaced with a table of four rows, each row storing encryptions of labels (each encryption has k bits).

2.2 Oblivious RAM

We recall *Oblivious RAM* (ORAM), a notion introduced and first studied by Goldreich and Ostrovsky [12,23]. ORAM can be thought of as a compiler that encodes the memory into a special format such that accesses on the compiled memory do not reveal the underlying access patterns on the original memory. An ORAM scheme consists of protocols $(\textsc{Setup}, \textsc{ObliviousAccess})$.

- $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$: $\textsc{Setup}$ takes as input the security parameter κ and a memory array M and outputs a secret state σ (for the client), and an encrypted memory EM (for the server).
- $\langle (\mathsf{M}[y], \sigma'), \mathsf{EM}' \rangle \leftrightarrow \textsc{ObliviousAccess}\langle (\sigma, y, v), \mathsf{EM} \rangle$: $\textsc{ObliviousAccess}$ is a protocol between the client and the server, where the client's input is the secret state σ, an index y and a value v which is set to null in case the access is a read operation (not a write). Server's input is the encrypted memory EM. Client's output is $\mathsf{M}[y]$ and an updated secret state σ' and the server's output is an updated encrypted memory EM' where $\mathsf{M}[y] = v$, if $v \neq \mathsf{null}$.

Correctness. Consider the following correctness experiment. Adversary A chooses memory M. Consider the encrypted database EM generated with $\textsc{Setup}$ (i.e., $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$). The adversary then adaptively chooses memory locations to read and write. Denote the adversary's read/write queries by $(y_1, v_1), \ldots, (y_q, v_q)$ where $v_i = \mathsf{null}$ for read operations. A wins in the correctness game if $\langle (\mathsf{M}_i[y_i], \sigma_i), \mathsf{EM}' \rangle$ are not the final outputs of the protocol $\textsc{ObliviousAccess}\langle (\sigma_{i-1}, y_i, v_i), \mathsf{EM}_{i-1} \rangle$ for any $1 \leq i \leq q$, where $\mathsf{M}_i, \mathsf{EM}_i, \sigma_i$ are the memory array, the encrypted memory array and the secret state, respectively, after the i-th access operation, and $\textsc{ObliviousAccess}$ is run between an honest client and server. The ORAM scheme is correct if the probability of A in winning the game is negligible in κ.

Semi-Honest Security. An ORAM Scheme is secure if for any adversary A, there exists a simulator Sim such that the following two distributions are computationally indistinguishable.

- $\mathrm{Real}_{\mathsf{A}}(\kappa)$: A chooses M. The experiment then runs $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$. A then adaptively makes read/write queries (y_i, v) where $v = \mathsf{null}$ on reads, for which the experiment runs the protocol

$$\langle (\mathsf{M}[y_i], \sigma_i), \mathsf{EM}_i \rangle \leftrightarrow \textsc{ObliviousAccess}\langle (\sigma_{i-1}, y_i, v), \mathsf{EM}_{i-1} \rangle \,.$$

  Denote the full transcript of the protocol by t_i. Eventually, the experiment outputs $(\mathsf{EM}, t_1, \ldots, t_q)$ where q is the total number of read/write queries.
- $\mathrm{Ideal}_{\mathsf{A},\mathsf{Sim}}(\kappa)$: The experiment outputs $(\mathsf{EM}, t'_1, \ldots, t'_q) \leftrightarrow \mathsf{Sim}(q, |\mathsf{M}|, 1^\kappa)$.

2.3 Hash Tables

A hash table is a data structure commonly used for mapping keys to values [5]. It often uses a hash function h that maps a key to an index (or a set of indices) in a memory array M where the value associated with the key may be found. In particular, h takes as input a keyword key and outputs a set of indices $i_1, \ldots, i_c$ for a fixed constant c. The value associated with key is in one of the locations $\mathsf{M}[i_1], \ldots \mathsf{M}[i_c]$. The keyword is not in the table if it is not in one of those locations. Similarly, to write a new $(key, value)$ pair into the table, $(key, value)$ is written into the first empty location among $i_1, \ldots, i_c$. More formally, we define a hash table $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$ using a tuple of algorithms and a parameter c denoting an upper bound on the number of locations to search.

- $(h, \mathsf{M}) \leftrightarrow \mathsf{hsetup}(S, size)$: hsetup takes as input an initial set S of keyword-value pairs and a maximum table size $size$ and outputs a hash function h and a memory array M.
- $value \leftrightarrow \mathsf{hlookup}(key)$: $\mathsf{hlookup}$, computes $i_1, \ldots, i_c \leftrightarrow h(key)$, looks for a key-value pair $(key, \cdot)$ in $\mathsf{M}[i_1], \ldots, \mathsf{M}[i_c]$. If such a pair is found it returns the second component of the pair (i.e., the value), else it returns $\perp$.
- $\mathsf{hwrite}(key, value)$: hwrite computes $i_1, \ldots, i_c \leftrightarrow h(key)$, if $(key, value)$ already exists in one of those indices in M it does nothing, else it stores $(key, value)$ in the first empty index.

2.4 Searchable Symmetric Encryption

A database D is a collection of documents d_i each of which consist of a set of keywords W_i. A document can be a webpage, an email, or a record in a database, and the keywords can represent the words in the document, or the attributes associated with it. A searchable symmetric encryption (SSE) scheme, introduced in the seminal work of Song et al. [26], allows a client to outsource a database to an untrusted server in an encrypted format, and have the server perform keyword searches that return a set of documents containing the keyword. For practical reasons, SSE schemes often return a set of identifier that point to the actual documents. The client can then present

these identifiers to retrieve the documents and decrypt them locally. In this work, we use identifiers and documents interchangeably and allow this to be determined by the application.

More precisely, a database is a set of document/keyword-set pair $\mathsf{DB} = (d_i, W_i)_{i=1}^N$. Let $W = \cup_{i=1}^N W_i$ be the universe of keywords. A keyword search query for w should return all d_i where $w \in W_i$. We denote this subset of DB by $\mathsf{DB}(w)$. A searchable symmetric encryption scheme consists of protocols SSESETUP, SSESEARCH and SSEADD.

- $\langle \sigma, \mathsf{EDB} \rangle \leftrightarrow \text{SSESETUP}\langle (1^\kappa, \mathsf{DB}), \perp \rangle$: SSESETUP takes as client's input a database DB and outputs a secret state σ (for the client), and an encrypted database EDB which is outsourced to the server.
- $\langle (\mathsf{DB}(w), \sigma'), \mathsf{EDB}' \rangle \leftrightarrow \text{SSESEARCH}\langle (\sigma, w), \mathsf{EDB} \rangle$: SSESEARCH is a protocol between the client and the server, where client's input is the secret state σ and the keyword w he is searching for. Server's input is the encrypted database EDB. Client's output is the set of documents containing w, i.e. $\mathsf{DB}(w)$ as well an updated secret state σ' and the server obtains an updated encrypted database EDB'.
- $\langle \sigma', \mathsf{EDB}' \rangle \leftrightarrow \text{SSEADD}\langle (\sigma, d), \mathsf{EDB} \rangle$: SSEADD is a protocol between the client and the server, where client's input is the secret state σ and a document d to be inserted into the database. Server's input is the encrypted database EDB. Client's output is an updated secrete state σ' and the server's output is an updated encrypted database EDB' which now contains the new document d.

Correctness. Consider the following correctness experiment. An adversary A chooses a database DB. Consider the encrypted database EDB generated using SSESETUP (i.e., $\langle \mathsf{EDB}, K \rangle \leftrightarrow \text{SSESETUP}\langle (1^\kappa, \mathsf{DB}), \perp \rangle$). The adversary then adaptively chooses keywords to search and documents to add to the database. Denote the searched keywords by $w_1, \ldots, w_t$. A wins in the correctness game if $\langle (\mathsf{DB}_i(w_i), \sigma_i), \mathsf{EDB}_i \rangle \neq \text{SSESEARCH}\langle (\sigma_{i-1}, w_i), \mathsf{EDB}_{i-1} \rangle$ for any $1 \leq i \leq t$, where $\mathsf{DB}_i, \mathsf{EDB}_i$ are the database and encrypted database, respectively, after the i-th search, and SSESEARCH and SSEADD are run between an honest client and server. The SSE scheme is correct if probability of A in winning the game is negligible in κ.

Semi-Honest Security. Security of SSE schemes is parametrized by a leakage function $\mathcal{L}$, which explains what the adversary (the server) learns about the database and the search queries, while interacting with a secure SSE scheme. A SSE Scheme is $\mathcal{L}$-secure if for any PPT adversary A, there exist a simulator Sim such that the following two distributions are computationally indistinguishable.

- $\text{Real}_A(\kappa)$: A chooses DB. The experiment then runs

$$\langle \mathsf{EDB}, \sigma \rangle \leftrightarrow \text{SSESETUP}\langle (1^\kappa, \mathsf{DB}), \perp \rangle .$$

  A then adaptively makes search queries w_i, which the experiment answers by running the protocol $\langle \mathsf{DB}_{i-1}(w_i), \sigma_i \rangle \leftrightarrow \text{SSESEARCH}\langle (\sigma_{i-1}, w_i), \mathsf{EDB}_{i-1} \rangle$. Denote the full transcripts of the protocol by t_i. Add queries are handled in a similar way. Eventually, the experiment outputs $(\mathsf{EDB}, t_1, \ldots, t_q)$ where q is the total number of search/add queries made by A.

– $\text{Ideal}_{\mathsf{A},\mathsf{Sim},\mathcal{L}}(\kappa)$: A chooses DB. The experiment runs $(\mathsf{EDB}', st_0) \leftrightarrow \mathsf{Sim}(\mathcal{L}(\mathsf{DB}))$. On any search query w_i from A, the experiment adds $(w_i, search)$ to the history H, and on an add query d_i it adds (d_i, add) to H. It then runs $(t'_i, st_i) \leftrightarrow \mathsf{Sim}(st_{i-1}, \mathcal{L}(\mathsf{DB}_{i-1}, H))$. Eventually, the experiment outputs $(\mathsf{EDB}', t_1, \ldots, t'_q)$ where q is the total number of search/add queries made by A.

Leakage and Full Security. The level of security one obtains from a SSE scheme depends on the leakage function $\mathcal{L}$. We say that a SSE scheme is *fully-secure* if $\mathcal{L}$ only outputs the total number of (w, d) pairs $\sum_{w \in W} |\mathsf{DB}(w)|$, the total numer of unique keywords $|W|$ and $|\mathsf{DB}(w)|$ for any searched keyword w. Achieving this level of security is only possible if the SSESEARCH operation outputs the documents themselves to the client. If instead (as is common for applications with large document sizes), it returns document identifiers which the client then uses to retrieve the actual documents, any SSE protocol would also leak the access pattern.

3 TWORAM: Oblivious RAM in two rounds

We are now ready to describe our round-optimal ORAM construction. We will use an abstract view of a tree-based ORAM scheme, and specifically that of *Path ORAM* [28]. We start by describing this abstract view informally. Then we show how to turn the interactive path ORAM protocol (e.g., the one by Stefanov et al. [28]) into a two-round ORAM protocol, using the abstraction that we present below. We now give some necessary notation that we need for understanding our abstraction.

3.1 Some necessary notation

Let $n = 2^L$ be the size of the initial memory that we wish to access obliviously. This memory is denoted by $A_L[1], A_L[2], \ldots, A_L[n]$ where $A_L[i]$ is the i-th *block* of the memory. Given location y that we wish to access, let $y_L, y_{L-1}, \ldots, y_1$ be defined as $y_L = y$ and $y_i = ceil(y_{i+1}/2)$, for all $i = L-1, L-2, \ldots, 1$. For example, for $L = 4$ and $y = 13$, we have

– $y_1 = ceil(ceil(ceil(y/2)/2)/2) = 2$.
– $y_2 = ceil(ceil(y/2)/2) = 4$.
– $y_3 = ceil(y/2) = 7$.
– $y_4 = 13$.

Also define $b_i = 1 - y_{i+1} \% 2$ to be a bit (namely b_i indicates if y_i is even or not). Finally, on input a $2L$ bit value x, $\mathsf{select}(x, 0)$ selects the first L bits, while $\mathsf{select}(x, 1)$ select the last L bits. We note here that both y_i and b_i are functions of y, but we do not indicate this explicitly so that not to clutter notation.

3.2 Path ORAM abstraction

We start by describing our abstraction of Path ORAM construction. In Appendix A we describe formally how this abstraction can be used to implement the interactive path

ORAM algorithm [28] (with $\log n$ rounds of interaction). We note that the details in the appendix are provided only for helping better understanding. Our construction can be understood based on just the abstraction defined below.

Path ORAM algorithms encode memory A_L in the form of L memories

$$A_L, A_{L-1}, \ldots, A_1 .$$

Each A_i has 2^i entries, each one storing blocks of $2L$ bits. Memories $A_L, A_{L-1}, \ldots, A_2$ are stored in trees $T_L, T_{L-1}, \ldots, T_2$ respectively. Memory A_1 is kept locally by the client. The invariant that is maintained is that any block $A_i[x]$ will reside in *some* leaf-to-root path of tree T_i, and specifically on the path that starts from leaf x_i in T_i. The value x_i itself can be retrieved by accessing A_{i-1}, as we detail in the following.

Reading a value $A_L[y]$. To read a value $A_L[y]$, one first reads $A_1[y_1]$ from local storage and computes $x_2 \leftarrow \mathsf{select}(A_1[y_1], b_1)$ (recall definitions of y_1 and b_1 from Section 3.1). Then one traverses the path starting from leaf x_2 in T_2. This path is denoted with $T_2(x_2)$. Block $A_2[y_2]$ is guaranteed to be on $T_2(x_2)$. Then one computes $x_3 \leftarrow \mathsf{select}(A_2[y_2], b_2)$, and continues in this way. In the end, one will traverse path $T_L(x_L)$ and will eventually retrieve block $A_L[y]$. See Figure 1.

Updating the paths. Once the above process finishes, we need to make sure that we do not access the same leaf-to-root paths in case we access $A_L[y]$ again in the future. Thus we set $\mathsf{select}(A_i[y_i], b_i) \leftarrow r_{i+1}$, where r_{i+1} is a fresh random number that replaces x_{i+1} from above, and we ensure $A_{i+1}[y_{i+1}]$ gets moved to a node on the leaf-to-root path $T_{i+1}[r_{i+1}]$. This is achieved by reading $T_i(x_i)$ into a local data structure C_i called *stash*, and by *evicting* blocks from C_i based on the new assignments. In our abstraction, the stash C_i is viewed as an extension of the root of tree T_i. We note here that the details of the eviction process are not important to us, besides the fact that only changes along the read paths need to be made.

Syntax. A *Path ORAM* consists of three procedures (INITIALIZE, EXTRACT, UPDATE) with syntax:

- $\mathcal{T} \leftarrow \text{INITIALIZE}(1^\kappa, A_L)$: Given a security parameter κ and memory A_L as input, SETUP outputs a set of $L-1$ trees $\mathcal{T} = \{T_2, T_3, \ldots, T_L\}$ and an array of two entries A_1. A_1 is stored locally with the client and $T_2, \ldots, T_L$ are stored with the server.
- $x_{i+1} \leftarrow \text{EXTRACT}(i, y, T_i(x_i))$ for $i = 2, \ldots, L$. Given the tree number i, the final memory location of interest y and a leaf-to-root path $T_i(x_i)$ (that starts from leaf x_i) in tree T_i, EXTRACT outputs an index x_{i+1} to be read in the next tree T_{i+1}. The client can obtain x_2 from local storage as $x_2 \leftarrow \mathsf{select}(A_1[y_1], b_1)$. The obtained value x_2 is sent to the server in order for the server to continue execution. Finally, the server outputs x_{L+1}, which is the desired value $A_L[y]$.

  EXTRACTBUCKET algorithm. In path ORAM [28], internal nodes of the trees store more than one block $(z, A_i[z])$, in the form of *buckets*. We note that EXTRACT can be broken to work on individual buckets along a root-to-leaf path in a tree T_i. In particular, we can define the algorithm $\pi \leftarrow \text{EXTRACTBUCKET}(i, y, b)$ where i is the tree of interest, y is the memory location that needs to be accessed, and b is a bucket corresponding to a particular node on the leaf-to-root path. π will be found

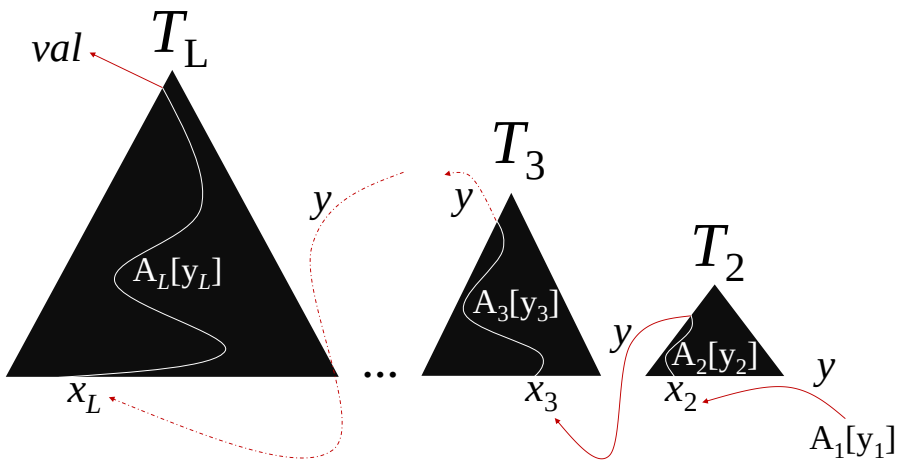

Fig. 1: Our Path ORAM abstraction for reading a value $val = A_L[y]$. $A_1[y_1]$ is read from local storage and defines x_2. x_2 defines a path p_2 in T_2. By traversing p_2 the algorithm will retrieve $A_2[y_2]$, which will yield x_3, which defines a path p_3 in T_3. Repeating this process yields a path p_L in T_L, traversing which yields the final value $A_L[y_L] = A_L[y]$. Note that y is passed from tree T_{i-1} to tree T_i so that the index y_i (and the bit b_i) can be computed for searching for the right block on path p_i.

at one of the nodes on the leaf-to-root path. Note that the algorithm EXTRACT can be implemented by repeatedly calling EXTRACTBUCKET for every b on $T_i(x_i)$.

– $\{A_1, T_2(x_2), \ldots, T_L(x_L)\} \leftarrow$ UPDATE$(y, op, val, A_1, T_2(x_2), \ldots, T_L(x_L)$. Procedure UPDATE takes as input the leaf-to-root paths (and the local storage) that were traversed during the access and accordingly updates these paths (and the local storage). Additionally, UPDATE ensures the new value val is written to $A_L[y]$, if operation op is a "write" operation.

An implementation of the above abstractions, for path ORAM [28], is given in Algorithms 1, 2, 3 in Appendix A.1. Note that the description of the UPDATE procedure [28] abstracts away the details of the eviction strategy. The SETUP and OBLIVIOUSACCESS protocols of the interactive path ORAM *using these abstractions* are given in Figures 6 and 7 respectively in the Appendix A.2. It is easy to see that the OBLIVIOUSACCESS protocol has $\log n$ rounds of interactions. By the proof of Stefanov et al. [28], we get the following:

Corollary 1. *The protocols* SETUP *and* OBLIVIOUSACCESS *from Figures 6 and 7 respectively in Appendix A.2 comprise a $O(\log n)$-round secure ORAM scheme (as defined in Section 2.2), assuming the encryption scheme used is CPA-secure.*

We recall that the bandwidth overhead for path ORAM [28] is $O(\log^3 n)$ bits and the client storage is $O(\log^2 n) \cdot \omega(1)$ bits, for a block size of $2L = 2 \log n$ bits.

3.3 From $\log n$ rounds to two rounds

Existing path ORAM protocols implementing our abstraction require $\log n$ rounds (see OBLIVIOUSACCESS protocol in Figure 7). The main reason for that is the following: In order for the server to figure out the index of leaf x_i from which the next path traversal begins, the server needs to access $A_{i-1}[y_{i-1}]$, which is stored *encrypted* at some node on the path starting from leaf x_{i-1} in tree T_{i-1}—see Figure 1. Therefore the server has to return all encrypted nodes on $T_{i-1}(x_{i-1})$ to the client, who performs the decryption locally, searches for $A_{i-1}[y_{i-1}]$ (via the EXTRACTBUCKET procedure) and returns the value x_i to the server (see Line 10 of the OBLIVIOUSACCESS protocol in Figure 7).

Our approach To overcome this difficulty, we do not encrypt the blocks in the buckets. Instead, for each bucket stored at a tree node u, we prepare a garbled circuit that hardcodes, among other things, the blocks that are contained in the bucket. Subsequently, this garbled circuit executes the EXTRACTBUCKET algorithm on the hardcoded blocks and outputs either $\perp$ or the next leaf index π, depending on whether the search performed by EXTRACTBUCKET was successful or not. The output, whatever that is, is fed as a garbled input to either the left child bucket or the right child bucket (depending on the currently traversed path) or the next root bucket (in case u is a leaf) of u. In this way, by the time the server has executed all the garbled circuits along the currently traversed path, he will be able to pass the index π to the next tree as a garbled input, and continue the execution in the same way without having to interact with the client. Therefore the client can obliviously retrieve his value $A_L[y]$ in only two rounds of communication.

Unfortunately, once these garbled circuits have been consumed, they cannot be used again since this would violate security of garbled circuits. To avoid this problem, the client downloads all the data that was accessed before, decrypts them, runs the UPDATE procedure locally, recomputes the garbled circuits that were consumed before, and stores the new garbled circuits locally. In the next access, these garbled circuits will be sent along with the query. Therefore the total number of communication rounds is equal to two (note that this approach requires permanent client storage—for transient storage, the client will have to send the garbled circuits immediately which would increase the rounds to three). We now continue with describing the bucket circuit that needs to be garbled for our construction.

Naive bucket circuit To help the reader, in Figure 2 we describe a naive version of our bucket circuit that leads to an inefficient construction. Then we give the full-fledged description of our bucket circuit in Figure 3. The naive bucket circuit has *hardcoded parameters*, *inputs* and *outputs*, which we detail in the following.

Hardcoded parameters. The circuit for node u hardcodes:

1. The node identifier u that consists of a triplet (i, j, k) where
   - $i \in \{2, \ldots, L\}$ is the tree number where node u belongs to;
   - $j \in \{0, \ldots, 2^{i-1}\}$ is the depth of node u;
   - $k \in \{0, \ldots, 2^j - 1\}$ is the oder of node u in the specific level.

Circuit $\mathsf{C}[u, \mathsf{bucket}, \mathsf{leftInputs}, \mathsf{rightInputs}, \mathsf{nState}](\mathsf{cState})$
Hardcoded parameters: $[u = (i, j, k), \mathsf{bucket}, \mathsf{leftInputs}, \mathsf{rightInputs}, \mathsf{nState}]$.
Inputs: $\mathsf{cState} = (p, y, \pi)$.
Outputs: Next node to be executed and garbled inputs for its bucket circuit, or final value $A_L[y]$.

1: **if** $\pi = \perp$ **then**
2: Set $\pi \leftarrow$ EXTRACTBUCKET(i, y, bucket); ▷ π will be the desired value x_{i+1} or $A_L[y]$.
3: **end if**
4: **if** u is not a leaf **then**
5: Based on p, **return** either $\left(\mathsf{left}(u), \mathsf{leftInputs}_{(p,y,\pi)}\right)$ or $\left(\mathsf{right}(u), \mathsf{leftInputs}_{(p,y,\pi)}\right)$;
6: **else**
7: **if** $i = L$ **then**
8: **return** π; ▷ Found $A_L[y] = \pi$.
9: **else**
10: **return** $\left(\mathsf{nextRoot}(u), \mathsf{nState}_{(\pi,y,\perp)}\right)$; ▷ Return gargled inputs for next root.
11: **end if**
12: **end if**

Fig. 2: Formal description of the naive bucket circuit. Notation: Given lab, the set of *input labels* for a garbled circuit, we let lab_a denote the garbled input labels (i.e., the labels taken from lab) corresponding to the input value a.

For example, the root of tree T_3 will be denoted $(3, 0, 0)$, while its right child will be $(3, 1, 1)$.

2. The bucket information bucket (i.e., blocks $(x, A_i[x], r)$ contained in node u—recall r is the path index in T_i assigned to $A_i[x]$);
3. The set of *input labels* $\mathsf{leftInputs}$, $\mathsf{rightInputs}$ and nState that can be used to compute the *garbled inputs* for the next circuit to be executed. Note that $\mathsf{leftInputs}$ and $\mathsf{rightInputs}$ are used to prepare the next garbled inputs when node u is an internal node (to go either to the left or the right child), while nState is used when node u is a leaf (to go to the next root).

Inputs. The inputs of the circuit is a triplet cState (current state), consisting of the following information:

1. The index of the leaf p from which the currently explored path begins;
2. The final location to be accessed y;
3. The output from previous bucket π (can be the actual value of the next index to be explored or $\perp$).

Outputs. The outputs of the circuit are the next node to be executed, along with its garbled inputs. For example, if the current node u is not a leaf (see Lines 4 and 5 in Figure 2), the circuit outputs the garbled inputs of either the left or the right child, whereas if the current node is a leaf (see Lines 6-12 in Figure 2), the circuit outputs the garbled inputs of the next root to be executed. Note that outputting the garbled inputs is easy, since the bucket circuit hardcodes the input labels of the required circuits.

Circuit $\mathsf{C}[u, \mathsf{bucket}, \mathsf{leftInputs}, \mathsf{rightInputs}](\mathsf{cState}, \mathsf{nState})$
Hardcoded parameters: $[u = (i, j, k), \mathsf{bucket}, \mathsf{leftInputs}, \mathsf{rightInputs}]$.
Inputs: $\mathsf{cState} = (p, y, \pi), \mathsf{nState}$.
Outputs: Next node to be executed and garbled inputs for its bucket circuit, or final value $A_L[y]$.

1: **if** $\pi = \perp$ **then**
2: Set $\pi \leftarrow$ EXTRACTBUCKET(i, y, bucket); ▷ π will be the desired value x_{i+1} or $A_L[y]$.
3: **end if**
4: **if** u is not a leaf **then**
5: Based on p, **return** either $\left(\mathsf{left}(u), \mathsf{leftInputs}_{(p,y,\pi,\mathsf{nState})}\right)$ or $\left(\mathsf{right}(u), \mathsf{leftInputs}_{(p,y,\pi,\mathsf{nState})}\right)$;
6: **else**
7: **if** $i = L$ **then**
8: **return** π; ▷ Found $A_L[y] = \pi$.
9: **else**
10: **return** $\left(\mathsf{nextRoot}(u), \mathsf{nState}_{(\pi,y,\perp)}\right)$; ▷ Return gargled inputs for next root.
11: **end if**
12: **end if**

Fig. 3: Formal description of the final bucket circuit.

Final bucket circuit In the naive circuit presented before, we hardcode the input labels of the root node root of every tree T_i into all the nodes/circuits of tree T_{i-1}. Unfortunately, in every oblivious access, the garbled circuits of all roots are consumed (and therefore root's circuit as well), hence *all* the garbled circuits of tree T_{i-1} will have to be recomputed from scratch. This cost is $O(n)$, thus very inefficient. We would like to mimimize the number of circuits in T_{i-1} that need to be recomputed and ideally make this cost proportional to $O(\log n)$.

To achieve that, we observe that, *instead of hardcoding the input labels* nState *in the garbled circuit of every node of tree* T_{i-1}*, we can just pass them as inputs*. The final circuit is given in Figure 3. Note that the only difference of the new circuit from the naive circuit is in the computation of the garbled inputs $\mathsf{leftInputs}_{(p,y,\pi,\mathsf{nState})}$ and $\mathsf{rightInputs}_{(p,y,\pi,\mathsf{nState})}$, where nState is added in the subscript (see Line 5 of both Figure 3 and Figure 2).

3.4 Protocols SETUP and OBLIVIOUSACCESS of our construction

We now describe in detail the SETUP and OBLIVIOUSACCESS protocols of TWORAM.

SETUP. The SETUP protocol is described in Figure 4. Just like the setup for the interactive ORAM protocol (see Figure 6 in Appendix A.2), in TWORAM, the client does some computation locally in the beginning (using his secret key) and then outputs some "garbled information" that is being sent to the server. In particular:

1. After producing the trees $T_2, T_3, \ldots, T_L$ using algorithm INITIALIZE, the client prepares the garbled circuit of Figure 3 for all the nodes $u \in T_i$, for all trees T_i. It is important this computation takes place from the leaves towards the root (that is why we write $j \in \{i-1, \ldots, 0\}$ in Line 2 of Figure 4), since a garbled circuit of

Protocol $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$:

Client:

1: Pick a κ-bit secret key s. Run $\{A_1, T_2, \ldots, T_L\} \leftarrow \mathsf{Initialize}(1^\kappa, \mathsf{M})$;

2: For all $i \in 2, \ldots, L$, for all $j \in \{i-1, \ldots 0\}$, for all $k \in \{0, \ldots, 2^j - 1\}$, let $u = (i, j, k)$ be the (j, k)-th node in T_i and let bucket be its bucket. Compute:

$$(\tilde{\mathsf{C}}^u, \mathsf{lab}^u = (\mathsf{cState}^u, \mathsf{nState}^u)) \leftarrow \mathsf{GCircuit}(1^\kappa, \mathsf{C}[u, \mathsf{bucket}, \mathsf{lab}^{(i,j+1,2k)}, \mathsf{lab}^{(i,j+1,2k+1)}]) \,,$$

$$\mathsf{X}^u \leftarrow \mathsf{Enc}_s(\mathsf{bucket}, \mathsf{lab}^{(i,j+1,2k)}, \mathsf{lab}^{(i,j+1,2k+1)}) \,,$$

where C is defined in Figure 3 and $(\mathsf{Enc}, \mathsf{Dec})$ is a semantically-secure encryption scheme;

3: For all u, send to server $\tilde{\mathsf{C}}^u, \mathsf{X}^u$;

4: For all $i \in \{2, \ldots, L\}$ send to server $\beta_i = \mathsf{nState}^{(i,0,0)}_{\mathsf{cState}^{(i+1,0,0)}}$;

5: **return** $(s, A_1, \mathsf{cState}^{(2,0,0)})$ as σ;

Server:

1: **return** all data sent by the client from above as EM;

Fig. 4: SETUP protocol for TWORAM.

a node u hardcodes the input labels of the garbled circuits of its children—so these need to be readily available by the time u's garbled circuit is computed.

2. Apart from the garbled circuits, the client needs to prepare garbled inputs for the nState inputs of all the roots of the trees T_i (recall, that nState contains the set of input labels for the cState of the root of tree T_{i+1}, which are only initialized by the bucket circuits of the leaves of tree T_i). These are essentially the β_i's computed in Line 4 of Figure 4.

OBLIVIOUSACCESS. The OBLIVIOUSACCESS protocol of TWORAM is described in Figure 5. The first step of the protocol is similar to that of the interactive scheme (see Figure 7 in Appendix), where the client accesses local storage A_1 to compute the path index x_2 that must be traversed in T_2. However, the main difference is that, instead of sending x_2 directly, the client sends the *garbled input that corresponds to* x_2 for the root circuit of tree T_2, denoted with α in Figure 5.

We note here that α is not enough for the first garbled circuit to start executing, and therefore the server complements this garbled input with β_2 (see Server Line 1), the other half that was sent by the client before and that represents the input labels of the next root. Subsequently, the server starts executing the garbled circuits one-by-one, using the outputs of the first circuit, as garbled inputs to the second one, and so on. Eventually, the clients retrieves all the paths $T_i(x_i)$, and by decrypting $T_L(x_L)$, the client can retrieve the value he was looking for (see Client Line 2). Finally, the client runs the UPDATE procedure, re-garbles the circuits that got consumed and either sends them to the server, or waits until the next query (in which case TWORAM is two-round with permanent storage). We can now state the main result of our paper.

Theorem 1. *The protocols* SETUP *and* OBLIVIOUSACCESS *from Figures 4 and 5 respectively comprise a two-round secure ORAM scheme (as defined in Section 2.2), as-*

Protocol $\langle(\mathsf{M}[y], \sigma'), \mathsf{EM}'\rangle \leftrightarrow$ OBLIVIOUSACCESS$\langle(\sigma, y, val), \mathsf{EM}\rangle$:

Client:

1: Compute $x_2 \leftarrow \mathsf{select}(A_1[y_1], b_1)$ and send to server $\alpha = \mathsf{cState}^{(2,0,0)}_{(x_2,y,\perp)}$;

Server:

1: Let $\mathsf{output} = \alpha || \beta_2$, where β_2 is defined in Line 4 of Protocol SETUP in Figure 4;
2: Set $i = 2$ and $j = 0$ and $k = 0$;
3: **while** $i \leq L$ **do**
4: $\quad(\mathsf{nextNode}, \mathsf{output}) \leftarrow \mathsf{Eval}(\tilde{\mathsf{C}}^{(i,j,k)}, \mathsf{output})$;
5: $\quad$**if** $j = i - 1$ and $i < L$ **then** ▷ node (i, j, k) is a leaf of trees $T_2, T_3, \ldots, T_{L-1}$
6: $\quad\quad i = i + 1$; ▷ go to next tree
7: $\quad\quad \mathsf{output} = \mathsf{output} || \beta_i$; ▷ prepare the garbled inputs for the root of the next tree
8: $\quad$**end if**
9: $\quad$**if** $\mathsf{nextNode} = \mathsf{left}(i, j, k)$ **then** ▷ node (i, j, k) is a not a leaf; decide whether to go next
10: $\quad\quad j = j + 1$; $k = 2k$; ▷ go to left child
11: $\quad$**else**
12: $\quad\quad j = j + 1$; $k = 2k + 1$; ▷ go to right child
13: $\quad$**end if**
14: **end while**
15: Let $x_2, \ldots, x_L$ be the indices of the paths that have been accessed above;
16: Send to client output and all X^u's corresponding to nodes u on paths $T_2(x_2), T_3(x_3), \ldots, T_L(x_L)$;

Client:

1: Decrypt all information contained in X^u and reconstruct $T_2(x_2), T_3(x_3), \ldots, T_L(x_L)$;
2: Retrieve block $(y, A_L[y], x_L)$ from $T_L(x_L)$; Set $val = A_L[y]$;
3: Run $\{A_1, T_2(x_2), \ldots, T_L(x_L)\} \leftarrow$ UPDATE$(y, op, val, A_1, T_2(x_2), \ldots, T_L(x_L))$;
4: For all $i \in 2, \ldots, L$, for all $j \in \{i - 1, \ldots 0\}$, let $u = (i, j, k)$ be the (j, k)-th node in $T_i(x_i)$ and let bucket be its bucket. Compute:

$$(\tilde{\mathsf{C}}^u, \mathsf{lab}^u = (\mathsf{cState}^u, \mathsf{nState}^u)) \leftarrow \mathsf{GCircuit}(1^\kappa, \mathsf{C}[u, \mathsf{bucket}, \mathsf{lab}^{(i,j+1,2k)}, \mathsf{lab}^{(i,j+1,2k+1)}]),$$

$$\mathsf{X}^u \leftarrow \mathsf{Enc}_s(\mathsf{bucket}, \mathsf{lab}^{(i,j+1,2k)}, \mathsf{lab}^{(i,j+1,2k+1)}),$$

$\quad$where C is defined in Figure 3. Send to server $\tilde{\mathsf{C}}^u$ and X^u;
5: Store locally $(s, A_1, \mathsf{cState}^{(2,0,0)})$ as σ';
6: For all $i \in \{2, \ldots, L\}$ send to server $\beta_i = \mathsf{nState}^{(i,0,0)}_{\mathsf{cState}^{(i+1,0,0)}}$;
7: **return** (val, σ').

Server:

1: **return** the data received by the client as EM'.

Fig. 5: OBLIVIOUSACCESS protocol for TWORAM.

suming the garbling scheme used is secure (as defined in Section 2.1) and the encryption scheme used is CPA-secure.

The proof of the above theorem can be found in Appendix A.3. Concerning complexity, it is clear that the only overhead that we are adding on path ORAM [28] is a garbled circuit per bucket—this adds a multiplicative security parameter factor on all the complexity measures of path ORAM. E.g., the bandwidth overhead of our construction is $O(\kappa \cdot \log^3 n)$ bits (for blocks of $2 \log n$ bits). subsectionOptimizations Recall

that in the garbling procedure of a circuit C, one has the following choices: (i) either to garble C in a way that during evaluation of the garbled circuit on x the output is the cleartext value $C(x)$; (ii) or to garble C in a way that during evaluation of the garbled circuit on x the output is the garbled labels corresponding to the value $C(x)$. We now describe an optimization for a specific circuit C that we will be using in our construction that uses the above observation.

General optimization. Consider a circuit that performs the following task: It hardcodes two k-bit strings s_0 and s_1, takes an input a bit b and outputs s_b. This cleartext circuit has size $O(k)$, so the garbled circuit for that will have size $O(k^2)$. To improve upon that we consider a circuit C' that takes as input bit b and outputs the same bit b! This cleartext circuit has size $O(1)$. However, to make sure that the output of the garbled version of C' is always s_b, we garble C' by outputting the garbled label corresponding to b, namely s_b (i.e., using (ii) from above). In particular, during the garbling procedure we use s_0 as the garbled label output for output $b = 0$ and we use s_1 as the garbled label output for the output $b = 1$. Note that the size of the new garbled circuit has size $O(k)$, yet it has exactly the same I/O behavior with the garbling of C, which has size $O(k^2)$.

- **Improving** cState**—not hard-coding input labels inside the bucket circuit.** In the construction we described, we include the input labels $\mathsf{leftInputs}, \mathsf{rightInputs}$ in the circuit $\mathsf{C}[u, \mathsf{bucket}, \mathsf{leftInputs}, \mathsf{rightInputs}]$. Consequently, the size of the ungarbled version of this circuit grows with the size of $\mathsf{leftInputs}$ and $\mathsf{rightInputs}$ which is $\kappa \cdot |\mathsf{cState}|$. We can easily use the general optimization described above, for each bit of $|\mathsf{cState}|$, to make the size of the ungarbled version of our circuit only grow with $|\mathsf{cState}|$.
- **Improving** nState**—input labels passing.** In the construction described previously, for each tree, an input value nState is passed from the root to a leaf node in the tree. However this value is used only at the leaf node. Recall that the nState value passed from the root to a leaf garbled circuits in the tree T_i is exactly the value $\mathsf{cState}^{i+1,0,0}$, the input labels of the root garbled circuit of the tree T_{i+1}. Since each ungarbled circuit gets this value as input, therefore each of one of them needs to grow with $\kappa \cdot |\mathsf{cState}|$.[7] We will now describe an optimization such that the size of the garbled version, rather than the clear version, grows linearly in $\kappa \cdot |\mathsf{cState}|$.

  Note that in our construction the value $\mathsf{cState}^{i+1,0,0}$ is not used at all in the intermediate circuits as it gets passed along the garbled circuits for tree T_i. In order to avoid this wastefulness, for all nodes $i \in \{1, \ldots, L\}, j \in [i], k \in [2^j]$ we sample a value $r^{(i,j,k)}$ of length $\kappa \cdot |\mathsf{cState}|$ and hardcode the values $r^{(i,j,k)} \oplus r^{(i,j+1,2k)}$ and $r^{(i,j,k)} \oplus r^{(i,j+1,2k+1)}$ inside the garbed circuit $\tilde{C}^{i,j,k}$ which output the first of two values if the execution goes left and the second if the execution goes right. Note that a garbled circuits grows only additively in $\kappa \cdot |\mathsf{cState}|$ because of this change. This follows by using the first optimization. Additionally, we include the value $\mathsf{cState}^{i+1,0,0} \oplus r^{(i,0,0)}$ with the root node of the tree T_i. The leaf garbled circuit $(i, i-1, k)$ in tree T_i is constructed assuming $r^{(i,i-1,k)}$ is the sequence

[7] This efficiency is achieved when the first optimization is used.

of input labels for the root garbled circuit of the tree T_{i+1}.[8] Let $\alpha_0, \ldots \alpha_{i-1}$ be the strings output during the root to a leaf traversal in tree T_i. Now observe that $\mathsf{cState}^{i+1,0,0} \oplus r^{(i,0,0)} \oplus_{j\in[i]} \alpha_j$ is precisely $\mathsf{cState}^{i+1,0,0} \oplus r^{(i,i-1,k)}$ where k is the leaf node in the traversed path. At this point it is easy to see that given the output of the leaf grabled circuit for tree T_i one can compute the required input labels for the root of tree T_{i+1}.

The update mechanism in our construction can be easily adapted to work with this change. Here note that we would now include the values $r^{(i,j,k)}, r^{(i,j+1,2k)}$ and $r^{(i,j+1,2k+1)}$ in the ciphertext $X^{(i,j,k)}$. Also note that we will use fresh $r^{(\cdot,\cdot,\cdot)}$ values whenever a fresh garbled circuit for a node is generated. The security argument now additionally uses the fact that the outputs generated by garbled circuits in two separate root to leaf traversals depend on completely independent $r^{(\cdot,\cdot,\cdot)}$ values.

Note that the above modification leaks what value is passed by the executed leaf garbled circuit in tree T_i to the root garbled circuit in tree T_{i+1}. This can be deduced based on what bit values of $\mathsf{cState}^{i+1,0,0} \oplus r^{(i,0,0)}$ are revealed. This can be tackled by randomly permuting the labels in $\mathsf{cState}^{i+1,0,0}$ and passing the information on this permutations along with in the tree to leaf garbled circuits. Note that the size of this information is small.

Taken together these two optimizations reduce the size of each garbled circuit to $O(\kappa \cdot (|\mathsf{bucket}| + |\mathsf{cState}|))$. Since $|\mathsf{bucket}| > |\mathsf{cState}|$ this expression reduces to $O(\kappa \cdot |\mathsf{bucket}|)$. This implies that the overhead of our construction is just κ times the overhead of the underlying Path ORAM scheme.

4 From ORAM to SSE

The natural way of designing a fully-secure SSE scheme using an ORAM scheme is to first use a data structure for storing keyword-document pairs, setup the data structure in memory using an ORAM setup and then read/write from it using ORAM operations. Since ORAM hides the read/write access patterns, but it does not hide the number of memory accesses, in order to obtain a fully-secure SSE scheme, one needs to ensure that in the data structure used, the number of memory accesses for each operation is data-independent. Fortunately, there are hash table implementations that have this property and our abstraction of a hash table captures this as well.

We start by describing constructions that can be instantiated using any ORAM scheme. We then show how to obtain a significantly more efficient instantiation using a combination of TWORAM and a non-recursive path ORAM scheme.

4.1 SSE from any ORAM

First approach. The common way of storing a database of documents in a hash table is to insert a key-value pair (w, d) into the table for any keyword w in a document d.

[8] Note that here the first optimization allows us to ensure that the size of the garbled leaf circuit, rather than the clear leaf circuit, grows with the length of $r^{(i,i-1,k)}$ as these hard-codings are performed.

Searching for a document with keyword w then reduces to looking up w in the table. If there is more than one document containing a keyword w, a natural solution is to create a bucket B_w stroing all the documents containing w and storing the bucket in position pt_w of an array A. One then inserts (w, pt_w) in a hash table. Now, to search for a keyword w, we first look up (w, pt_w), and then access $\mathsf{A}[pt_w]$ to obtain the bucket B_w of all the desired documents. A subtle issue is that the distribution of bucket sizes would leak information about the database even before any keyword is searched. As a result, for this approach to be fully-secure, one needs to pad each bucket to an upperbound on the number of searchable documents per keyword.

Next we describe the SSE scheme more formally. Given a hash table $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$, and an ORAM scheme $ORAM = (\textsc{Setup}, \textsc{ObliviousAccess})$, we construct an SSE scheme $(\textsc{SSESetup}, \textsc{SSESearch}, \textsc{SSEAdd})$ as follows.

1. $\langle \sigma, \mathsf{EDB} \rangle \leftrightarrow \textsc{SSESetup}\langle (1^\kappa, max, \mathsf{DB}), \perp \rangle$: Given an initial set of documents DB, client lets S be the set of key-value pairs (w, pt_w) where pt_w is an index to an array of buckets A such that $\mathsf{A}[pt_w]$ stores the bucket of all documents in DB containing w. Each bucket is padded to the maximum size max with dummy documents.
   Client first runs $\mathsf{hsetup}(S, size)$ to obtain (h, M). $size$ is the anticipated maximum size of hash table H. Then client and server run $\langle \sigma_1, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$. Cleint and server also run $\langle \sigma_2, \mathsf{EA} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{A}), \perp \rangle$
   Note that server's output is $\mathsf{EDB} = (\mathsf{EM}, \mathsf{EA})$ and client's output is $\sigma = (\sigma_1, h, \sigma_2)$.
2. $\textsc{SSESearch}\langle (\sigma, w), \mathsf{EDB} \rangle$: Client computes $i_1, \ldots, i_c \leftarrow h(w)$. Then, client and server run $\textsc{ObliviousAccess}\langle ((\sigma_1, i_j, \mathsf{null}), \mathsf{EM} \rangle$ for $j \in \{1, \ldots, c\}$ for client to obtain $\mathsf{M}[i_j]$. If client does not find (w, pt_w) in one of the retrieved locations it lets $pt_w = 0$, corresponding to a dummy access to the index 0 in A.
   Client and server then run $\textsc{ObliviousAccess}\langle (\sigma_2, pt_w, \mathsf{null}), \mathsf{EA} \rangle)$ for client to obtain the bucket B_w stored in $\mathsf{A}[pt_w]$. Client outputs all the non-dummy documents in B_w.
3. $\textsc{SSEAdd}\langle (\sigma, d), \mathsf{EDB} \rangle$: For every w in d, client computes $i_1, \ldots, i_c \leftarrow h(w)$ and client and server run $\textsc{ObliviousAccess}\langle (\sigma_1, i_j, \mathsf{null}), \mathsf{EM} \rangle$ for $j \in \{1, \ldots, c\}$ for client to obtain $\mathsf{M}[i_j]$. If (w, pt_w) is in the retrieved locations let i_j^* be the location it was found at. If not, let pt_w be the first empty location in A, and let $i*_j$ be the first empty location from the retrieved ones in M. Client and server run $\textsc{ObliviousAccess}\langle (\sigma_1, i_j^*, (w, pt_w)), \mathsf{EM} \rangle$.
   Client and server run $\textsc{ObliviousAccess}\langle (\sigma_2, pt_w, \mathsf{null}), \mathsf{EA} \rangle$ to retrieve $\mathsf{A}[pt_w]$. Let B_w be the retrieved bucket. Client inserts d in the first dummy entry of B_w, denoting the new bucket by B'_w. Client and server run $\textsc{ObliviousAccess}\langle (\sigma_2, pt_w, B'_w), \mathsf{EA} \rangle$.

The main disadvantage of the above construction is that we need to anticipate an upperbound on the bucket sizes, and pad all bukets to that size. Given that in practice there are often keywords that appear in a large number of documents, and keywords that only appear in a few, the padding will lead to inefficiency. Our next solution addresses this issue but instead has a higher round complexity.

Second approach. Instead of storing all documents matchig a keyword w in one bucket, we store each of them separately in the hash table, using a different keyword. In particular, we can store the key-value pair $(w||i, d)$ in the hash table for the ith document d containing w. This works fine except that it requires looking up $w||count$ for an incremental counter $count$ until the keyword is no longer found in the table.

To make this approach cleaner and the write operations more efficient, we maintain two hash tables, one for storing the counter representing the number of documents containing the keyword, and one storing the incremental key-value pairs as described above. To lookup a keyword w, one first looks up the counter $count$ in the first table and then makes $count$ lookup queries to the second table.

We describe the above SSE scheme in more detail below. Given a hash table $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$ and an ORAM scheme $ORAM = (\textsc{Setup}, \textsc{ObliviousAccess})$, we construct an SSE scheme $(\textsc{SSESetup}, \textsc{SSESearch}, \textsc{SSEAdd})$ as follows:

1. $\langle \sigma, \mathsf{EDB} \rangle \leftrightarrow \textsc{SSESetup}\langle (1^\kappa, \mathsf{DB}), \perp \rangle$: Given an initial set of documents DB. Let S_1 be the set of $(w, count_w)$ pairs and S_2 be the set of key-value pairs $(w||i, d_i)$ for $1 \leq i \leq count_w$ where $count_w$ is the number of documents containing w, and d_i denotes the ith document in DB containing w.
   Cleint runs $\mathsf{hsetup}(S_i, size_i)$ to obtain (h_i, M_i). $size_i$ is the anticipated maximum size of the hash table H_i. Then client and server run $\langle \sigma_i, \mathsf{EM}_i \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}_i), \perp \rangle$. Note that server's output is $\mathsf{EDB} = (\mathsf{EM}_1, \mathsf{EM}_2)$ and client's output is $\sigma = (\sigma_1, \sigma_2, h_1, h_2)$.
2. $\textsc{SSESearch}\langle (\sigma, w), \mathsf{EDB} \rangle$: Client computes $i_1, \ldots, i_c \leftarrow h_1(w)$ and client and server run $\textsc{ObliviousAccess}\langle (\sigma_1, i_j, \mathsf{null}), \mathsf{EM}_1 \rangle)$ for $j \in \{1, \ldots, c\}$ for client to obtain $(w, count_w)$ among the retrieved locations. If such a pair is not found, client lets $count_w = 0$.
   For $1 \leq k \leq count_w$, client computes $i_1^k, \ldots, i_c^k \leftarrow h_2(w||k)$ and client and server run $\textsc{ObliviousAccess}\langle (\sigma_2, i_j^k, \mathsf{null}), \mathsf{EM}_2 \rangle)$ for $j \in \{1, \ldots, c\}$ for client to obtain $\mathsf{M}_2[i_j^k]$. Client outputs d for all d where $(w||k, d)$ is in the retrieved locations from M_2.
3. $\textsc{SSEAdd}\langle (\sigma, d), \mathsf{EDB} \rangle$: For every w in d, client computes $i_1, \ldots, i_c \leftarrow h_1(w)$ and client and server run $\textsc{ObliviousAccess}\langle (\sigma_1, i_j, \mathsf{null}), \mathsf{EM}_1 \rangle$ for $j \in \{1, \ldots, c\}$ for client to obtain $\mathsf{M}_1[i_j]$. If $(w, count_w)$ is in the retrieved locations let i_j^* be the location it was found at. If not, let $count_w = 0$ and let i_j^* be the first empty location from the retrieved ones. Client and server run $\textsc{ObliviousAccess}\langle (\sigma_1, i_j^*, (w, count_w + 1)), \mathsf{EM}_1 \rangle$ to increase the counter by one.
   Client then computes $i'_1, \ldots, i'_c \leftarrow h_2(w||count_w + 1)$ and client and server run $\textsc{ObliviousAccess}\langle (\sigma_2, i'_j, \mathsf{null}), \mathsf{EM}_2 \rangle$ to retrieve $\mathsf{M}_2[i'_j]$ for $j \in \{1, \ldots, c\}$. Let i'_k be the first empty location among them. Client and server run $\textsc{ObliviousAccess}\langle (\sigma_2, i'_k, (w||count + 1)), \mathsf{EM}_2 \rangle$.

The main disadvantage of our second approach is that for each search, it requires $count_w$ ORAM accesses to retrieve all matching documents. This means that the bandwidth/computation overhead of ORAM scheme is multipled by $count_w$ which can be large for some keywords. More importantly, it would require $O(count_w)$ rounds since the ORAM accesses cannot be paralleized in our constant-round ORAM construction.

In particular, note that each memory garbled circuit in the construction can only be used once and needs to be replaced before the next memory access. Finally, the constant-round ORAM needs to store a memory array that is proportional to the number of (w, d) tuples associated with the database, which is significantly larger than the number of unique keywords, increasing the storage obverhead of the resulting SSE scheme.

Next, we address all these efficiency concerns, showing a cosntruction that only requires a single ORAM access using our constant-round construction.

4.2 SSE from Path ORAM

The idea is to not only store a per-keyword counter $count_w$ as before, but also to store a $access_w$ that represents the number of search/add queries performed on w so far. Similar to the previous approach, the tuple $(w, (count_w, access_w))$ is stored in a hash table that is implemented using our constant-round ORAM scheme TWORAM. The $count_w$ is incremented whenever a new document containing w is added and the $access_w$ is incremented after each search/add query for w.

The tuples $(w||i, d_i)$ for all d_i containing w are then stored in a one-level (non-recursive) path ORAM. In order to avoid storing a large client-side position map for this non-recursive path ORAM, we generate/update the positions pseudorandomly using a PRF $F_K(w||i||access_w)$. Since each document d_i has a different index and each search/add query for w will increment $access_w$, the pseudorandomness property of F ensures that this way of generating the position maps is indisntinguihsable from generating them at random. Now the client only needs to keep the secret key K. Note that since we are using a one-level path ORAM to store the documents, we can handle multiple parallel accesses without any problems, hence obtaining a constant-round search/add complexity. Furthermore, we only access TWORAM(which uses garbled circuits) once per keyword search to retrieve the tuple $(w, (count_w, access_w))$, so TWORAM's overhead is not multiplied by $count_w$ for each search/add query. Similary, the storage overhead of TWORAMis only for a memory array of size $|W|$ (number of unique keywords in documents) which is significantly smaller than the number of keyword-doucment pairs needed in the general approach.

We need to make a few small modifications to the syntax of the abstraction for path ORAM here. First, since we generate the position map on the fly using a PRF, it is convenient to modify the syntax of the UPDATE procedure to take the new random position as input, instead of internally generating it in our original syntax. Also, since we are not extracting an index y from the path ORAM and instead are extracting a tuple of the form $(w||i, d_i)$, we will pass $w||i$ as input in place of y in the EXTRACT and UPDATE operations.

We now describe the SSE scheme in detail. Given a hash table $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$, our countant-round ORAM scheme
$\mathsf{TWORAM} = (\text{SETUP}, \text{OBLIVIOUSACCESS})$, a single-level (non-recursive) Path ORAM scheme with procedures
(INITIALIZE, EXTRACT, UPDATE), and a PRF function F, we construct an SSE scheme (SSESETUP, SSESEARCH, SSEADD) as follows:

1. $\langle \sigma, \mathsf{EDB} \rangle \leftrightarrow \text{SSESETUP}\langle (1^\kappa, \mathsf{DB}), \perp \rangle$: Given an initial set of documents DB, let S be the set of $(w, (count_w, access_w = 0))$ where $count_w$ is the number of docu-

ments containing w, and $access_w$ denotes the number of times the keyword w has been searched/added.

Client runs $\mathsf{hsetup}(S, size)$ to obtain (h, M). $size$ is the anticipated maximum size of the hash table H. Then client and server run $\langle \sigma_s, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$.

Let A_L be an initially empty memory array with a size that estimates an upper bound on total number of (w, d) pairs ind DB. Client runs $\mathcal{T} \leftarrow \textsc{Initialize}(1^\kappa, A_L)$, and only sends the tree T_L for the last level to server, and discards the rest.

Client generates a PRF key $K \leftarrow \{0,1\}^\kappa$.

For every item $(w, (count_w, access_w))$ in S, and for $1 \le i \le count_w$ (in parallel):

(a) Client lets $val_{w,i} = (w||i, d_i)$ where d_i denotes the ith document in DB containing w.

(b) Client lets $x_{w,i} = F_K(w||i||acess_w)$ and sends $x_{w,i}$ to server who returns the encrypted buckets on path $T_L(x_{w,i})$ which client decrypts itself.

(c) Client then runs $\{T_L(x_{w,i})\} \leftarrow \textsc{Update}(w||i, write, val_{w,i}, T_L(x_{w,i}), x'_{w,i} = F_K(w||i||acess_w{+}1))$ to insert $val_{w,i}$ into the path along its new path $T_L(x'_{w,i})$. Client then encrypts the updated path $T_L(x_{w,i})$ and sends it to server who updates T_L.

Note that server's output is $\mathsf{EDB} = (\mathsf{EM}, T_L)$ and client's output is $\sigma = (\sigma_s, h, K)$.

2. $\textsc{SSESearch}\langle(\sigma, w), \mathsf{EDB}\rangle$: Client computes $i_1, \ldots, i_c \leftarrow h(w)$ and client and server run $\textsc{ObliviousAccess}\langle(\sigma_s, i_j, \mathsf{null}), \mathsf{EM}\rangle)$ for $j \in \{1, \ldots, c\}$. If client finds $(w, (count_w, access_w))$ in one of the retrieved locations, let i^*_j be the location it was found at. If such a pair is not found the search ends here. Client and server run $\textsc{ObliviousAccess}\langle(\sigma_s, i^*_j, (w, count_w, access_w + 1)), \mathsf{EM}\rangle$ to increase the $access_w$ by one.

   For $1 \le i \le count_w$ (in parallel):

   (a) Client lets $x_{w,i} = F_K(w||i||acess_w)$ and sends $x_{w,i}$ to server who returns $T_L(x_{w,i})$ which client decrypts.

   (b) Client runs $(w||i, d_i) \leftarrow \textsc{Extract}(L, w||i, T_L(x_{w,i}))$, and outputs d_i. Client then runs $\{T_L(x_{w,i})\} \leftarrow \textsc{Update}(w||i, read, (w||i, d_i), T_L(x_{w,i}, x'_{w,i} = F_K(w||i||$ $access_w + 1))$ to update the location of $(w||i, d_i)$ to $x'_{w,i}$. Client then encrypts the updated path and sends it to server to update T_L.

3. $\textsc{SSEAdd}\langle(\sigma, d), \mathsf{EDB}\rangle$:

   For every w in d:

   (a) Client computes $i_1, \ldots, i_c \leftarrow h(w)$ and client and server run $\textsc{ObliviousAccess}\langle($ $\sigma_s, i_j, \mathsf{null}), \mathsf{EM}\rangle)$ for $j \in \{1, \ldots, c\}$. If client finds $(w, (count_w, access_w))$ in one of the retrieved locations, let i^*_j be the location it was found at. Else, it lets i^*_j be the first empty location among the retrieved ones.

   (b) Client and server run $\textsc{ObliviousAccess}\langle(\sigma_s, i^*_j, (w, (count_w{+}1, access_w{+}$ $1))), \mathsf{EM}\rangle$ to increase $count_w$ and $access_w$ by one.

   (c) Client lets $x_{w,count_w} = F_K(w||count_w||acess_w)$ and sends $x_{w,count_w}$ to server who returns encrypted $T_L(x_{w,count_w})$ back. Client decrypts the path.

   (d) Client lets $x' = F_K(w||count_w{+}1||acess_w{+}1)$ and runs $\{T_L(x_{w,count_w})\} \leftarrow$ $\textsc{Update}(w||i, write, (w||count_w{+}1, d), T_L(x_{w,count_w}), x')$ to update the path. Client then encrypts the updated path and sends it to server to update T_L.

Before stating the security theorem for the above SSE scheme, we first need to make the leakage function associated with the scheme more precise. The leakage function $\mathcal{L}(\mathsf{DB}, H)$ for our scheme outputs the following (DB is the database and H is the search/add history): $|W|$, number unique keywords in all documents; $|\mathsf{DB}(w)|$ for every w searched; $\sum_{w \in W} |\mathsf{DB}(w)|$ i.e. the number of (w, d) pairs where w is in d. See Appendix A.4 for the proof.

Theorem 2. *The above SSE scheme is $\mathcal{L}$-secure (cf. definition of section 2.4), if* TWORAM *is secure (cf. definition of section 2.2), F is a PRF, and the encryption used in the one-level path ORAM is CPA-secure.*

Efficiency. The setup cost for our SSE shceme is the sum of the setup cost for TWORAM for a memory of size $|W|$, and the setup for a one-level path ORAM of size $n = \sum_{w \in W} |\mathsf{DB}(w)|$ which is $O(n \log n \operatorname{loglog} n)$.

The bandwidth cost for each search/add query w is the cost of one ORAM read in TWORAMplus $O(|\mathsf{DB}(w)| * (\log n \operatorname{loglog} n))$ for $n = \sum_{w \in W} |\mathsf{DB}(w)|$.

Acknowledgments

This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.

Sanjam Garg was supported in part from a DARPA/ARL SAFEWARE award, AFOSR Award FA9550-15-1-0274, and NSF CRII Award 1464397.

Charalampos Papamanthou was supported in part by NSF grants #1514261 and #1526950, by a NIST award, by a Google Faculty Research Award and by Yahoo! Labs through the Faculty Research Engagement Program (FREP).

The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

References

1. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 784–796, Raleigh, NC, USA, October 16–18, 2012. ACM Press.
2. David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.
3. Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.
4. M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

5. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
6. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.
7. Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious ram. In *TCC*, 2016.
8. Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. Cryptology ePrint Archive, Report 2015/307, 2015. http://eprint.iacr.org/2015/307.
9. Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *47th Annual ACM Symposium on Theory of Computing*, pages 449–458. ACM Press, 2015.
10. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422, Copenhagen, Denmark, May 11–15, 2014. Springer, Berlin, Germany.
11. Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.
12. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194, New York City,, New York, USA, May 25–27, 1987. ACM Press.
13. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
14. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 576–587, 2011.
15. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 157–167, 2012.
16. Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography (FC)*, 2013.
17. Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976, 2012.
18. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, 2012.
19. Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.
20. Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, 2013.
21. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734, Athens, Greece, May 26–30, 2013. Springer, Berlin, Germany.
22. Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 862–873, 2015.

23. Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd Annual ACM Symposium on Theory of Computing*, pages 514–523, Baltimore, Maryland, USA, May 14–16, 1990. ACM Press.
24. Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *TCC*, pages 457–473, 2009.
25. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 197–214, 2011.
26. Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55, 2000.
27. Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
28. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.
29. Peter van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management*, pages 87–100, 2010.
30. Xiao Shaun Wang, T.-H. Hubert Chan, , and Elaine Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202, 2015.
31. Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 293–304, 2012.
32. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

A More Details on Path ORAM

A.1 Path ORAM abstraction algorithms

A.2 Path ORAM protocols with $\log n$ rounds of interaction using the abstraction

A.3 Proof of security for TWORAM

Now we prove TWORAM is a secure realization of an oblivious RAM scheme as described in Section 2.2. We start by arguing correctness. Note that the garbled circuits implement the exact same procedures as are required in our abstraction. Therefore the correctness of our scheme follows directly from the correctness of the underlying Path ORAM scheme and garbled circuits construction. Next we argue security. In other words we need to argue that for any adversary A, there exists a simulator Sim for which the following two distributions are computationally indistinguishable.

Algorithm 1 Setting up path ORAM data structures.

procedure $\mathcal{T} \leftarrow$ INITIALIZE($1^\kappa, A_L$)
  Store block $(x, A_L[x], \pi_L(x))$ at leaf $\pi_L(x)$ of tree T_L, where π_L is a n-permutation;
  for $i = L$ down to 3 **do**
    Let A_{i-1} be an array of 2^{i-1} entries such that $A_{i-1}[x] = \pi_i(2x-1)||\pi_i(2x)$ for $x = 1, \ldots, 2^{i-1}$;
    Store block $(x, A_{i-1}[x], \pi_{i-1}(x))$ at leaf $\pi_{i-1}(x)$ of tree T_{i-1}, where π_{i-1} is a 2^{i-1}-permutation;
  end for
  Let A_1 be an array of 2 entries such that $A_1[x] = \pi_2(2x-1)||\pi_2(2x)$ for $x = 1, 2$;
  return $\{A_1, T_2, \ldots, T_L\}$;
end procedure

Algorithm 2 Extraction algorithm for buckets.

procedure $\pi \leftarrow$EXTRACTBUCKET(i, y, b)
  Search bucket b to retrieve block $(y_i, A_i[y_i], p)$;
  if found **then**
    return $\pi \leftarrow \mathsf{select}(A_i[y_i], b_i)$; ▷ π is the index of the path to be explored in T_{i+1}.
  else
    return $\perp$;
  end if
end procedure

Algorithm 3 Update algorithm. It takes as input $L-1$ paths and local storage A_1, then assigns random indices to the paths that were read before and performs the eviction, outputting new paths.

procedure $\{A_1, T_2(x_2), \ldots, T_L(x_L)\} \leftarrow$UPDATE($y, val, A_1, T_2(x_2), \ldots, T_L(x_L)$)
  $\mathsf{select}(A_1[y_1], b_1) \leftarrow r_2$; ▷ r_i is random in $[1, 2^{i+1}]$.
  for $i = 2$ to L **do**
    $T_i.\mathsf{root} \leftarrow T_i.\mathsf{root} \cup \mathsf{readPath}(T_i(x_i))$; ▷ $T_i.\mathsf{root}$ serves as the stash C_i.
    Update block $(y_i, A_i[y_i], x_i)$ to $(y_i, A_i[y_i], r_i)$ in $T_i.\mathsf{root}$;
    $\mathsf{select}(A_i[y_i], b_i) \leftarrow r_{i+1}$; ▷ if $i = L$ **if** $val \neq \mathsf{null}$ then set $A_L[y] \leftarrow val$ **else** do not change.
    $[T_i.\mathsf{root}, T_i(x_i)] \leftarrow \mathsf{evictPath}(T_i.\mathsf{root})$;
  end for
  return $A_1, T_2(x_2), T_3(x_3), \ldots, T_L(x_L)$;
end procedure

Protocol $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow$ SETUP$\langle (1^\kappa, \mathsf{M}), \perp \rangle$:

Client:

1: Pick a κ-bit s; Run $\{A_1, T_2, \ldots, T_L\} \leftarrow$ INITIALIZE$(1^\kappa, \mathsf{M})$;
2: For all $i > 1$, for all $u \in T_i$, set $\mathsf{B}_u \leftarrow \mathsf{Enc}_s(\mathsf{bucket}_u)$, where $\mathsf{Enc}_s(.)$ is a CPA-secure encryption;
3: For all $i > 1$, for all $u \in T_i$, send to server data B_u;
4: **return** s and A_1 as σ;

Server:

1: **return** all data B_u sent by the client from above as EM;

Fig. 6: Formal description of the SETUP protocol for the interactive ORAM.

Protocol $\langle (\mathsf{M}[y], \sigma'), \mathsf{EM}' \rangle \leftrightarrow$ OBLIVIOUSACCESS$\langle (\sigma, y, val), \mathsf{EM} \rangle$:

Client(1)**:**

1: Compute $x_2 \leftarrow \mathsf{select}(A_1[y_1], b_1)$. Send to server index x_2; ▷ run **Server**(2)

Server(i)**:**

1: For all $u \in T_i(x_i)$ send to client B_u; ▷ run **Client**(i)

Client(i)**:**

1: $\pi = \perp$;
2: **for** $u \in T_i(x_i)$ **do**
3: $\quad \mathsf{bucket}_u \leftarrow \mathsf{Dec}_s(\mathsf{B}_u)$;
4: $\quad$ **if** $\pi = \perp$ **then**
5: $\quad\quad \pi \leftarrow$ EXTRACTBUCKET$(i, y, \mathsf{bucket}_u)$;
6: $\quad\quad x_{i+1} = \pi$;
7: $\quad$ **end if**
8: **end for**
9: **if** $i < L$ **then**
10: $\quad$ Send to server new index x_{i+1}; ▷ run **Server**$(i+1)$
11: **else**
12: $\quad \{A_1, T_2(x_2), \ldots, T_L(x_L)\} \leftarrow$ UPDATE$(y, val, A_1, T_2(x_2), \ldots, T_L(x_L))$;
13: $\quad$ For all $i > 1$, for all $u \in T_i(x_i)$, send to server $\mathsf{B}_u \leftarrow \mathsf{Enc}_s(\mathsf{bucket}_u)$; ▷ run **Server**$(L)$
14: $\quad$ **return** x_{L+1} as $\mathsf{M}[y]$ and A_1 as σ';
15: **end if**

Server(L)**:**

1: **return** the data received by the client as EM';

Fig. 7: Formal description of the OBLVIOUSACCESS protocol for the interactive ORAM.

– $\mathrm{Real}_A^{\Pi}(\kappa)$: A chooses M. The experiment then runs $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\kappa, \mathsf{M}), \perp \rangle$. A then provides read/write queries (y_i, v) where $v = \mathsf{null}$ on reads, for which the experiment runs the protocol $\langle (\mathsf{M}[y_i], \sigma_i), \mathsf{EM}_i \rangle \leftrightarrow \textsc{ObliviousAccess}\langle (\sigma_{i-1}, y_i, v), \mathsf{EM}_{i-1} \rangle$, Denote the full transcript of the protocol by t_i. Eventually, the experiment outputs $(\mathsf{EM}, t_1, \ldots, t_q)$ where q is the total number of read/write queries.
– $\mathrm{Ideal}_{\mathsf{Sim}}^{\Pi}(\kappa)$: The experiment outputs $(\mathsf{EM}, t'_1, \ldots, t'_q) \leftarrow \mathsf{Sim}(q, |\mathsf{M}|, 1^\kappa)$.

Our Simulator. Note that the simulator needs to provide to the server, for all u $\tilde{\mathsf{C}}^u, \mathsf{X}^u$ and for all $i \in \{2, \ldots, L\}$ $\beta_i := \mathsf{nState}^{(i,0,0)}_{\mathsf{cState}^{(i+1,0,0)}}$. Furthermore replacement circuits need to be provided as read/write queries are implemented. Our simulator Sim generates these as follows:

– For each $u = (i, j, k)$, let $(\tilde{\mathsf{C}}^u, \mathsf{lab}^u \leftarrow \mathsf{GCircuit}(1^\kappa, \mathsf{P}[u, b_u, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)}])$, where b_u is random bit and P is a circuit that, if $j = i$ outputs $(\mathsf{nextRoot}, \mathsf{lab}_{\mathbf{0}}^{(i+1,0,0)})$, else if $b = 0$ then it outputs $(\mathsf{left}, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)})$ and $(\mathsf{right}, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)})$ otherwise.
– Each X^u is generated as as encryption of a zero-string, namely $\mathsf{Enc}_s(\mathbf{0})$. Similarly for all $i \in \{2, \ldots, L\}$ $\beta_i := \mathsf{nState}_{\mathbf{0}}^{(i,0,0)}$.

Note that as the provided garbled circuits are executed, replacements circuits need to be given and they are generated in the same manner as above.

Proof of Indistinguishability. The proof follows by a hybrid argument.

- H_0: This hybrid corresponds to the honest execution $\mathrm{Real}_A^{\Pi}(\kappa)$ as done honestly.
- H_1: This hybrid is same as H_0 except that we now generate all the X^u values as encryptions of zero-strings of appropriate length. Specifically, for each u we set $\mathsf{X}^u \leftarrow \mathsf{Enc}_s(\mathbf{0})$.
  The indistinguishability between H_0 and H_1 follows from the security of the encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$.
- H_2: In this hybrid the simulator maintains the entire Path ORAM tree internally but does not include it in the provided garbled circuits. In other words garbled circuits are generated as follows:
  - For each $u = (i, j, k)$, let $(\tilde{\mathsf{C}}^u, \mathsf{lab}^u \leftarrow \mathsf{GCircuit}(1^\kappa, \mathsf{P}[u, b_u, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)}])$, where b_u is 0 or 1 depending on whether the execution as per ORAM would go left or right and P is a circuit that, if $j = i$ outputs $(\mathsf{nextRoot}, \mathsf{lab}_{\mathbf{0}}^{(i+1,0,0)})$, else if $b = 0$ then it outputs $(\mathsf{left}, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)})$ and $(\mathsf{right}, \mathsf{lab}_{\mathbf{0}}^{(i,j+1,2k+b)})$ otherwise.
  - Each X^u is generated as as encryption of a zero-string, namely $\mathsf{Enc}_s(\mathbf{0})$. Similarly for all $i \in \{2, \ldots, L\}$ $\beta_i := \mathsf{nState}_{\mathbf{0}}^{(i,0,0)}$.

  The indistinguishability between H_1 and H_2 follows by a sequence of hybrids where each garbled circuit is replaced by a simulated garbled circuit. Here these hybrids must be performed in sequence in which garbled circuits are consumed. Note that for the unconsumed garbled circuits the input labels aren't provided (or hardcoded inside any other circuit) and hence they can also be simulated.

– H_3: Same as H_2, except that each b_u is now chosen uniformly random, independent of the Path ORAM execution. Note that this is same as the simulator.
The indistinguishability between H_2 and H_3 from the security of the Path ORAM scheme.

This concludes the proof. □

A.4 Proof of Security for the SSE Scheme

We prove Theorem 2 on security of the SSE scheme next, Following the definition of section 2.4, we first describe a simulator Sim who generates the transcripts for the ideal distribution $\mathrm{Ideal}^{\Pi}_{A,\mathsf{Sim},\mathcal{L}}(\kappa)$. Sim takes as input $\mathcal{L}(\mathsf{DB}, H)$, and does the following: To generate full transcripts of the constant round ORAM scheme for the adversary A, Sim runs Sim', the simulator that exists for that scheme due its security (see definition of section 2.2). That is, he runs $(\mathsf{EM}, t_1, \ldots, t_q) \leftarrow \mathsf{Sim}'(q, |\mathsf{M}|, 1^\kappa)$, where he drives $|\mathsf{M}|$ from $|W|$. To simulate the transcripts of the path-ORAM component, it generates a one-level path ORAM tree T_L for a memory array of size $\sum_{w \in W} |\mathsf{DB}(w)|$ filled with all 0 values. For each read/add query, it replaces the PRF-genenerated paths by uniformly random paths, and generates freshly generated ciphertexts of 0 for updated paths. Sim knows the number of paths to retrieve/update for each query from the lekage function which outputs $|\mathsf{DB}(w)|$ for every query w. This completes the description of the simulator. We now need to show that $\mathrm{Ideal}^{\Pi}_{A,\mathsf{Sim},\mathcal{L}}(\kappa)$ is indinstinguishable from $\mathrm{Real}^{\Pi}_{A}(\kappa)$, which constitutes the first in the sequence of our Hybrids:

Proof of Indistinguishability. The proof follows by a hybrid argument.

- H_0: This hybrid corresponds to the honest execution $\mathrm{Real}^{\Pi}_{A}(\kappa)$ for the SSE scheme which we repeat here for completeness. A chooses DB. The experiment then runs $\langle \mathsf{EDB}, \sigma \rangle \leftrightarrow \text{SSESETUP}\langle (1^\kappa, \mathsf{DB}), \perp \rangle$. A then adaptively makes search queries w_i, which the experiment answers by running the protocol $\langle \mathsf{DB}_{i-1}(w_i), \sigma_i \rangle \leftrightarrow \text{SSESEARCH}\langle (\sigma_{i-1}, w_i), \mathsf{EDB}_{i-1} \rangle$. Denote the full transcripts of the protocol by t_i. Add queries are handled in a similar way. Eventually, the experiment outputs $(\mathsf{EDB}, t_1, \ldots, t_q)$ where q is the total number of search/add queries made by A.
- H_1:Similar to H_0, except that the portions of t_i's corresponding to the constant-round ORAM are instead generated by $\mathsf{Sim}'(q, |\mathsf{M}|, 1^\kappa)$ where Sim' is the simulator in the proof of the ORAM scheme.
The indistinguishability of H_0 and H_1 follows from security of the ORAM scheme.
- H_2: Similar to H_1 except that all ciphertexts in the path ORAM tree are replaced by encryptions of 0, and all updated ciphertexts will be fresh encryption of 0.
The indistinguihsability of H_2 and H_1 follows from the semantic security of the encryption scheme used in the path ORAM.
- H_3: Similar to H_2 except that all PRF-generated positions are replaced by uniformly random positions. Note that H_3 is essentially $\mathrm{Ideal}^{\Pi}_{A,\mathsf{Sim},\mathcal{L}}(\kappa)$.
The indistinguihsability of H_3 and H_2 follows from the pseudorandomness of the the PRF.

This concludes the proof □.