Abstract—We study the problem of dynamic searchable encryption (DSE) with forward-and-backward privacy. Many DSE schemes have been proposed recently but the most efficient ones have one limitation: they require maintaining one operation counter for each unique keyword, either stored locally at the client or stored encrypted at the server and accessed obliviously (e.g., with an oblivious map) during every operation. We propose three new schemes that overcome the above limitation and achieve constant permanent client storage with improved performance, both asymptotically and experimentally, compared to prior state-of-the-art works. In particular, our first two schemes adopt a “static-to-dynamic” transformation which eliminates the need for oblivious accesses during searches. Due to this, they are the first practical schemes with minimal client storage and non-interactive search. Our third scheme is the first quasi-optimal forward-and-backward DSE scheme with only a logarithmic overhead for retrieving the query result (independently of previous deletions). While it does require an oblivious access during search in order to keep permanent client storage minimal, its practical performance is up to four orders of magnitude better than the best existing scheme with quasi-optimal search.

I. INTRODUCTION

With the advent of data outsourcing and the increasing awareness for user data privacy, the ability to compute on encrypted data stored on a remote untrusted server has emerged as a necessary tool. A fundamental task in this area is searching in encrypted datasets. E.g., assuming a collection of encrypted documents, a query may be to return all the identifiers of the documents containing the keyword \( w \), without first decrypting the documents. In the literature, this is known as searchable encryption (SE) and it has found applications such as building encrypted email [42] and encrypted image storage in the cloud [2] with search capabilities. Since its first introduction by Song et al. [51], almost twenty years ago, SE has been a topic of significant study, e.g., in order to improve its efficiency aspects, or to accommodate more advanced functionalities, such as boolean queries [13], [34], sub-string, wild-card and phrase queries [26], and a variety of database queries, such as point [23], [24], range [22], [21], [26] and more general SQL queries [35] (e.g., join and group-by queries).

In order to achieve good practical performance, it has become commonplace in the literature to allow SE schemes to reveal some information about the dataset to the server, referred to as leakage. This leakage can occur at an initialization phase (e.g., the dataset size) or during the execution of a query (e.g., access and search pattern, the identifiers of the documents containing \( w \) and when this search was performed previously, respectively).

**Dynamic SE.** Recent research has focused on dynamic searchable encryption (DSE) schemes that can efficiently support modifications in the encrypted dataset, without the need to re-initialize the protocol. From a security perspective, developing secure DSE schemes is challenging due to the additional information that may be revealed to the server because of updates. Two relevant security notions have been proposed for dynamic SE schemes—forward and backward privacy. Forward privacy [15], [52] ensures that a new update cannot be related to previous operations (until the related keyword is searched). Besides the obvious benefit of allowing the encrypted dataset to be built “on-the-fly” (crucial for certain applications, e.g., encrypted e-mail storage starting from a new mailbox), forward privacy is essential for mitigating certain leakage-abuse attacks that depend on adversarial file injection [56].

On the other hand, backward privacy ensures that if a document containing keyword \( w \) is deleted before a search for \( w \), the result of this search does not reveal anything about this document. Backward privacy is much less studied than forward privacy. It was first proposed in NDSS 2014 by Stefanov et al. [52] and formally defined recently in CCS 2017 by Bost et al. [10] who proposed three types of backward-privacy. During a search, BP-I reveals only the identifiers of files currently containing \( w \) and when they were stored, BP-II additionally reveals the timestamps and types (insertion/deletion) of all prior updates for \( w \), and BP-III additionally reveals for each prior deletion which insertion it canceled.

**Challenge 1: DSE with small client storage.** The majority of practical DSE constructions from the literature (e.g., [9], [10], [29], [25]) require the client to locally store a table that holds for every keyword in the dataset a counter \( a_w \) that counts the number of updates for \( w \) (some schemes store an additional counter for searches). This allows for very efficient schemes in practice, e.g., insertion of the entry \((w, id, add)\) after \( a_w \) updates can be done by encrypting \((w, id)\) and placing the ciphertext in a hash map (stored at the server) at position \( F(k, (w, a_w + 1)) \), where \( F \) is a pseudorandom

Network and Distributed Systems Security (NDSS) Symposium 2020
23–26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24423
www.ndss-symposium.org
function (updates also can contain deletes which are handled by inserting cancelation tuples). Later, to search for \( w \) the client simply looks up the value of \( a_w \) and queries the map at locations \( F(k, \langle w, 1 \rangle), \ldots, F(k, \langle w, a_w \rangle) \).

With some variations, this is the basic blueprint of many existing schemes. This local word counter gives very efficient schemes but it has an obvious drawback: increased client storage. Compared to storing an inverted index for \( DB \) locally, the client needs to store a table \( W \) of unique keywords which, depending on the dataset, may be rather large. E.g., for the Enron e-mail dataset, \( |DB| \approx 2.6\text{M} \) and \( |W| \approx 77\text{K} \), i.e., the client has to go through the trouble of deploying a DSE (and leaking information) just to reduce its local storage by \( 33\times \). When using SE to store relational database records (e.g., [35], [24], [23]) the savings can be significantly smaller, i.e., in the case of a real dataset with crime incidents in Chicago [1] (used in [24], [23]) with \( |DB| \approx 6\text{M} \) tuples, 22 attributes, and \( |W| \approx 17\text{M} \), the reduction in local storage for supporting point queries for these attributes will be \( 5\times \) at best (similar results are observed in TPC-H benchmark [3]). In general, for relational database search many attributes may contain unique values, (e.g., every record may contain a different value) and in these cases the improvement in local storage will be negligible. The aforementioned examples clearly illustrate that in many cases storing locally a counter per word is problematic. Moreover, if we would like to support the capability to access the encrypted database from multiple devices, this approach would be especially cumbersome as it entails synchronization and state transfer among them.

Using oblivious primitives. To avoid this, previous works (e.g., [12], [25], [9], [10]) have proposed to store \( W \) at the server encrypted. This would trivially violate forward privacy, unless one uses an oblivious map (OMAP) [55] that hides from the server which word entry is accessed every time. One downside of this is that the construction of [55] and subsequent improvements [48] require a logarithmic number of rounds of interaction. The only existing DSE that avoids this is the forward-private scheme of [27] (later made backward private in [10]). However, it uses the recursive Path-ORAM construction of [53] and it relies on heavy garbled circuit computation to make it non-interactive. Therefore, its potential for adoption in practice is quite limited and it serves mostly as a feasibility result.\(^1\) Hence, we ask whether it is possible to design a practical backward-and-forward-private DSE with small client storage. \( N \) is an upper bound for total insertions, \( |W| = \# \text{ distinct keywords} \). For keyword \( w \): \( a_w = \# \text{ updates} \), \( i_w = \# \text{ insertions} \), \( d_w = \# \text{ deletions} \), \( n_w = \# \text{ files containing } w \). RT is \# roundtrips for retrieving \( DB(w) \). BP stands for backward privacy (the smaller, the better) and am. for amortized efficiency. The \( O \) notation hides polylogarithmic factors. WO stands for storing search/insertion counters for each \( w \) at an oblivious map. To minimize client storage, oblivious map stashes are stored at the server and downloaded every time.

\(^1\)Alternatively, this can be achieved with the use of trusted hardware [5].

**TABLE I:** Comparison of existing forward-and-backward-private DSE with small client storage. \( N \) is an upper bound for total insertions, \( |W| = \# \text{ distinct keywords} \). For keyword \( w \): \( a_w = \# \text{ updates} \), \( i_w = \# \text{ insertions} \), \( d_w = \# \text{ deletions} \), \( n_w = \# \text{ files containing } w \). RT is \# roundtrips for retrieving \( DB(w) \). BP stands for backward privacy (the smaller, the better) and am. for amortized efficiency. The \( O \) notation hides polylogarithmic factors. WO stands for storing search/insertion counters for each \( w \) at an oblivious map. To minimize client storage, oblivious map stashes are stored at the server and downloaded every time.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Computation</th>
<th>Update</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MONETA [10]</td>
<td>( O(a_w \log N + \log^3 N) )</td>
<td>( O(\log^2 N) )</td>
<td>( O(a_w \log N + \log^3 N) )</td>
</tr>
<tr>
<td>WO+MITRA [29]</td>
<td>( O(a_w + \log^2</td>
<td>W</td>
<td>) )</td>
</tr>
<tr>
<td>SD(_1)</td>
<td>( O(a_w + \log N) )</td>
<td>( O(\log N) )</td>
<td>( a_w + \log N )</td>
</tr>
<tr>
<td>SD(_2)</td>
<td>( O(a_w + \log N) )</td>
<td>( O(\log N) )</td>
<td>( a_w + \log N )</td>
</tr>
<tr>
<td>ORION [29]</td>
<td>( O(n_w \log N) )</td>
<td>( O(\log N) )</td>
<td>( n_w \log N )</td>
</tr>
<tr>
<td>ROSES [30]</td>
<td>( O(n_w \log N) )</td>
<td>( O(\log N) )</td>
<td>( n_w \log N )</td>
</tr>
<tr>
<td>GOS</td>
<td>( O(n_w \log^2 N + \log^2</td>
<td>W</td>
<td>) )</td>
</tr>
</tbody>
</table>

Challenge 2: DSE with (quasi-)optimal search.\(^2\) With a plaintext dataset, the \( n_w \) document identifiers of files currently containing \( w \) can be optimally retrieved with \( n_w \) operations. The same performance can be achieved for DSE (e.g., [9], [25]), albeit for insertion-only schemes (where \( n_w = a_w \), the total number of updates for \( w \)). With deletion-supporting DSE \( n_w \) can be arbitrarily smaller than \( a_w \). The only two backward-private schemes that come close to achieving this optimal performance are from [29]. At a high-level, they replace the \( n_w \) accesses necessary for retrieving the result with oblivious accesses and achieve a polylogarithmic overhead over the optimal cost (see Table I for more details). According to Definition 4, these schemes achieve quasi-optimal search time. However, their "black-box" use of oblivious primitives results in schemes with rather poor performance, especially due to communication cost (e.g., [29] reports \( \approx 1\text{MB} \) communication for returning just \( n_w = 100 \) identifiers). Therefore, we aim to develop a DSE with quasi-optimal search and much better practical performance—our QOS scheme (see Table I).

A. Our results

In this work, we present novel schemes that address the above challenges as follows:

(i) We present a black-box reduction from any result-hiding static SE to a backward-and-forward private DSE. We instantiate it with [12] and call the resulting scheme SD\(_w\). It has \( O(a_w + \log N) \) search cost, where \( a_w \) denotes the total number of updates for keyword \( w \), and \( O(\log N) \) amortized update cost. Most importantly, SD\(_w\) is the first DSE with \( O(1) \) permanent client storage without using oblivious primitives, hence it greatly outperforms all existing schemes for searches.

(ii) During amortized updates the temporary client storage of SD\(_w\) may grow arbitrarily large (up to \( O(N) \)). To avoid this, we present a version with de-amortized

\(^2\)As per Definition 4, a DSE scheme has optimal search time, if the asymptotic complexity of search is \( O(n_w) \) and quasi-optimal if search time is \( O(n_w + \text{polylog}(N)) \).
updates called SD$_a$ that has the same search overhead as SD$_d$ and it outperforms state-of-the-art low-client-storage DSE schemes in many scenarios (see our experimental evaluation in Section V).

(iii) Finally for delete-intensive query workloads, we present QOS, a DSE with quasi-optimal search time $O(n_w \log i_w)$ and $O(1)$ client storage that vastly outperforms existing quasi-optimal schemes during searches, where $n_w$ denotes the number of files containing keyword $w$ and $i_w$ the number of insertions for $w$. Indeed, for large deletion percentages (approximately $40-80\%$, depending on the deployment setting) it outperforms all other schemes.

All our constructions are forward-and-backward private (BP-II for SD$_a$ and SD$_d$, BP-III for QOS). Formal theorem statements and proofs can be found in Appendix A. Our schemes are secure in the programmable random oracle model but this assumption can be removed with standard techniques without decrease in asymptotic efficiency, similar to [12], [25].

A detailed comparison with other DSE can be seen in Table I where we only focus on schemes with small client storage. We also consider WO$+\text{MITRA}$ (WO stands for storing search/insertion counters for each $w$ at an oblivious map), the result of combining the most efficient backward-private scheme from [29] with the “word counter + oblivious map” approach described above (this technique can be used with other schemes, e.g., FIDES, JANUS from [10] and JANUS++ from [54], but MITRA outperforms all of them both in terms of performance and security). All schemes in Table I use OMAPs, except for SD$_a$; they can achieve $O(1)$ storage by storing the stashes at the server and generating keys with a PRF. One general conclusion from the table is that our schemes achieve much better search performance at the cost of increased overhead for updates. We note that this trade-off can be favorable, e.g., it seems suitable for OLAP databases and data warehouses [18] in which search is more crucial than the update performance.

Overview of techniques. SD$_a$ and SD$_d$ utilize classic techniques for transforming static data structures to dynamic ones [45], [46]. They store the result of $N$ updates in a sequence of $\log N$ separate indexes with sizes $2^0, \ldots, 2^{\log N}$. Each update is first stored in the smallest index and whenever two indexes of the same size exist they are downloaded and merged to a single new one by the client. Searches are then executed at each data structure independently. Ensuring that the underlying static scheme is result-hiding allows us to prove that SD$_a$ is backward private. Similar to the majority of DSE constructions from the literature, we store triplets of the form $(w, id, op)$ where $op = \text{add}/\text{del}$, that is, deletions are also “stored”. During search, the client filters out deleted entries. 

Private lazy rebuild. For SD$_d$, we propose a way to implement the lazy rebuild de-amortization technique of [46] for searchable encryption. At a high level, a fixed small number of steps of the setup routine of the underlying static SE need to be run for each update. In order to achieve a backward-and-forward private DSE scheme, the static SE must have a setup that is naturally “decomposable” into steps with small, efficiently retrievable local state. For our instantiation, we choose PiBas, an efficient static scheme from [12]. During setup, PiBas parses the input dataset in a deterministic manner, one keyword-document identifier pair at a time, storing the corresponding entry $(w, id)$ in a pseudorandom position at a hash table. The only state for the client is a counter of the times each keyword $w$ has been inserted so far (Figure 12). In SD$_d$, we store this state at the server using an OMAP. For every update, the client performs one step of the PiBas setup process for every data structure size in the worst-case. For each of these, it needs the counter for the related keyword which is retrieved with an OMAP access. Due to the deterministic access pattern of PiBas and the oblivious accesses, the resulting scheme is forward-and-backward private.

OMAP for updates; not for searches. A very interesting aspect of SD$_d$ is that, although it uses an oblivious map, it is only accessed during updates. Due to this, searches are executed non-interactively.

QOS: Practical quasi-optimal search. Unlike all previous quasi-optimal search DSE that need to perform $n_w$ oblivious accesses, QOS requires a single OMAP query which makes it much faster in practice. This is achieved by decoupling the result retrieval from the oblivious primitives.

QOS stores inserted entries of the form $(w, id)$ in a hash table $T$ where the position is computed based on a pseudorandom token, similar to most existing DSE schemes. Where it deviates is in the way it treats deletions. For each $w$, the client maintains a “conceptual” binary tree that describes its update history (see Figure 5). It has $N$ leafs, where $N$ is an upper bound in the number of supported insertions; initially all nodes are white. Every node is numbered naturally, starting from leaves $1, \ldots, N$, on to their parents $N+1, \ldots, 3N/2+1$, etc. Inserted entries are “mapped” to the leafs of the tree from left to right. In practice, this is done by storing the $i$-th insertion $(w_i, id_i)$ to a position in $T$ that depends on the leaf label $i$. Deletions mark tree nodes as black, e.g., deleting $(w_1, id_1)$ would color the $i$-th leaf black. Black node then “propagate” upwards: if two siblings are black, their parent is also colored black. After every deletion, let $j$ be the single top-most node that was just colored black. The client marks a position (computed pseudorandomly based on $w, j$ and the search counter of $w$) in a separate hash table $D$. To hide the tree manipulation part (coloring the nodes), and to be able to efficiently retrieve the leaf where $(w_i, id_i)$ is stored, QOS involves two additional OMAPs, in addition to the “standard one” for retrieving search and insertion counters for $w$.

During a search for $w$, after retrieving the search counter from the OMAP, the client releases the PRF tokens. The server uses them to non-interactively find the positions in $T, D$ for the nodes of the update tree for $w$. To speed up the process, it starts from the Best Range Cover (selects the minimum number of nodes that cover exactly the range) of leafs $[1, i_w]$ and proceeds downwards, always checking whether it found a black node by looking up positions of $D$, in which case it abandons this path. Upon reaching non-black leafs, the server looks up their positions in $T$; by construction, this is where non-deleted entries are found. After decrypting and retrieving the result, the client “re-maps” the accessed entries in $T, D$ to new locations using freshly computed PRF tokens with increased search counter.
Clean-up. As discussed above, many DSE schemes, including SD$_a$ and SD$_d$, store deletions as actual entries. Thus, the server storage is not reduced after deletions. However, this is almost unavoidable when storing the encrypted entries in a hash table/map (where memory is not de-allocated). One notable exception is the forward-private DSE of [39] but no backward-private scheme with this property exists. On the other hand, reducing search time for future searches is—arguably—more important than saving storage space at the server. Many existing DSE schemes have a special “clean-up” phase for this, typically executed in tandem with searches. Our SD$_a$ and SD$_d$ schemes are amenable to such a clean-up process taking place during updates (the first very naturally whereas the second requires some additional bookkeeping). On the other hand, quasi-optimal schemes like QOS inherently achieve this since searches are (almost) unaffected by deletions.

Experimental evaluation. We implemented our three schemes and compare their search, update, and storage performance with existing forward-and-backward private DSE (Section V). In particular, we compare them with the best low-client-storage scheme, M$\text{ITRA}$ [29] with the word counter stored in an oblivious map, and HORUS [29], the faster quasi-optimal scheme. In terms of search time, SD$_a$ and SD$_d$ take less than 0.1ms for retrieving a result of 100 elements from a dataset of 1M records. Moreover, for small results, they are up to $34 \times$ and $20 \times$ faster than M$\text{ITRA}$, with the added benefit of being non-interactive. Turning to quasi-optimal schemes, QOS takes 1.3ms for the same setting, vastly outperforming HORUS (4-16531x throughout our experiments). Where our schemes perform worse is in updates (as is evident from the asymptotic analysis in Table I), e.g., for our tested cases QOS is roughly $2 \times$ slower than HORUS (with the same blowup factor for communication size), whereas M$\text{ITRA}$ is up to $21 \times$ faster than SD$_d$ (in the worst case). All these results are for 10% deleted entries. For larger delete percentages we show that QOS has the potential to become the most efficient solution. It outperforms both M$\text{ITRA}$ and SD$_d$ after different ratios between 40-80%, depending on the number of insertions.

B. Related Work

Searchable encryption—considered a special type of structured encryption [17]—was introduced by Song et al. [51]. Curtmola et al. [19] proposed the most widely used security definition and the first scheme with non-trivial search time. It has since been improved in several ways, e.g., support for multiple users [49], [50], [33], more expressive queries including relational databases, conjunctive keywords, and graph queries [13], [34], [17], [41], [35], [22], [40], or efficient on-disk storage [14], [43], [7], [23], [20].

The first DSE schemes were presented in [37], [36]; these schemes achieve optimal search time at the expense of increased leakage (none of these schemes are forward or backward private). The notion of forward privacy was first discussed in [16] and improved in multiple subsequent works (e.g., [52], [32], [12], [44], [9], [39], [27], [25]). Backward privacy was first considered by Stefanov et al. [52] and formally defined much later by Bost et al [10]. Since then, Ghareh Chamani et al. [29], and Sun et al. [54] presented more efficient schemes, and Amjad et al. [5] proposed a scheme using trusted hardware. The use of classic “static-to-dynamic” data structure techniques for DSE has been proposed before, e.g., [22], [23]. However, these works only consider forward privacy and amortized solutions—our scheme SD$_d$ is the first to achieve backward privacy and worst-case low-storage updates. A forward-private DSE with quasi-optimal search was first proposed in [52]. The property was defined in [29] that also presented quasi-optimal DSE with forward and backward privacy. A general performance comparison of our schemes with previous low-storage DSE is shown in Table I.

Our QOS scheme organizes updates in a conceptual binary tree and uses each node’s natural label to compute its corresponding storage position via a PRF. We believe a very similar scheme can be achieved using the classic tree-based GGM PRF/DPRF [31], [38], [8], [11], mapping leafs to its outputs and nodes higher in the tree to “intermediate” evaluations. This construction would avoid the random oracle assumption (without additional interaction), however we instead chose to build QOS in a black-box way from any PRF.

Recent advances in ORAM. Recently, [47], [6] proposed novel ORAM constructions, with improved efficiency matching the theoretical optimal overhead of $O(\log N)$. It is possible to modify these to yield oblivious maps, thus asymptotically improving our schemes SD$_d$ and QOS as well as [27], [10], [29]. However, they achieve amortized performance. While it may be possible to de-amortize them, this will undoubtedly result in additional cost.$^3$

II. Preliminaries

We denote by $\lambda \in \mathbb{N}$ a security parameter. PPT stands for probabilistic polynomial-time. Our protocols are executed between two parties, a client and a server. Slightly abusing notation, we let $(x'; y') \leftrightarrow P(x; y)$ denote a (possibly multi-round) protocol execution with input $x$ and output $x'$ for the client, and input $y$ and output $y'$ for the server. We consider a collection of $D$ documents with identifiers $id_1, \ldots, id_D$, each of which contains textual keywords from a given alphabet $\Sigma$. Let the dataset $DB$ consist of pairs of keyword-file identifiers, such that $(w, id) \in DB$ if and only if the file $id$ contains keyword $w$. For each $w$, let $DB(w)$ denote the set of documents that contain keyword $w$. Let $W$ denote a set of keywords that contains all the keywords from $DB$ (possibly more).

Pseudorandom functions. Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a pseudorandom function (PRF) family. $F$ is a secure PRF family if for all PPT adversaries $Adv_{\lambda} [K \leftarrow Gen(1^\lambda); Adv^{F(K)}(1^\lambda) = 1] - Pr[Adv^{F(R)}(1^\lambda) = 1] \leq \nu(\lambda)$, where $R : \{0,1\}^\ell \to \{0,1\}^\ell$ is a truly random function.

Searchable encryption. A dynamic symmetric searchable encryption scheme (DSE) $\Sigma = (\text{Setup}, \text{Search}, \text{Update})$ consists of algorithm $\text{Setup}$, and protocols $\text{Search}, \text{Update}$ that are executed between a client and a server:

- $\text{Setup}(\lambda)$ on input $\lambda$ outputs $(K, \sigma, EDB)$ where $K$ is a secret key for the client, $\sigma$ is the client’s local state,
and EDB is an (initially empty) encrypted database that is sent to the server. The notation Setup(λ, N) refers to a setup process that takes a parameter N for the maximum supported number of entries.

- Search(K, q, σ; EDB) is a protocol for searching the database. Here, we consider search queries for a single keyword i.e., q = w ∈ Λ*. The client’s output is DB(w). The protocol may also modify K, σ and EDB.

- Update(K, op, w, id, σ; EDB) inserts an entry to or removes an entry from DB. Input consists of op = add/del, file identifier id and keyword w. The protocol may modify K, σ and EDB.

In the above, we mostly followed the description of [9], [10], [29]. Given the above API, on input the data collection the client can run Setup, followed by N calls to Update to “populate” EDB. Assuming the scheme is forward private (see below) this leaks nothing more than running an initial setup operation on the DB. Other works [25], [39] model Update as “file” addition or deletion, where the protocol adds/removes all the relevant keywords to/from DB. This is functionally equivalent as this process can be decomposed to multiple calls of the above Update protocol.

At a high level, Σ is correct if the returned result DB(w) is correct for every query (for a formal definition, see [12]). The privacy of Σ is parametrized by a leakage function L = (LStp, LStrch, LUdp)[t] that describes the information revealed to the server throughout the protocol execution. LStp refers to leakage during setup, LStrch during a search operation, and LUdp during updates. Standard search leakage types form the literature include search pattern that reveals which searches are related to the same w, and access pattern that reveals DB(w) during a search for w. Note that access pattern leakage is unavoidable if the client wishes to retrieve the actual files and not just their identifiers (unless the files themselves are stored in a protected manner, e.g., Oblivious RAM). Schemes that avoid this leakage are called result hiding.

Informally, a secure SSE scheme with leakage L should reveal nothing about the database DB other than this leakage. This is formally captured by a standard real/ideal experiment with two games RealSSE, IdealSSE shown in Figure 13 in Appendix A, following the definition of [52].

Definition 1 ([52]): A DSE scheme Σ is adaptively-secure with respect to leakage function L, iff for any PPT adversary Adv issuing poly(λ) queries/updates q, there exists a stateful PPT simulator Sim = (SimInit, SimSearch, SimUpdate) such that |Pr[RealSSEAdv(λ, q)] - Pr[IdealSSEAdv, Sim, L(λ, q) = 1]] ≤ v(λ).

Forward and backward privacy. SE schemes with forward and backward privacy aim to control what information is revealed in relation to updates. Informally, a scheme is forward private if it is not possible to connect a new update to previous operations, when it takes place. E.g., it should be impossible to tell whether an addition is for a new keyword or a previously searched one.

Definition 2 ([10]): An L-adaptively-secure DSE scheme that supports single-keyword additions/deletions is forward private iff the update leakage function LUpdt[•] can be written as: LUpdt(op, w, id) = L′Updt(op, id) where L′ is a stateless function, op = add/del, and id is a file identifier.

Backward private DSE schemes limit the information that the server learns during a search for w for which some entries have been previously deleted. Ideally, the scheme should reveal nothing about these deleted entries and, at the very least, not their corresponding file identifiers [52]. Bost et al. [10] gave the first formal definition for three types of backward privacy with different leakage patterns, from Type-I which reveals the least information to Type-III which reveals the most. In order to present their definition, we need to first define some additional functions.

Let Q be a list with one entry for each operation. For searches the entry is (u, w) where u is the timestamp and w is the searched keyword. For updates it is (u, op, (w, id)) where op = add/del and id is the modified file. TimeDB(w) = {(u, id) | (u, add, (w, id)) ∈ Q ∧ ∀u’, (u’, del, (w, id)) ∉ Q} is the function that returns all timestamp file-identifier pairs of keyword w that have been added to DB and have not been deleted. Updates(w) = {u|(u, add, (w, id)) ∈ Q} is the function that returns the timestamp of each insertion/deletion operation for w. Finally, DelHist(w) = {(u, add, del) | ∃ id : (u, add, (w, id)) ∈ Q ∧ (u, del, (w, id)) ∈ Q} is the function that returns for each deletion timestamp the timestamp of the corresponding insertion it cancels. Using the above functions, backward privacy is defined as follows.

Definition 3 ([10]): An L-adaptively-secure SSE scheme has backward privacy:

BP-I (BP with insertion pattern): if \( L^\text{Updt}(op, w, id) = L^\prime(op) \) and \( L^\text{Strch}(w) = L''(\text{TimeDB}(w), a_w) \).

BP-II (BP with update pattern): if \( L^\text{Updt}(op, w, id) = L^\prime(op, w) \) and \( L^\text{Strch}(w) = L''(\text{TimeDB}(w), \text{Updates}(w)) \).

BP-III (weak BP): if \( L^\text{Updt}(op, w, id) = L^\prime(op, w) \) and \( L^\text{Strch}(w) = L''(\text{TimeDB}(w), \text{DelHist}(w)) \).

where \( L^\prime \) and \( L'' \) are stateless functions. We stress that the above definitions (even BP-I) reveal the files currently containing w due to TimeDB(w)—this is in order to account for the leakage from retrieving the actual files. One could define an even stronger definition that avoids this leakage (in practice this could be achieved by using oblivious storage, or when limited to applications that look to return just the identifiers and not the files). None of our constructions explicitly leaks TimeDB(w); indeed we never use it in our proofs for simulation.

An efficient static scheme. Our SDa and SDd schemes use as a building block PiBas, a very simple and efficient static SE scheme from [12]. Static schemes do not allow for updates as the entire DB is set up ahead of time. Security is modified analogously in Definition 1 by allowing only for search oracles after initialization and removing SimUpdate. Since we need a result-hiding scheme (in order to get backward privacy), we slightly modify PiBas. We present the scheme in detail in Appendix A.

---

Footnote:

The version we use corresponds to \( \Pi^{\text{IBas}}_{\text{sd}} \) from [12].

5
DSE with optimal search time. The majority of existing DSE schemes adopt the approach of “storing” deletions as regular entries. During searches, they are used to filter out which insertion entries have been removed. This approach implies that the search cost will be \( \Omega(n_w) \), i.e., linear in the total number of total updates for \( w \), as opposed to the optimal cost \( O(n_w) \), linear in the number of files currently containing \( w \). Notable exceptions to these are the construction of Stefanov et al. [52] (which, however, is not backward private) and two constructions from the recent work of Ghareh Chamani et al. [29] which have quasi-optimal search time according to the following definition.

**Definition 4 ([29]):** A DSE scheme \( \Sigma \) has optimal (resp. quasi-optimal) search time, if the asymptotic complexity of Search is \( O(n_w) \) (resp. \( O(n_w \cdot \text{polylog}(N)) \)).

Oblivious Maps. Our constructions use in a black-box manner an oblivious map (OMAP) which is a privacy-preserving version of a key/value map data structure that aims to hide the type and content of a sequence of operations performed. Intuitively, for any two possible sequences of polynomially many operations, their resulting access patterns (i.e., the sequence of memory addresses accessed while performing the operations) must be indistinguishable. An OMAP offers a setup algorithm for initializing the structure, and two interactive protocols get, put for retrieving the value for a key, and inserting a key/value pair, respectively (see [55], [48] for a formal definition). In our constructions, we use the OMAP of [55] with block size \( O(\log N) \). When reporting OMAP asymptotic efficiency, we always measure number of blocks.

**Range Cover Techniques.** Consider a domain \( A \) of size \( N \). We construct a full binary tree over its values bottom-up. Nodes are labeled with numbers from 1 to \( 2|A| - 1 \) in a left-to-right and left-to-right manner. Given a range (i.e., a sequence of contiguous values) over \( A \), a range covering technique selects a set of nodes whose subtrees collectively cover the given range entirely. Best Range Cover (BRC) essentially selects the minimum number of nodes that cover exactly the range (also called minimum dyadic intervals). For range size \( R \), there are \( O(\log R) \) such nodes. In Figure 5, for \( A = 1, \ldots, 8 \), BRC covers the range \([2, 5]\) with the nodes 2, 10 and 5.

### III. From Static to Dynamic Schemes

**A. Amortized construction**

Our starting point is a static, result-hiding searchable encryption scheme \( SE \), which we modify to store triplets of the form \((w, id, op)\) (instead of the standard \((w, id)\), where \(op = add/del\)). The main idea behind our DSE construction called \( SD_d \) (Figure 2), is to organize \( N \) (without loss of generality, let \( N \) be a power-of-two) updates into a collection of \( \log N \) independent encrypted indexes \( EDB_0, \ldots, EDB_{\log(N-1)} \) for sizes \( 2^0, \ldots, 2^{\log(N-1)} \), each one created with a separate invocation of \( SE.Setup \) with a fresh key.

Initially, all \( EDB_i \) are empty. For the first update the client sets up an encrypted index for the singleton set \((w, id, op)\) using \( SE.Setup \) and sends it to server who stores it as \( EDB_0 \). For future updates, let \( j \) be the smallest value for which \( EDB_j \) is empty. The server first sends to the client all \( EDB_i \) for \( i < j \) and deletes them locally. The client fully decrypts them (we denote this in Figure 2 with \( SE.DecryptAll \) function) and runs \( SE.Setup \) for the union of their entries, together with the current update \((w, id, op)\). Note that the total size of the returned \( EDB_j \) is \( 2^j - 1 \), thus the output of \( SE.Setup \) is a new encrypted index of size \( 2^j \); this is sent to the server who stores it as \( EDB_j \). At all times, the client stores locally the corresponding keys and states of the different non-empty instance of \( SE \) as \( K \) and \( \sigma \).

For searches, the parties run \( SE.Search \) for each (i.e., non-empty) instance of \( SE \) and return all the individual search results. Since \( SE \) is result-hiding, the client needs to do the extra work of decrypting the returned values and extracting the pairs \((id, op)\). The final answer is the result of “filtering out” the deleted entries. Figure 1 illustrates the collection of the encrypted indexes after each of five consecutive inserts.

**Security.** We assume that the underlying \( SE \) scheme is adaptively secure. Regarding forward privacy, note that each update \((w, id, op)\) results in running \( SE.Setup \) with a freshly chosen key. The size of the encrypted index \( (2^j \text{ in the above description}) \) is fully determined by the number of previous updates, thus an update operation can be perfectly emulated by the setup simulator of \( SE \), even if the setup leakage of \( SE \) is just the database size. This implies that the information the server sees during updates, is independent of any previous entries in \( EDB \) (including entries about \( w \)) which gives us forward privacy. Regarding backward privacy, things are also straightforward. Firstly, since \( SE \) is result-hiding and we store deletions as regular entries, the server does not learn the indexes of files that previously contained \( w \). Moreover, during searches the server learns \( |DB(w)| \) as well as how many result elements come from each of \( EDB_i \). In order to simulate the second part, we only need to know when each update for \( w \) took place—this information together with the total update count so far, determines in which \( EDB_i \) each update resides. We previously defined this information as \( Updates(w) \), hence our scheme is BP-II.

Observe that \( SimSearch \) does not always need \( Updates(w) \) to simulate the search transcript. It suffices
Let SE = (Setup, Search, DecryptAll) be a result-hiding, static searchable encryption scheme.

\((K, \sigma, EDB) \leftarrow \text{Setup}(1^\lambda)\)

1. Set \(EDB\) to be an empty vector of indexes \(EDB_i\)
2. Set \(K, \sigma\) to be empty vectors

\((K, \sigma, EDB) \leftarrow \text{Update}(K, op, w, id, \sigma; EDB)\)

Server:
1. Find the minimum \(j\) such that \(EDB_j = \emptyset\)
2. Send to client \(EDB_0, \ldots, EDB_{j-1}\)

Client:
3. Set \(A \leftarrow \emptyset\)
4. \(\textbf{for } i = 0, \ldots, j - 1 \textbf{ do} \)
5. \(X_0 \leftarrow A \cup \text{SE.DecryptAll}(K[i], \sigma[i], EDB_i)\)
6. \(X_i \leftarrow X_{i-1} \cup \emptyset\)
7. \((K[j], \sigma[j], EDB_A) \leftarrow \text{SE.Setup}(1^\lambda, A \cup (w, id, op))\)
8. Send \(EDB_A\) to server

Server:
9. Set \(EDB_j \leftarrow EDB_A\)
10. \(\textbf{for } i = 0, \ldots, j - 1 \textbf{ do} \)
11. Set \(EDB_i \leftarrow \emptyset\)

\(DB(w) \leftrightarrow \text{Search}(K, q, \sigma; EDB)\)

Client ↔ Server:
1. \(X \leftarrow \emptyset\)
2. \(\textbf{for all } i \text{ such that } EDB_i \neq \emptyset \textbf{ do} \)
3. \(X_{i+1} \leftarrow \text{SE.Search}(K[i], q, \sigma[i]; EDB_i)\)
4. \(X \leftarrow X_{i+1} \cup X_i\)
5. Decrypt entries of \(X\) with \(K\) and parse them as \((id, op)\)
6. \(DB(w) \leftarrow \{id \mid (id, add) \in X \land (id, del) \notin X\}\)

Fig. 2: SD\(_2\): from static to dynamic (amortized version).

To know which index each update should be mapped to according to its timestamp. The actual leakage can be much smaller—depending on the update counter upd it may be as small as |Updates(w)| (e.g., if upd = 2\(^i\) for some \(i \in \mathbb{N}\), the largest index has just been rebuilt and the previous ones are empty, hence all the entries for \(w\) will come from the same index and SimSearch does not need their individual timestamps).

**Efficiency.** After \(N\) updates, SD\(_2\) consists of log \(N\) encrypted indexes, each of which is either empty or stores exactly 2\(^i\) items. Assuming SE has linear storage, SD\(_2\) has server storage \(O(N)\). If SE has optimal search time, the query cost for retrieving all the updates for \(w\) is \(O(a_w)\). Since there can be at most \(\log N\) non-empty indexes \(EDB_i\), and a search needs to be performed in each of them, the total search time for SD\(_2\) is \(O(a_w + \log N)\). Finally, after 2\(^i\) updates the client will have run SE.Setup once for size 2\(^i\) and once for 2\(^j\)-1, twice for 2\(^j\)-2, etc., all the way down to 2\(^1\)-1 times for size one. Assuming an underlying static scheme with linear setup time, the amortized cost per update after \(N\) updates is \(O(\log N)\).

One static scheme that satisfies these assumptions is the PiBas construction of [12], which we describe in Figure 12.

Moreover, with PiBas the client has to store one key for each instance and this requires from the client to store \(O(\log N)\) keys. In order to reduce the local storage to \(O(1)\), we can generate the key for each instance pseudorandomly from a single master secret key using a PRF. Instantiated with PiBas, SD\(_2\) requires a single roundtrip for retrieving the result \(DB(w)\). Updates require one roundtrip for retrieving the old indexes to be merged, and one more message from the client to the server (possibly “piggy-backed” to the next operation) for writing the new \(EDB_j\).

With SD\(_2\) it is easy to clean-up deleted entries. During updates, before creating the merged \(EDB_j\) the client identifies all the entries in \(EDB_i\), for \(i < j\), that have corresponding deletions and removes them (padding with dummy records to fill up \(EDB_j\)).

**B. De-amortized construction**

Recall that our key goal is to design schemes with small client storage. SD\(_2\) has excellent performance, albeit in the amortized setting: during updates the client needs to download and locally rebuild an encrypted index. Most times, that index will be relatively small but once in a while this index will become very large (up to the entire DB) as it is shown in Figure 9(c), which invalidates our key goal. To overcome this obstacle, we present here a de-amortized version of our SD\(_2\) construction which we call SD\(_d\). Unlike our amortized scheme that can work with any result-hiding static scheme, SD\(_d\) requires the setup process of the static scheme to be efficiently decomposable to discrete steps so that the necessary local state for executing each step is small and efficiently retrievable—PiBas is again a natural candidate, hence SD\(_d\) is specifically instantiated with it. The reason for this is that the key technical idea is inspired by the classic lazy rebuild technique of Overmars and van Leeuwen [46]. The \(O(2^i)\) steps necessary for running PiBas.Setup for a database of 2\(^i\) elements are split over the previous 2\(^i\) updates, executed one at a time.
Let (KeyGen, Setup, Map, Search) refer to the PiBas routines [12], as described in Figure 12.

\[(K, \sigma; EDB) \leftarrow \text{Setup}(\lambda, N)\]

1: Set \(\ell \leftarrow \lceil \log N \rceil\)
2: for \(i = 0, \ldots, \ell \) do
3: Initialize OMAP with capacity \(2^i\)
4: Set OLDEST\(_i\), OLDER\(_i\), OLD\(_i\), and NEW\(_i\) to \(\emptyset\)
5: Set \(cnt_i \leftarrow 0\)
6: Set \(EDB \leftarrow \{\text{OMAP}_i, \text{OLDEST}_i, \text{OLDER}_i, \text{OLD}_i, \text{NEW}_i, cnt_i\}\) for \(i = 0\)
7: Set \(update_i \leftarrow 0\)
8: Set \(\sigma \leftarrow \{update_i, cnt_i\} \) for \(i = 0\) and the OMAP states
9: Set \(K\) an empty matrix of size \(4 \cdot (\ell + 1)\)
10: for \(i = 0, \ldots, 3\) do
11: for \(j = 0, \ldots, \ell \) do
12: \(K[i][j] \leftarrow \text{PiBas.KeyGen}(1^\lambda)\)

\[(K, \sigma; EDB) \leftarrow \text{Update}(K, op, w, id, \sigma; EDB)\]

Client \(\leftrightarrow\) Server:
1: for \(i = \ell, \ldots, 1\) do
2: if \(\text{OLDEST}_{i-1} \neq \emptyset \land \text{OLDER}_{i-1} \neq \emptyset\) then
3: if \(cnt_i < 2^{i-1}\) then
4: Server sends to client OLDEST\(_{i-1}[cnt_i]\) who decrypts it with \(K[i-1][0]\) and parses it as \((w', id', op')\)
5: else server sends to client OLDER\(_{i-1}[cnt_i] \) which decrypts it with \(K[i-1][1]\) and parses it as \((w', id', op')\)
6: \(cnt_i + 1\)
7: Client computes \(num\) as the number of times NEW\(_i\) has been fully rebuilt and \(c_w \leftarrow \text{OMAP}_i.get(w', num)\)
8: if \(c_w = \bot\) then client sets \(c_w = 0\)
9: Client sets \(c_{w'} = c_w + 1\) and runs \(\text{OMAP}_i.put((w', num), c_{w'})\)
10: Client sends to server \((key, value) \leftarrow \text{PiBas.Map}(w, id', op', c_w, K[i][3])\)
11: Server runs NEW\(_i.put(key, value)\)
12: if \(\text{NEW}_i = 2^i\) then \(\triangleright \) Client can deduce this from \(update_i\)
13: Server sets OLDEST\(_{i-1} \leftarrow \text{OLD}_i\(_{i-1}\) \) and \(\text{OLDER}_i \leftarrow \emptyset\)
14: if \(\text{OLDEST}_i = \emptyset\) then server sets \(\text{OLD}_i\(_{i}\) \leftarrow \text{NEW}_i\) and client sets \(K[i][0] \leftarrow K[i][3]\)
15: else if \(\text{OLDER}_i = \emptyset\) then server sets \(\text{OLD}_i\(_{i}\) \leftarrow\text{NEW}_i\(_{i}\) \) and client sets \(K[i][1] \leftarrow K[i][3]\)
16: else server sends \(\text{OLDER}_i\(_{i}\) \leftarrow\text{NEW}_i\(_{i}\) \) and client sets \(K[i][2] \leftarrow K[i][3]\)
17: Client sets \(K[i][3] \leftarrow \text{PiBas.KeyGen}(1^\lambda)\)
18: Client sets \(K[0][3] \leftarrow \text{PiBas.KeyGen}(1^\lambda)\)
19: Client runs \(\text{PiBas.Setup}(K[0][3], (w, id, op))\) and sends the output to server who stores it as \(\text{NEW}_0\)
20: if \(\text{OLDEST}_0 = \emptyset\) then server sets \(\text{OLDEST}_0 \leftarrow \text{NEW}_0\) and client sets \(K[0][0] \leftarrow K[0][3]\)
21: else if \(\text{OLDER}_0 = \emptyset\) then server sets \(\text{OLD}_0 \leftarrow \text{NEW}_0\) and client sets \(K[0][1] \leftarrow K[0][3]\)
22: else server sets \(\text{OLD}_0 \leftarrow \text{NEW}_0\) and client sets \(K[0][2] \leftarrow K[0][3]\)
23: Client sets \(update_i \leftarrow update_i + 1\)

\(DB(w) \leftarrow \text{Search}(K, q, \sigma; EDB)\)

Client \(\leftrightarrow\) Server:
1: \(X \leftarrow \emptyset\)
2: for \(i = \ell \cdots 0\) do
3: if \(\text{OLDEST}_i \neq \emptyset\) then
4: \(X \leftarrow X \cup \text{PiBas.Search}(K[i][0], \text{OLDEST}_i)\)
5: if \(\text{OLDER}_i \neq \emptyset\) then
6: \(X \leftarrow X \cup \text{PiBas.Search}(K[i][1], \text{OLDER}_i)\)
7: if \(\text{OLD}_i \neq \emptyset\) then
8: \(X \leftarrow X \cup \text{PiBas.Search}(K[i][2], \text{OLD}_i)\)

Client:
9: Decrypt the entries of \(X\) with \(K\) and parse them as \((id, op)\)
10: \(DB(w) \leftarrow \{id \mid (id, add) \in X \land (id, del) \notin X\}\)

Fig. 4: \(SD_d\): from PiBas to DSE (de-amortized version).

With \(SD_d\), four encrypted indexes OLDEST\(_i\), OLDER\(_i\), OLD\(_i\), and NEW\(_i\) are maintained for each \(i = 0, \ldots, \log N - 1\) (as it is shown in Figure 3). Each of the “old” indexes is either empty or contains exactly \(2^i\) items. Moreover, if OLDEST\(_i\) is empty then so is OLDER\(_i\), and if OLDER\(_i\) is empty then so is OLD\(_i\). The fourth data structure NEW\(_i\) is either empty or a partially built index. The setup process of NEW\(_i\) is executed over \(2^i\) updates. In this manner, we guarantee that each entry is stored in exactly one of the “old” encrypted index (across all sizes). Hence, the search protocol (Figure 4) is almost unchanged—the server just needs to search in at most three indexes per size.

Where \(SD_d\) strongly deviates from the amortized construction \(SD_a\) is during updates (Figure 4). Recall that we are using PiBas (Figure 12) but we are storing triplets of the form \((w, id, del)\). The update algorithm passes through all sizes \(i\) from largest to smallest and moves one element from the set of indexes of level \(i - 1\) to the set of indexes of level \(i\). For each level, if both OLDEST\(_{i-1}\), OLDER\(_{i-1}\) are non-empty, this implies that within the next \(2^i\) updates an index of size \(2^i\) needs to have been fully rebuilt—else we would run out of space at level \(i - 1\)! Therefore, one step of PiBas.Setup needs to be executed during each of these \(2^i\) updates, moving one entry \((w, id, op)\) from OLDEST\(_{i-1}\) \cup OLDER\(_{i-1}\) into NEW\(_i\). Moreover, to preserve forward privacy we must guarantee that this step does not reveal any information to the server. We explain how we achieve this next.

The \(EDB\) encrypted index of each PiBas instance con-
tains one map $T$ the keys and values of which are computed with the PiBas.Map function. For each of the $2^i$ updates, the client retrieves from the server and decrypts one entry from the maps $T$ corresponding to OLDEST$_{i-1}$ and OLDER$_{i-1}$, sequentially from beginning to end, i.e., treating the maps as arrays (the position of the next entry to retrieve can be computed efficiently based only on the current global update counter). The Map algorithm takes as input $K, w, id, op, c$ where $K$ is a key for PiBas freshly chosen every time the client starts rebuilding NEW$_i$, $w, id, op$ are read from the retrieved entry, and $c$ is a counter that counts how many times $w$ has already appeared in the NEW$_i$ index. Unfortunately, $c$ cannot be stored locally in an efficient manner. In order to retrieve it, we deploy one oblivious map OMAP$_i$ for each size $2^i$ that maps $w$ to $c$. At every step, the client queries the corresponding OMAP$_i$, uses the retrieved $c$ to run Map, increments it, and stores it back to the same OMAP$_i$.

This leaves one issue to be handled: Between rebuilds of NEW$_i$, the counters $c$ need to be reset as PiBas searches always start from zero. Since the client cannot do that in one pass efficiently, we use an alternative approach. OMAP$_i$ maps $(w, num)$ to $c$, where $num$ is the number of times NEW$_i$ has previously been rebuilt (computable from the current global update counter). When querying OMAP$_i$, for $(w, num)$, the client treats all returned entries with $num' < num$ as null and can safely overwrite them.

Every time NEW$_i$ is fully built (i.e., has size $2^i$), the server moves it to the oldest non-empty index among OLDEST$_i$, OLDER$_i$, OLD$_i$. Moreover, both OLDEST$_{i-1}$ and OLDER$_{i-1}$ are deleted since their purpose is served—all of their entries have been moved to NEW$_i$. Then, if OLD$_{i-1}$ exists, the server moves it to OLDEST$_{i-1}$. Finally, every update creates a “singleton” encrypted index at the oldest available slot for size 0 for the newly inserted entry. All PiBas instances are always instantiated with a freshly chosen key. Figure 3 illustrates the encrypted indexes after seven consecutive inserts.

**Security.** The backward privacy of SD$_d$ is proven exactly in the same manner as that of SD$_a$ since the search protocol is essentially the same. Forward privacy follows from these observations. First, for each update $(w, id, op)$ the server sees a new PRF evaluation since we choose new PiBas keys for each instance and always increment the keyword counter for that keyword-instance combination. Second, our modified version of PiBas is response-hiding. Third, each update accesses a predetermined position in at most $2 \cdot \log N$ map data structures, and $\log N$ read/write oblivious map queries that do not reveal anything to the server about the accessed entries.

**Efficiency.** Updates require $O(\log N)$ OMAP queries. With the oblivious map of [55], their total access overhead is $O(\log^3 N)$, which is the dominating cost for updates. This is worst-case asymptotic update efficiency, as opposed to the amortized performance of SD$_a$. Search time is $O(a_w + \log N)$, same as that of the amortized version (up to three times slower due to multiple structures per size). Server storage is linear to the number of total updates; more concretely, the client chooses an upper bound on the total number of updates ahead of time and initializes the oblivious maps—for better server space efficiency, the above initialization can be split into multiple smaller steps, i.e., when the set of indexes of size $i − 1$ becomes full we start initializing with dummy values the oblivious maps for the indexes of size $i + 1$.

Crucially, permanent client storage is $\tilde{O}(\log^2 N)$ in order to store all the OMAP stashes and the PiBas keys. For appropriately chosen parameters, this is very small in practice since these stashes are usually sparsely populated (see experimental evaluation in Section V). If we want to minimize client storage, there are two additional tricks: (i) stashes are stored at the server and downloaded as necessary during updates without affecting the update asymptotics, and (ii) the keys for PiBas are generated with a PRF from a single master secret key. These make the client storage $O(1)$. While describing SD$_d$ in Figure 4, we assume parties store a counter $cnt_i$ that is used to deduce which are the next elements to be retrieved for lazy rebuild. This is just for clarity of presentation; they can be efficiently computed by keeping a global update counter. Finally, although SD$_d$ entails oblivious maps, they are only used during updates and not during searches. As a result of this, while updates require $O(\log N)$ rounds of interaction, searches are still non-interactive.

Clean-up of is somewhat more involved with SD$_d$ but it only affects the performance of updates and not searches. At a high level, the client maintains an additional OMAP $OM_{del}$ accessed with key $(w, id)$. During updates, while writing a record to NEW$_i$, for $i > 0$, the client looks up $OM_{del}$. If he receives ⊥ he proceeds normally, else he writes a dummy value to NEW$_i$ instead.

**IV. EFFICIENT DSE WITH QUASI-OPTIMAL SEARCH**

In this section, we present our third construction QOS that achieves quasi-optimal search time, according to Definition 4. The only existing backward-private constructions that achieve this are Orion and Horus from [29]. Both these schemes replace each of the $n_w$ accesses necessary for retrieving the result $DB(w)$ with an oblivious map/oblivious RAM access. Contrary to this, QOS requires a single read and write to an oblivious map during search (independently of $n_w$); the remaining computation for retrieving the result is executed at the server by traversing a tree data structure that serves as a “pivot” to identify deleted entries.

The basic idea behind QOS is described in Figure 5. Consider a full binary tree with $N$ leafs, where $N$ is an upper bound on the total number of insertions in the DSE $(N$ can also serve as a trivial bound for the number of deletions). The function $label(v)$ returns a value in $[1, 2N − 1]$ which is the result of the “natural” labeling of tree nodes as follows: The $N$ leafs are labeled from leftmost to rightmost with $1, \ldots, N$. The remaining nodes are labeled in an increasing order per level and from left to right, e.g., the parent of the two leftmost leafs is labeled with $N + 1$, its right sibling with $N + 2$, and so on, all the way to the root that is labeled with $2N − 1$. Every node has a corresponding color $c_v \in \{\text{white, black}\}$; all nodes are initially white. The client holds a “conceptual” tree like this for every keyword $w$. In said tree, inserted entries correspond to leafs that are being populated from left to right, i.e., after $i_w$ insertions (and without deletions), the result of the search is related to the $i_w$ first leafs with labels $1, \ldots, i_w$.

For each deletion, the client needs to mark one node as black. Assuming the deleted entry was previously stored at
Fig. 5: Update tree for QOS with maximum insertions \( N = 8 \). Nodes are labeled with \([1, 15]\) leaves-to-root and left-to-right. This is the tree state after five insertions 1-5, and three deletions for 1, 2, 4. A subsequent search starts from the Best Range Cover of leaves \([1, 5] = (13, 5)\) and proceeds downwards until it finds a black node or a leaf. The result is \((3, 5)\).

The \( j \)-th leaf, this is the node that will be marked as black. However, additional nodes may be marked black according to the following simple rule: “if both children of a node are black, it is also marked black.” Hence, for the above deletion the client needs to access the colors of all the ancestors of the \( j \)-th leaf and their siblings. With this information, he can update their colors accordingly. Simply, each deletion “eliminates” an entire subtree by marking its root black.

During a search after \( i_w \) insertions, the leaves that contain the result can be reached as follows. First, we compute the Best Range Cover for leaves with labels \([1, i_w]\). Then, starting independently from each node in the Best Range Cover the search progresses downwards towards the leaves. If it encounters a black node it stops (knowing that there is no undeleted entry below). Upon reaching a leaf that is not black, the corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs corresponding entry is added to the result.

**Setup.** During Setup (Figure 6), the client initializes three empty OMAPS with capacity \([W], N, N\), respectively:

(i) \( OM_{cnt} \) maps keywords \( w \) to \( cnt_w \) and \( i_w \), where \( cnt_w \) is the number of previous searches for \( w \), and \( i_w \) is number of previous insertions for \( w \).

(ii) \( OM_{del} \) maps each keyword-file identifier pair \( w, id \) to \( label(v) \) where \( v \) is the leaf to which it was inserted; during deletions, this is used to retrieve the “position” of the entry to be deleted.

(iii) \( OM_{state} \) maps a keyword-node label pair \( w, label(v) \) to the color of the node \( v \).

The encrypted index \( EDB \) stored at the server consists of the oblivious maps and two empty maps \( I, D \) of capacity \( N, D \) respectively (\( D \) is an upper bound on deletions that can also serve as the capacity of \( OM_{state} \); trivially it can be set to \( O(N) \)). The client stores locally the states of the three oblivious maps, two PRF keys \( k_I, k_D \) and a symmetric encryption key \( k \).

**Update.** For updates (Figure 6), the client first retrieves the number of previous searches \( cnt_w \) and the insertion count \( i_w \) via \( OM_{cnt} \). Then, we describe the two cases separately. For insertions (lines 2-6), the client increments the update count and writes it to \( OM_{cnt} \). He also writes an entry at \( OM_{del} \) that maps \((w, id)\) to the leaf location where it is stored in \( I \) (this will be used for deleting this entry in the future). Finally, the client encrypts \( id, i_w + 1 \). The resulting ciphertext is stored at the server in map \( I \) at a location computed by the hash function \( H \), using a token \( tk \) that the client computes pseudorandomly with \( k_D \) for \((w, i_w + 1)\).

For deletions (lines 7-17), the client retrieves the label \( pos \) of the tree leaf at which \( w, id \) has been stored via \( OM_{del} \). Then, he computes the labels of all the ancestors and the siblings of the ancestors of \( pos \), and retrieves their colors from \( OM_{state} \) (lines 9-12). With these, he can update the colors of all the ancestors of \( pos \) (in the simplest case, \( pos \) is set to black, more generally this deletion may cause some of its ancestors to become black too). Finally, the client finds \( d_j \), the furthest ancestor of \( pos \) that was first set to black during this deletion. He then marks an entry at \( D \) (at the server) at a location computed by the hash function \( H' \), using a token \( tk \) that the client computes pseudorandomly with \( k_D \) for \((w, d_j)\), as well as store its new color at \( OM_{state} \).

**Search.** During searches Figure 6, the client first retrieves \( cnt_w, i_w \) from \( OM_{cnt} \) and pseudorandomly computes two search tokens for \( w, cnt_w \): (i) \( tk_I \) is computed with \( k_I \) and will be used to retrieve the result, and (ii) \( tk_D \) is computed with \( k_D \) and will be used to identify black nodes encountered during the search, corresponding to deletions. These tokens and \( i_w \) are sent to the server. The client also increments the search counter \( cnt_w \) and stores it to \( OM_{cnt} \).

The server first computes the set of tree nodes \( d_0, \ldots, d_m \) that constitute the Best Range Cover of leaf nodes \([0, i_w]\)—each entry of \( DB(w) \) will be related to a descendant of one of \( d_i \). The search process is quite simple and it entails a recursive search process starting from each of \( d_i \) and progressing downwards at the tree (Figure 6, Algorithm RecSrc). At each node \( d \), the server checks whether the location \( H'(tk_D, d) \) has been written at \( D \), in which case, this is a “black” node, i.e., any previously inserted entries at the subtree with root \( d \) have since been deleted. Hence, the server can simply record its node label and return. Otherwise, he proceeds to parse its children. Upon reaching a leaf that is not black, the server returns the encrypted entry from \( I \) at position \( H(tk_I, d) \)—since \( d \) is a non-deleted leaf, it corresponds to an entry of \( DB(w) \).

The server returns to the client all retrieved values from \( I \) and all marked entries from \( D \) that correspond to black nodes encountered during the tree traversal (and removes them from \( I, D \)). The client computes \( DB(w) \) by decrypting the first ones. Finally, he “re-maps” all the entries of \( I \) and \( D \), using new pseudorandom tokens with keys \( k_I, k_D \) respectively but increased search counter \( cnt_w \), and sends them back to the server who stores them at \( I \) and \( D \).

**Security.** QOS is forward-private because during updates the server observes two types of accesses: (i) a fixed number of oblivious map operations (depending on the type of update) that reveal nothing, and (ii) a pair \((key, value)\) that consists of the outputs of a hash function modeled as a random oracle, and a semantically secure ciphertext. The latter clearly reveals nothing. For the former, note that we ensure that the same input
is never passed to the random oracle twice during updates. This follows from incrementing $i_w$ during insertions and from the fact that deletions never mark the same node as black. Since the input to the random oracle contains a token computed from a PRF for which the server does not have the key (and is only revealed during a future search), querying the oracle for “valid” values not previously seen is infeasible. Finally, note that after every search both tokens are changed so the server cannot connect future updates with ones prior to the search.

Regarding backward privacy, during searches the server learns the PRF tokens $k_T$, $k_D$ which allows him to compute the $I, D$ locations that he needs to access. This also allows him to recall when these entries in $I, D$ where made, i.e., the timestamp and type for all update operations for the queried keyword $w$. Moreover, since the topology of the tree is revealed to the server and the leaves of the tree are naturally mapped to timestamps of insertions, the server can deduce exactly which deletion canceled which prior insertion. As a result of this, QOS achieves BP-III.

**Efficiency.** Updates with QOS require $O(\log N)$ OMAP queries resulting to a total of $O(\log^3 N)$ operations and $\log N$ roundtrips, using the OMAP of [55]. Setup is linear to $N, D$, the upper bound on insertions and deletions, as is the server's storage. The search time can be computed as follows. The OMAP queries take $O(\log^2 |W|)$ operations. Computing the Best Range Cover takes $O(\log |W|)$, same as the cover size itself. Parsing the tree in order to retrieve the result, takes $O(n_w \log i_w)$ since $n_w$ leaves will be reached and the maximum height from each of them to the one of the nodes in the Best Range Cover is $\log i_w$. Even if every node along this traversal has a black sibling (which is a huge overestimation), the total number of black nodes encountered is $O(n_w \log i_w)$ as well. From all the above, the total search overhead with QOS is $O(n_w \log i_w + \log^2 |W|)$ and it takes $O(\log |W|)$ rounds of interaction.
The client’s permanent storage is $O(\log^2 N)$ due to the OMAP stashes. If necessary, this can again be reduced to $O(1)$ at no asymptotic cost by storing the stashes at the server.

V. EXPERIMENTAL EVALUATION

We implemented our three schemes in C++ in order to benchmark their performance and compare them with previous works. We used the OpenSSL [4] library for AES for our PRF and semantically secure encryption. For our experiments we used r5.8xlarge AWS machines with 32-core Intel Xeon 8259CL 2.5GHz processor, running Ubuntu 16.04 LTS, with 256GB RAM, 100GB SSD (GP2), and AES-NI enabled. All schemes were instantiated on a single machine with in-memory storage. Our implementations are publicly available in [28].

We compared SD$_a$ and SD$_d$ with the previous state-of-the-art schemes with small client storage which can be achieved by the “word counter + oblivious map” approach. As described in the introduction, several schemes can be used in this manner, but MITRA [29] is simultaneously the most efficient and most secure (BP-II). For QOS, the main competitor is HORUS [29] which is the fastest existing quasi-optimal scheme. Orion achieves BP-I but it is considerably slower in practice. For both these schemes, we used the code provided in [30].

Since we do not adopt a “clean-up” phase for SD$_a$ and SD$_d$, for fairness we also run MITRA without clean-up (this is faster than the MITRA* numbers reported in [29] by up to 50%). We stress that both schemes are compatible with clean-up, with an additional update cost for SD$_d$ and at no additional search cost for either one. For SD$_a$ and SD$_d$ we used one additional optimization, by storing the first 10 levels of the index collections locally. As we demonstrate below, the effect of this on local storage is small enough to be negligible, but it helps further improve their performance otherwise.

Our basic efficiency measurement is computation time and total communication size for search and update operations. In our experimental evaluation, we consider as unit-size a record/tuple (e.g., x-axis in Figures 7(b), 9(a) and 9(b))—the database in bytes can be obtained by multiplying the number of tuples with the tuple size (e.g., in crime dataset [1] the tuple-size is 210 bytes). For real-world application a tuple size may vary from a few bytes to GB, but in this section we focus on the index costs/overheads, i.e., find the tuple/record (e.g., x-axis in Figures 7(b), 9(a) and 9(b))—the communication size when $|DB|$ is $10^9$, SD$_d$ takes 0.09ms, SD$_a$ 0.12ms, and MITRA 1.11ms. As $|DB|$ grows, the OMAP overhead becomes less important, and the performance of the three schemes converges, e.g., for $10^4$ and $|DB| = 10^6$, SD$_d$ takes 9.8ms, SD$_a$ 13.3ms, and MITRA 13.2ms.

QOS has tremendously better search performance compared to the previous best quasi-optimal scheme HORUS, ranging from 4.4 up to 16531× faster. This comes naturally as the number of oblivious operations for HORUS is $O(|DB|)$ ORAM accesses, whereas for QOS it is a single OMAP access to retrieve the counter $a_w$. E.g., for $|DB| = 10^4$ and $|DB| = 10^6$, HORUS takes ~50sec and QOS takes 33.5ms. The performance of QOS is worse than MITRA, which is explained from the relatively small deletion rate (10%)—quasi-optimal schemes like QOS perform better for large deletion rates (for 10% deletions $a_w$ is very close to $n_w$).

Communication size. Figure 7(c) shows the search communication size when $|DB| = 10^5$ for $|DB| = 10^6$. For all schemes, communication is increasing almost linearly with the result size. One exception is QOS and MITRA where for small result sizes (e.g., < 1000) because the communication cost is dominated by the OMAP operations. In practice, QOS requires 19-53× less communication than HORUS whose overhead is dominated by the ORAM accesses. For $|DB| = 10^5$, QOS sends 40KB whereas HORUS sends 2MB. Furthermore, SD$_d$ requires 2-10× smaller communication size for search than MITRA, since both in SD$_d$ and SD$_a$ search does not depend on oblivious accesses. For the same result size as above, SD$_a$ sends 19.7KB, SD$_d$ 19.9KB, and MITRA 44KB.

In addition to the results presented in Figure 7, we also compared our schemes with MITRA with local storage (which can be up to $O(N)$ as we have discussed in the Introduction). We measured the search computation time and communication size of variable database size between $10^6$ and $10^8$ and we report that MITRA is more efficient than QOS, has similar performance with SD$_a$, while it is worse than SD$_d$. In Figure 8, we repeated the above experiments for the search costs, i.e., communication time and computation size, for the crime dataset and we observe similar conclusions.
B. Update performance

**Computation time.** Figure 9 shows (a) the update computation time and (b) the update communication size for variable database sizes for all schemes except for SD_a, which has amortized update cost. The update performance of SD_a is reported in Figure 9(c), which shows the update time (step-by-step) for a sequence of $10^3$ consecutive updates (insertions/deletions), starting from empty DB. As explained above, we store the first 10 levels of SD_d locally to optimize performance, hence for small database sizes the update time is negligible (less than 0.01ms).

For our tested sizes, MITRA is 1 to 21× faster than SD_d (e.g. for $|DB| = 10^6$ its update time is 1ms while SD_d takes 14ms). We stress that the update time of SD_d is increasing with the number of updates, as more OMAP accesses are necessary. Regarding QOS, we consider only delete operations since they are costlier than insertions. Compared to HORUS our deletion time is, as expected, slightly worse—up to 1.7× slower for the tested sizes (e.g., for $|DB| = 10^3$ QOS takes 281ms and HORUS 233ms). Figure 9(c) shows the SD_d update time for $10^3$ consecutive updates. For each update, the client has to fetch and merge some of the previously filled indexes, which corresponds to the variable cost shown in the plot. For $10^3$ updates, the minimum and maximum observed times are 1μs and 777μs, and the average is 7μs.

**Communication size.** Figure 9(b) shows the update communication size which (not surprisingly) has similar patterns with Figure 9(a). For SD_d, due to our optimization (keeping 10 levels locally) the communication size for $|DB| \leq 10^4$ is zero.

For $|DB| > 10^3$, the cost of SD_d is larger than MITRA (e.g., 508KB vs. 12KB for $|DB| = 10^6$). Regarding QOS versus HORUS, the first requires 0.9 to 3.1× more communication than the latter (e.g. for $|DB| = 10^8$ QOS sends 1.6MB, whereas HORUS sends 536KB).

As it is expected, MITRA with local storage has significant more efficient update costs compared to our schemes, since it takes advantage of storing locally the word counters (assuming up to $O(N)$ local storage)—for each update MITRA with local storage requires to compute two PRF evaluations and to store the new value on the server. However, we consider these sizes essentially negligible for modern devices, even for tablets and mobile phones.

**D. Quasi-optimal search performance for variable deletion percentages**

Our main motivation for studying DSE with quasi-optimal search time is to avoid paying the cost of past deletions during searches. In all the above experiments, we assume a 10% deletion ratio, rendering the effect of deletions for search negligible. Now, we focus on our new quasi-optimal scheme QOS and we provide experiments for variable deletion ratios.

As is evident from the experiments so far, QOS vastly outperforms the previous state-of-the-art quasi-optimal DSE HORUS for searches. In this set of experiments, we compare QOS with SD_d and MITRA (the latter two schemes had better performance for 10% deletions). In this setting, we first insert a fixed number of entries $i_w$ for keyword $w$ and then report the search time after deleting a percentage of $i_w$ between 0-90%. We focus on two cases $i_w = 100$ (small results) and
The results are different for \( i \) OMAP accesses. However, similar search times since the main bottleneck for both is the rate increases. On the other hand, the search time of search using OMAP, (c) \( i \) Fig. 11: Crime dataset—Search computation time and variable deletion percentage for: (a) \( i = 100 \) using OMAP, (b) \( i = 20K \) using OMAP, (c) \( i = 100 \) storing word counters locally, (d) \( i = 20K \) storing word counters locally.

\( i = 20K \) (large results). Since both SD\(_d\) and Mitra have search \( \Omega(a_i) \) their performance should worsen as the deletion rate increases. On the other hand, the search time of QOS should be better. Hence, we want to find at which deletion ratio QOS will outperform the others.

Figure 10 shows the results for small (a) and large (b) \( i \) respectively. First, for \( i = 100 \) QOS and Mitra have similar search times since the main bottleneck for both is the OMAP accesses. However, QOS becomes slightly faster and Mitra slightly slower as the deletion ratio increases; the first outperforms the second after roughly 60%. On the other hand, SD\(_d\) remains much faster than both of them throughout the experiment since it does not need to perform OMAP accesses. The results are different for \( i = 20K \) in Figure 10(b). QOS starts off much slower (as was the case in the experiments above), however, it becomes faster very quickly as deletions increase. It outperforms Mitra at \( \approx 65\% \) and even SD\(_d\) at \( \approx 80\% \) ! The reason is that for large \( i \), the OMAP cost becomes a small percentage of the search process.

We believe that these results serve as a good indication for the practical potential of schemes with (quasi-)optimal search time while there is still room for improvement. To further support this point, we consider another scenario in which permanent client storage is not a bottleneck and we implement both QOS and Mitra to store the word counter maps locally (avoiding the OMAP overhead) The results are shown in Figures 10(c) and (d) for \( i = 100 \) and 20K, respectively. They follow the trends of the corresponding Figures 10(a),(b), but the crossover points are moved to the left. QOS becomes better than Mitra for \( \approx 40\% \) deletions for \( i = 100 \), and for \( \approx 60\% \) for \( i = 20K \). Compared to SD\(_d\), it becomes better in both cases at \( \approx 80\% \) (in the previous scenario, SD\(_d\) was strictly better for \( i = 100 \)). In Figure 11, we repeated the experiments for the search costs with variable deletion percentage for the crime dataset and we observe similar conclusions.
VI. CONCLUSION

In this paper, we revisited the problem of forward and backward private DSE. Prior state-of-the-art schemes either require from the client to store a counter per keyword, or obliviously access this information at the server limiting their practicality for real-world applications. We presented three new schemes with constant permanent client storage and better search performance, both asymptotically and experimentally, than previous works. Moreover, our two schemes SD$_d$ and SD$_a$ not only eliminate the need for oblivious accesses during searches but also minimize the required round-trips. The main drawback of SD$_d$ is during updates, since it has an $O(\log N)$ amortized update cost. The latter means that some updates that are cheap (e.g., $O(1)$), and other that are very expensive (e.g., $O(N)$)—see Figure 9(c). This is the main motivation for proposing SD$_a$, which tackles the aforementioned problem by providing a de-amortized update cost. As we have shown, for small deletion percentage both SD$_d$ and SD$_a$ provide better search performance (both asymptotically and experimentally) than the previous state-of-the-art MTRA. However, the main problem of the above schemes is that the search time depends on prior deleted documents, which in the worst-case may lead to $O(N)$ search overheads. Thus, for delete intensive query workloads, we propose QOS which is the most efficient DSE with quasi-optimal search time, improving previous performance of quasi-optimal DSE schemes by orders of magnitude. For many practical scenarios, we believe that SD$_d$ will be a better choice than QOS; however QOS has been a step towards achieving optimal search performance, since it has only a logarithmic overhead compared to the optimal search time and as we show experimentally for $>80\%$ deletion percentage (hypothetical delete-intensive workload) QOS becomes a better choice. Regarding future work, one possible direction is to develop forward and backward private schemes with optimal search time, ideally without oblivious primitives and with small client storage.

ACKNOWLEDGEMENTS

This work was supported in part by NSF awards #1514261 and #1652259, the National Institute of Standards and Technology, and by Hong Kong RGC grant ECS-26208318. We thank Jeremiah Blocki for shepherding the paper and the anonymous reviewers for their valuable suggestions and comments.

REFERENCES


APPENDIX

Figure 13 shows the RealSSE and IdealSSE games for the DSE security definition 1.

The original PiBa$S$ scheme was slightly different: entries for each $w$ were encrypted by a different key (pseudorandomly generated from a master secret key). During search, this key was sent to the server who could decrypt them and directly return the indexes. Since we need a result-hiding scheme (in order to get backward privacy), we modify the scheme in the following manner. First, all entries are encrypted with the same key. During search, the server sends the encrypted values back and the client decrypts them locally. The scheme is described in detail in Figure 12. It is adaptively secure in the random oracle model with setup leakage $|DB|$ and search leakage $|DB(w)|$. The random oracle assumption can be removed without any change in the efficiency of the scheme, by replacing $H$ with a PRF $F$ and having the client send all the PRF evaluations in a “streaming” manner until a stop message has been sent by the server (see [12] for details).

Let $\text{RND} = (\text{KeyGen}, \text{Enc}, \text{Dec})$ be a semantically-secure encryption scheme, $F$ be a PRF, and $H$ be a collision-resistant hash function.

$$(K, EDB) \leftarrow \text{Setup}(1^\lambda, DB)$$

1: Initialize an empty map $T$
2: Set $(k, k') \leftarrow \text{KeyGen}(1^\lambda)$
3: for each $w \in DB$ do
4: Set counter $c \leftarrow 0$
5: $(\text{key}, \text{value}) \leftarrow \text{Map}(K, w, id, c)$
6: Store $(\text{key}, \text{value})$ to $T$; $c++$
7: Set $K \leftarrow (k, k')$; $EDB \leftarrow T$

$$(k, k') \leftarrow \text{KeyGen}(1^\lambda)$$

1: Choose random PRF key $k$ for $F$
2: Set $k' \leftarrow \text{RND.Enc}(1^\lambda)$

$(\text{key}, \text{value}) \leftarrow \text{Map}(K, w, id, c)$

1: key $\leftarrow H(F(k, w), c)$
2: value $\leftarrow \text{RND.Enc}(k', w, id)$

$DB(w) \leftrightarrow \text{Search}(K, q; EDB)$

Client:
1: Send $tk \leftarrow F(k, w)$ to server
2: Set $\mathcal{X} \leftarrow \emptyset$; $c \leftarrow 0$
3: while true do
4: Set $res \leftarrow T.\text{get}(H(tk), c)$
5: if $res = \bot$ then break
6: else $\mathcal{X} \leftarrow \mathcal{X} \cup res$; $c++$
7: Send $\mathcal{X}$ to client
8: Decrypt entries of $\mathcal{X}$ with $k'$ and return them as $DB(w)$

![Fig. 12: Static searchable encryption PiBa$S$ [12].](image)

**Theorem 1:** Assuming $\text{SE}$ is an adaptively-secure result-hiding static searchable encryption scheme, $\text{SD}_a$ is an adaptively-secure DSE according to Definition 1 with $\mathcal{L}^{\text{Upd}}(\text{op}, w, id) = \bot$ and $\mathcal{L}^{\text{Src}}(w) = \text{Updates}(w)$.

**Proof sketch.** Building a simulator $\text{Sim}$ is straightforward, given the existence of a simulator $\text{Sim}_{\text{SE}} = \{\text{SimInit}_{\text{SE}}, \text{SimSearch}_{\text{SE}}\}$. $\text{SimInit}$ returns empty vector $EDB$ and initializes and update counter $upd = 0$. During each update, $\text{SimUpdate}$ computes $j$ as the least significant zero bit position of $upd$, runs a new instance $\text{Sim}_{\text{SE}}^{(j)} = \{\text{SimInit}^{(j)}, \text{SimSearch}^{(j)}\}$, executes $\text{SimInit}^{(j)}$ on input $2^j$, and sends the result to the adversary. It also terminates currently running instances of $\text{SimInit}_{\text{SE}}^{(j)}$ for $i = 0, \ldots, j-1$, and increments $upd$. During a search for $w$, let $upd$ be the current update counter. $\text{SimSearch}$ receives as input $\text{Updates}(w)$. It then initializes values $t_0, \ldots, t_{\lceil \log upd \rceil}$ to 0. For each entry $u \in \text{Updates}(w)$, it computes $i$ as the index in which the update with timestamp $u$ was stored (determined by $upd, u$) and increments $t_i$ by one. Finally for $j = 0, \ldots, \lceil \log upd \rceil$, it runs $\text{SimSearch}_{\text{SE}}^{(j)}$ on input $t_j$, and sends all the outputs to the adversary. Assuming $\text{SE}$ is secure and result-hiding,
\( b \leftarrow \text{Real}^{\text{SSE}}(\lambda, q); \)

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

\( b \leftarrow \text{Ideal}^{\text{SSE, Sim, L}}(\lambda, q); \)

1: \( N \leftarrow \text{Adv}(1^\lambda) \)
2: \((\text{st}_S, EDB_0) \leftarrow \text{SimInit}(1^\lambda, N)\)
3: for \( k = 1 \) to \( q \) do
4: \((\text{type}_k, \text{id}_k, w_k) \leftarrow \text{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})\)
5: if \( \text{type}_k = \text{search} \) then
6: \((\text{st}_S; t_k, EDB_k) \leftarrow \text{SimSearch}(\text{st}_S, \mathcal{L}^{\text{Srch}}(w_k); EDB_{k-1})\)
7: else if \( \text{type}_k = \text{update} \) then
8: \((\text{st}_S; t_k, EDB_k) \leftarrow \text{SimUpdate}(\text{st}_S, \mathcal{L}^{U\text{pdt}}(w_k); EDB_{k-1})\)
9: \( b \leftarrow \text{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q); \)
10: return \( b; \)

Fig. 13: Real and ideal experiments for the SSE scheme.

and each instance \( \text{Sim}_{\text{SE}} \) is spawned independently with fresh randomness, and given that the timestamp of an update fully determines the corresponding index structure for its entry, the transcript produced by \( \text{Sim} \) is indistinguishable from the messages observed by the adversary during the real protocol execution. □

Theorem 2: Assuming \( \text{PiBas} \) is an adaptively-secure result-hiding static SE scheme, and OMAP\(_P\) are secure oblivious maps, \( \text{SD}_a \) is an adaptively-secure DSE according to Definition 1 with \( \mathcal{L}^{U\text{pdt}}(\text{op}, w, \text{id}) = \perp \) and \( \mathcal{L}^{\text{Srch}}(w) = \text{Updates}(w) \).

Proof sketch. Let \( \text{Sim}_{PB} = \{\text{SimInit}_{PB}, \text{SimSearch}_{PB}\} \) be the simulator for \( \text{PiBas}. \) First, we observe that \( \text{SimInit}_{PB} \) can be decomposed into calls to a stateful \( \text{SimInitOne}_{PB} \) that simulates just one step of the setup simulation at a time. The input state of \( \text{SimInitOne}_{PB} \) is the partially built table, and the leakage \( N. \) After \( N \) executions, \( \text{SimInitOne}_{PB} \) provides an output that is identically distributed with that of \( \text{SimInit}_{PB} \) on input \( N. \) This follows easily by the fact that the setup process of \( \text{PiBas} \) consists of populating a hash table with \( N \) semantically secure encryptions, stored at pseudorandomly computed positions. The simulator \( \text{SimInitOne}_{PB} \) just needs to remember its previous randomly chosen positions so that eventually he outputs the entire table.\(^5\)

With that observation, we build our simulator \( \text{Sim} \) as follows. First, all calls to OMAP\(_P\), are replaced by simulated accesses. During setup, \( \text{SimInit} \) launches \( 4 \cdot (\ell+1) \) independent instances of \( \text{SimInitOne}_{PB} \) for \( i = 0, \ldots, \ell \) and corresponding sizes 1, 2\(^\ell \), and initializes update counter \( \text{upd} = 0. \) For each update, whenever \( \text{OLDER}_i \) and \( \text{OLDER}_j \) are full (which can be computed from \( \ell \) and \( \text{upd} \)), \( \text{SimUpdate} \) calls \( \text{SimInitOne}_{PB}^{i+1} \). If \( \text{SimInitOne}_{PB}^{i+1} \) is full (after \( 2^{i+1} + 1 \) calls), the simulator terminates the existing \( \text{SimInitOne}_{PB}^{i+1} \) instances mapped to \( \text{OLDER}_i \) and maps the \( \text{SimInitOne}_{PB}^{i+1} \) instance of \( \text{OLD}_i \); if it is not vacant). Moreover, it treats the \( \text{SimInitOne}_{PB}^{i+1} \) instance as mapped to the oldest vacant instance for size \( 2^{i+1} \), and launches a new instance mapped to \( \text{NEW}_i \). Finally, it always launches a new instance of \( \text{SimInitOne}_{PB}^{i+1} \) maps it to the oldest non-vacant instance for size 1, and increments \( \text{upd} \). The search simulator \( \text{SimSearch} \) is identical to that of \( \text{SD}_a \) (it just has to call up to three instances of \( \text{SimSearch}_{PB} \) per size, depending on \( \text{upd} \)).

By the same reasoning as that for \( \text{SD}_a \) above, and since OMAP\(_P\), are independently instantiated with secure oblivious maps, the transcript produced by \( \text{Sim} \) is indistinguishable from the messages observed by the adversary during the real protocol execution. □

Theorem 3: Assuming \( F \) is a PRF, RND is a semantically secure encryption scheme, and the three OMAP\(_P\) are secure oblivious maps, QOS is an adaptively-secure DSE according to Definition 1 in the programmable random oracle model, with \( \mathcal{L}^{U\text{pdt}}(\text{op}, w, \text{id}) = \text{op} \) and \( \mathcal{L}^{\text{Srch}}(w) = \text{Updates}(w) \).

Proof. We prove the security of QOS by defining a sequence of games as follows:

- **Game—0**: This is the \( \text{Real}^{\text{SSE}} \) game as defined in Appendix A.
- **Game—1**: This is the same as Game—0 but during setup the OMAP\(_P\) initializations are replaced with calls to the OMAP\(_S\) simulators for sizes \( W, N, N \) respectively. All future OMAP\(_P\) accesses are emulated by calls to the corresponding access simulators. Game—1 is indistinguishable from Game—0 due to the security of the oblivious maps.
- **Game—2**: This is the same as Game—1, except that the encryptions value computed during update and search are all replaced with dummy zero encryptions. Game—2 is indistinguishable from Game—1 due to the semantic security of RND.
- **Game—3**: This is the same as Game—2, except that the tokens \( t_k, t_k \) generated during update and search are generated uniformly at random from the range of the PRF \( F \), \( \{0,1\}^\lambda \). The first time a token is created for a certain is mapped to the oldest vacant instance for size \( 2^{i+1} \), and launches a new instance mapped to \( \text{NEW}_i \). Finally, it always launches a new instance of \( \text{SimInitOne}_{PB}^{i+1} \) maps it to the oldest non-vacant instance for size 1, and increments \( \text{upd} \). The search simulator \( \text{SimSearch} \) is identical to that of \( \text{SD}_a \) (it just has to call up to three instances of \( \text{SimSearch}_{PB} \) per size, depending on \( \text{upd} \)).

By the same reasoning as that for \( \text{SD}_a \) above, and since OMAP\(_P\), are independently instantiated with secure oblivious maps, the transcript produced by \( \text{Sim} \) is indistinguishable from the messages observed by the adversary during the real protocol execution. □

For simplicity, we assume that the first time \( \text{SimInit}_{PB} \) is called, it just simulates (internally) the key generation process, hence \( \text{SimInit}_{PB} \) will be called a total of \( N + 1 \) times to emulate the execution of \( \text{SimInit}_{PB} \) on input \( N \).
combination it is appended to one of the two lists $\mathbf{TokensI}(w)$, $\mathbf{TokensD}(w)$ (for insertions and deletions respectively), that are different for every keyword. Game−3 is indistinguishable from Game−2 due to the security of the PRF.

- **Game−4**: This is the same as Game−3, except that calls to $H$ are replaced with a programmable random oracle as follows. For general $H$-calls from the adversary, if the input has not be queried before and the result has not been programmed, return a value chosen uniformly at random from the range of $H$ and store the input-result pairs for future consistency. Else, return the previously stored result for this input.

Specifically during insertion updates (line 6), $H$-calls are entirely eliminated and instead $key$ is chosen uniformly at random from the range of $H$. The client holds a list $T_{I}$ where he appends the chosen $key$. If the update is a deletion he appends $\bot$. Note that this also eliminates token generation at line 5.

Then, during search, let $U = (u_{1}, op_{1}), \ldots, (u_{n_{w}}, op_{n_{w}})$ be the list of timestamp-update type pairs corresponding to all previous updates for the queried keyword $w$, sorted by timestamp in increasing order. Let $u'_{1}, \ldots, u'_{i_{w}}$ be the sub-list of $U$ such that $op_{i} = add$, again sorted in increasing order, and let $d_{1}, \ldots, d_{i_{w}}$ be the natural ordering of $u'_{i}$ from $1, \ldots, i_{w}$. The client then programs the oracle such that $H(tk_{I}, d_{i}) = T_{I}[u'_{i}]$. If $H(tk_{I}, d_{i})$ has been set previously (due to an adversarial query involving $tk_{I}$ before this token was revealed), then the game aborts. Finally, line 13 of the search algorithm is replaced with choosing $key$ uniformly at random from the range of $H$.

Let $d_{j} = leaf$, then client sets $T_{I}[u'_{j}] = key$, in preparation of future searches.

First, note that unless the game aborts it produces a transcript identical to Game−3, in the programmable random oracle model for $H$. Given that the range of $H$ is $\{0, 1\}^{3}$, whereas the total number of $H$-calls that the adversary can do beyond the ones required during searches is polynomial in $\lambda$ (since the adversary is PPT), the probability of aborting is negligible in $\lambda$, hence Game−4 is indistinguishable from Game−3.

- **Game−5**: This is the same as Game−4 but we now also replace $H'$ with a programmable random oracle. For general $H'$-calls from the adversary, if the input has not been queried before and the result has not been programmed, return a value chosen uniformly at random from the range of $H'$ and store the input-result pairs for future consistency. Else, return the previously stored result for this input.

Specifically during deletion updates (line 17), $H'$-calls are entirely eliminated and instead $key$ is chosen uniformly at random from the range of $H'$. The client has a list $T_{D}$ where he appends the chosen $key$. If the update is an insertion he appends $\bot$. Note that this also eliminates token generation at line 16.

Then, during search, let $Dels = (v_{1}, v'_{1}), \ldots, (v_{d_{w}}, v'_{d_{w}})$ be the list of all timestamp-pairs that match each deletion timestamp $v_{i}$ to the timestamp $v'_{i}$ of the previous insertion it cancels out, sorted in increasing order such that $v_{i} > v_{i-1}$. Using $U$ (from Game−4) and $Dels$ the client builds the entire update tree for $w$ as follows. First create an empty binary tree with $2^{\lceil \log i_{w} \rceil}$ leaves. Match each leaf $[1, i_{w}]$ to an insertion operation’s timestamp $u'_{i}$ (as computed in Game−4) starting from the leftmost leaf. Then, for every $v_{i} \in Dels$, mark the leaf with timestamp $v'_{i}$ as black and then keep moving upwards, reading at every level its ancestor and the sibling of its ancestor. If both children of a node is black mark it black. After finishing all steps for deletion with timestamp $v_{i}$, let $d'_{i}$ be the node closest to the root that you just marked black. The client then programs the oracle such that $H'(tk_{D}, d'_{i}) = T_{D}[v_{i}]$. If $H'(tk_{D}, d'_{i})$ has been set previously (due to an adversarial query involving $tk_{D}$ before this token was revealed), then the game aborts.

Finally, line 18 of the search algorithm is replaced with choosing $key$ uniformly at random from the range of $H'$. Let $d'_{i} = y$, then client sets $T_{I}[v_{j}] = key$, in preparation of future searches.

First, note that unless the game aborts it produces a transcript identical to Game−3, in the programmable random oracle model for $H$. This holds since the combination of $U, Dels$ uniquely define the colors of the nodes of the update tree for $w$. Then, using the same argument as above but for $H'$, we conclude that Game−5 is indistinguishable from Game−4.

- **Game−6**: This is the same as Game−5 but client receives $op$ instead of $op, w, id$ during updates, and $\mathbf{Updates}(w), \mathbf{DelHist}(w)$ instead of $w$ during searches. Since he does not have access to $w$, he populates lists $\mathbf{TokensI}, \mathbf{TokensD}$ as follows. For a search at timestamp $w$, the client first checks whether the input update history $\mathbf{Updates}(w)$ is an extension of one observed during a previous search that took place during timestamp $w'$. If so, this implies that the searches at times $\tilde{w}$ and $\tilde{w}'$ are for the same keyword and he retrieves $tk_{I}, tk_{D}$ as the latest entries from $\mathbf{TokensI}(\tilde{w}), \mathbf{TokensD}(\tilde{w})$. Else, he chooses fresh random tokens $tk_{I}, tk_{D}$ and appends them to $\mathbf{TokensI}(\tilde{w}), \mathbf{TokensD}(\tilde{w})$.

The client’s code as described in Game−6, is essentially the code of the simulator in the IdealSSE game since it only takes as input the leakage specified in Theorem 3. By a standard hybrid argument, the produced transcript is indistinguishable from the one produced in Game−0, and the result follows. $\square$