SWiSSSE: System-Wide Security for Searchable Symmetric Encryption

Zichen Gui
ETH Zürich
Zürich, Switzerland
zichen.gui@inf.ethz.ch

Kenneth G. Paterson
ETH Zürich
Zürich, Switzerland
kenny.paterson@inf.ethz.ch

Sikhar Patranabis
IBM Research India
Bangalore, Karnataka, India
sikhar.patranabis@ibm.com

Bogdan Warinschi
University of Bristol and DFINITY
Zürich, Switzerland
bogdan.warinschi@dfinity.org

ABSTRACT

This paper initiates a new direction in the design and analysis of searchable symmetric encryption (SSE) schemes. We provide the first comprehensive security model and definition for SSE that takes into account leakage from the entirety of the SSE system, including not only from access to encrypted indices but also from access to the encrypted database documents themselves. Such system-wide leakage is intrinsic in end-to-end SSE systems, and can be used to break almost all state-of-the-art SSE schemes (Gui *et al.*, IEEE S&P 2023). We then provide a static SSE construction meeting our new security notion. The proposed SSE scheme involves a combination of novel techniques: bucketization to hide volumes of responses to queries, and delayed, pseudorandom write-backs to disrupt access pattern. Our implementation and analysis of the proposed scheme demonstrates that it offers very strong security against general classes of (system-wide) leakage-abuse attacks with moderate overhead. Our scheme scales smoothly to databases containing hundreds of thousand of documents and millions of keyword-document pairs. To the best of our knowledge, this is the first end-to-end SSE scheme that effectively suppresses system-wide leakage while maintaining practical efficiency.

1 INTRODUCTION

Searchable Symmetric Encryption (SSE). The goal of Searchable Symmetric Encryption (SSE) [24, 29, 31, 47, 99] is two-fold: (a) to allow a server to execute keyword search queries directly on a collection of a client's encrypted documents in an efficient manner, and (b) to ensure client privacy by minimizing the amount of information "leakage" to the server in the process. Existing proposals for SSE in the literature can be broadly divided into two categories – *static* SSE schemes that support keyword searches for a fixed collection of documents [24, 29, 31], and *dynamic* SSE schemes that also allow for updates of encrypted document collections [16, 19, 26, 28, 66].

Proceedings on Privacy Enhancing Technologies 2024(1), 549–581

https://doi.org/10.56553/popets-2024-0032

Leakage vs Efficiency. The most general notion of SSE with optimal privacy guarantees (wherein little or no information is leaked to the server) can be achieved using techniques like fully homomorphic encryption (FHE) [43] and Oblivious RAM (ORAM) [49]. However, these techniques today incur significant computational and/or communication overheads. So designers of SSE currently opt for better practical performance at the cost of leaking some information to the server [24, 29, 31]. The main challenge in designing SSE schemes is ensuring the right balance between leakage and efficiency, especially because leakage can be exploited by an adversarial server to infer sensitive client information, resulting in privacy breaches [13, 21, 22, 32, 59, 61, 88, 90, 94].

Leakage Mitigation in SSE. Defending SSE schemes against leakage-abuse attacks while maintaining acceptable search performance is difficult. The state-of-the-art is represented by recent work [34, 45, 65, 92] giving SSE schemes that provably leak little or no information to the adversarial server. A common feature underlying these recent proposals (and indeed the majority of prior work) is that they *only* consider leakage from the *encrypted search index*, a specialized data structure for recovering the document identifiers matching a given query. Specifically, these proposals *ignore* leakage from the final step in processing a query, in which the client *actually fetches the encrypted documents from the server.* This leads to the question of whether one could transfer existing leakage-abuse attacks from the index to the document level, thereby breaking the privacy guarantees of the schemes in [34, 45, 65, 92] when properly viewed from a *system-wide* perspective.

System-Wide Leakage. In a very recent paper, Gui *et al.* [54] answer this question in the affirmative by demonstrating that end-to-end SSE systems built in a natural way by applying the volume-hiding EMMs in [34, 45, 65, 92] to the search index only incur *system-wide co-occurrence leakage* during document retrieval that can be exploited to completely break the query privacy guarantees of these schemes in practice.[1] These attacks affect the vast majority of SSE schemes proposed to date [16, 19, 23, 24, 26, 29, 31, 41, 62, 64, 66, 86, 101], and highlight the importance of acknowledging that system-wide leakage is a *major privacy issue* in existing SSE schemes that needs to be addressed.

[1]While Gui *et al.* [54] show concrete attacks on the end-to-end SSE systems built from the volume-hiding EMMs in [65, 92], their attacks extend naturally to end-to-end SSE systems where only the encrypted index (and not the document retrieval step) is protected by dynamic volume-hiding EMMS [45] or ORAM-style techniques [34].

The authors of [54] also demonstrate that known leakage mitigation techniques such as volume-hiding EMMs cannot be efficiently scaled to the overall encrypted database. As one concrete example, the authors of [54] show that for the Enron email database,[2] applying state-of-the-art leakage-mitigation techniques (such as volume-hiding EMMs [34, 45, 65, 92], Oblivious RAMs (ORAMs) [27, 102], or private information retrieval (PIR) [3, 5]) to encrypt whole database incurs storage overheads ranging from $4\times$ to $800\times$ and bandwidth overheads ranging from $20\times$ to $110,000\times$, as compared to a baseline SSE system with no leakage mitigation for document retrieval. To summarize, there exists no practically efficient and scalable end-to-end SSE system that provides system-wide security for both index and document retrieval.

1.1 Our Contributions

These observations motivate a change of perspective. Instead of focussing on securing only one component of an SSE system, namely the encrypted search index, we need to take a system-wide view of SSE and its leakage. In this paper, we propose a new approach to designing and analyzing low-leakage yet efficient SSE schemes supporting keyword searches over static databases. Crucially, we include in our model, construction and analysis for SSE both the (encrypted) index *and* the document retrieval component of a searchable encryption *system*.

System-wide Security Definition. We propose a new security definition for static SSE schemes that takes into account the system-wide leakage during setup and searches. Our definition is for an honest-but-curious server[3] and we model security using the real world/ideal world simulation paradigm with leakage profiles. Security consists of showing that, for every adversary, there is a simulator that given only the leakage profile, can provide an ideal world view for the adversary that is indistinguishable from what it would observe in the real world. Informally, this means that the adversary gains no information beyond the leakage profile. In contrast to previous analysis, we take a system-wide view, including in our execution model and leakage profile the action of retrieving encrypted documents from the data storage component as well as the leakage from the encrypted search index.

New Construction Techniques. We propose a new static SSE scheme named SWiSSSE that supports efficient keyword searches over large real-world document collections. Keyword searches in SWiSSSE require *two* rounds of communication between the client and the server. We prove that SWiSSSE is adaptively secure under our new system-wide security definition with respect to a well-defined leakage profile. To the best of our knowledge, SWiSSSE is the only end-to-end SSE system that is secure against the system-wide leakage-abuse attacks proposed in [54] while supporting practically efficient searches. At the heart of our construction are two techniques, *keyword frequency bucketization* and *delayed, pseudorandom write-backs*.

Bucketization. Keyword frequency bucketization refers to dividing the list of all keywords across multiple buckets such that each keyword in a given bucket is "padded" to have the same frequency as the most frequent keyword in that bucket. This mitigates attacks based on volume leakage. Unlike worst case padding which either imposes a linear search overhead [63] or a quadratic storage complexity, our technique imposes a search overhead proportional to the frequency of the most frequent keyword in each bucket, while retaining linear storage complexity. There is an interplay between the bucketization strategy, the padding overhead, and the security that we obtain. We explore this in detail subsequently.

We note here that a number of prior works on SSE have explored padding-based strategies to mitigate volume leakage (e.g., [17, 21, 22, 34, 65, 92]). However, all of these works primarily consider *index-only* padding; more concretely, they only focus on hiding the volume leakage in the search index, with no clear guidelines on whether the same volume-hiding techniques are subsequently applied during document retrieval. When viewed from a *system-wide* perspective, volume leakage from the document retrieval phase can be used to launch query recovery attacks on any end-to-end SSE system built by using the aforementioned approaches in their original index-only form [54]. Unfortunately, in many cases (such as [65, 92]), extending these padding strategies to document retrieval is practically infeasible due to the huge storage and communication overheads involved, as demonstrated by Gui *et al.* in [54] and also in Section 6.4 of this paper.

On the other hand, our padding strategy is designed for preventing volume leakage from a system-wide perspective (i.e., from both the index and document retrieval phases) in SWiSSSE, while (a) scaling in a communication- and storage-efficient manner to large databases (as demonstrated by our experiments evaluating the performance of SWiSSSE in Section 6.2, Table 1, and also Section 6.4), and (b) resisting system-wide volume leakage-based cryptanalysis (as demonstrated by our cryptanalysis experiments on SWiSSSE over real-world databases in Appendix C). To the best of our knowledge, no existing padding-based strategy achieves both of these features simultaneously.

Write-backs. To mitigate access pattern leakage, we use a delayed, pseudorandom write-back strategy. After each search operation involving a given keyword, we update all locations pertaining to the keyword and the documents which contain it, throughout the entire encrypted search index and the encrypted database, in a pseudorandom manner. To limit the leakage from this "write-back" step, we store the information in a stash on the client side and flush a pseudorandomly chosen fraction of the stash back to the server at regular intervals. Data pertaining to one keyword then gets spread across multiple write-backs. This ensures that the server cannot correlate search operations through write-backs. Now, for correctness, search operations must also be done over the stash, since the server storage at a given moment no longer fully captures the state of the system. We design our scheme such that the latency of search operations is not affected by the latency of write-back operations. The two rounds of online communication needed to service search operations in SWiSSSE contrasts with the polylogarithmic round complexity of ORAM-style schemes [19, 26].

Our write-back strategy also allows SWiSSSE to support significantly faster online query processing as compared to existing ORAM-style SSE solutions [19, 26]. This is because the additional

[2] https://www.cs.cmu.edu/~enron/
[3] This is standard in the SSE literature; see [16, 19, 23, 24, 26, 29, 31, 41, 62, 64, 66, 86, 101].

pseudorandom write-backs in SWiSSSE do not happen during online query processing. This contrasts with existing ORAM-style solutions, where read and write operations are executed entirely during online query processing. In particular, SWiSSSE is designed such that the latency of searches is not affected by the latency of write-back operations, which occur independently and periodically at pseudorandom times. We refer to Section 6.2 (Table 2) for an empirical evaluation of the security vs efficiency tradeoffs offered by a variety of write-back strategies (in particular, by varying the write-back rate and the mean stash size at the client).

System-Wide Leakage Analysis. We formally prove the security of SWiSSSE with respect to our system-wide security definition and specific leakage profile. This leaves open the important question of determining the security impact of that leakage. To answer that question, and as is common in the SSE literature, we perform a detailed cryptanalysis of our scheme. In particular, we show that the known leakage-abuse attacks [13, 21, 59, 88] fail against our scheme. We also show that the highly refined system-wide leakage-abuse attacks of [54] (that are powerful enough to break the query privacy of the vast majority of existing SSE schemes [16, 19, 23, 24, 26, 29, 31, 41, 62, 64, 66, 86, 101]) also fail against SWiSSSE, provided we choose an appropriate bucketization strategy. We have made the cryptanalysis of our own scheme as "unfriendly" as possible but our scheme will benefit from further, independent analysis.

Implementation. We describe a Java implementation of SWiSSSE and evaluate its performance. We used Redis[4] as the underlying database system. For comparability with previous research in SSE, we use the Enron email corpus for our experiments. We experimented with a range of subsets of this corpus to gauge performance for different database and keyword set sizes. As a flavor of our results, for a database of 400K documents which corresponds to 1.3 GB of uncompressed plaintext storage, the client storage used by our scheme is less than 10 MB and the storage used by the server is about 3.6 GB; the throughput of our scheme is 2370 documents per second for setup and 8400 documents per second for search.[5]

Ethics Discussion. We believe that it is justifiable to use Enron for our attack experiments given: (a) the lack of alternative realistic datasets for conducting experiments for our attack, (b) the extensive usage of this dataset for attack experiments in prior works [13, 59, 88], (c) the fact that the data has long already been public, and (d) there has been an effort by the researchers curating the version of the dataset used in this paper to remove users upon request.[6]

Extension to Dynamic Databases. Our system-wide security definitions for SSE and construction techniques for SWiSSSE can be extended in a natural way to handle dynamic databases. In Appendix E, we present new system-wide security definitions for dynamic SSE, detail a dynamic version of SWiSSSE, and prove its security against system-wide leakage-abuse attacks. We choose to focus on the static version of SWiSSSE in the main body of the paper as we consider it to be our primary contribution. Additionally, our two-phase approach of first focusing on static SWiSSSE before extending it to the dynamic setting allows for an easier exposition of our core ideas. We note here that such two-phase approaches are common in the SSE literature (e.g., volume hiding EMMs were first proposed and analyzed extensively in the static SSE setting [65, 92] before being extended to dynamic searchable indices [45]). In the rest of the main body, we use SWiSSSE as a shorthand for *the static version of* SWiSSSE (unless explicitly specified otherwise).

1.2 Related Work

In this section, we compare SWiSSSE with existing SSE schemes and other popular techniques for encrypted database search.

Leakage-Resilient Static/Dynamic SSE. SWiSSSE addresses a new threat model in the design of end-to-end SSE systems (namely system-wide SSE security), and attempts to thwart a new class of system-wide leakage cryptanalysis attacks, which were conceptualized for the first time in [54] and have been formalized and refined further in this paper. A number of recent works in SSE (both static and dynamic) have focused on countering leakage cryptanalysis by using a variety of leakage-suppression mechanisms [4, 25, 33, 37, 45, 56, 57, 65, 92]. However, all of the (dynamic) SSE schemes proposed in these works fall under the category of index-only (dynamic) SSE schemes in the sense that they propose mechanisms to specifically suppress leakage from the encrypted search index, and do not specify how to realize an end-to-end SSE system by building upon such an encrypted index. Hence, when naturally extended to design end-to-end SSE systems (with no additional leakage suppression during actual document retrieval), these schemes incur exactly the same system-wide leakage as observed in [54], and as considered in our security definitions for system-wide SSE as well as our system-wide leakage cryptanalysis experiments. As already demonstrated in [54], this system-wide leakage can be exploited to break the query privacy guarantees of end-to-end SSE systems obtained by naturally extending the aforementioned index-only (dynamic) SSE schemes (with no additional protection for document retrieval). On the other hand, SWiSSSE (in both its static and dynamic versions) is an end-to-end SSE system by design with built-in protections against system-wide leakage abuse attacks. As demonstrated by our leakage cryptanalytic experiments, SWiSSSE is not broken by (highly refined versions of) the system-wide leakage cryptanalysis techniques proposed in [54].

We believe that a fair comparison of the above index-only schemes with SWiSSSE can only be done by extending the leakage suppression mechanisms used by these index-only schemes to the document retrieval phase as well (so as to achieve system-wide leakage protection). Unfortunately, as already shown in [54], the latest advances in leakage-resilient (dynamic) SSE cannot be used to design practically efficient end-to-end (dynamic) SSE systems with protection against system-wide leakage-abuse attacks. Indeed, as shown in [54] and in our own experiments (see Table 3 in Section 6.4), end-to-end SSE systems where the document retrieval phase is also additionally protected using encrypted multi-maps [65, 92] are extremely inefficient in practice due to large storage overheads and high computational/communication costs of query processing. This remains the same for both static and dynamic settings, and would hold despite the latest improvements achieved in leakage-resilient (dynamic) SSE when applied to the index-only setting. While such

[4]https://redis.io/

[5]The query response time for search queries depends on the queries themselves; here we assume a uniform query distribution.

[6]See https://www.cs.cmu.edu/~enron/ for detailed documentation.

techniques and the improvements thereof may be practical and scalable when applied only to the index (which is typically small), they simply do not scale practically to the entire database, and hence cannot be used to protect document retrieval in and end-to-end SSE setting, which is the focus of SWiSSSE. We refer to [54] and to Section 6.4 of our paper for additional discussion.

ORAM and PIR. As the vast majority of SSE schemes in the literature are index-only schemes, it is natural to consider extending these schemes to end-to-end SSE schemes by adding an encrypted document retrieval step. There are two major primitives in the literature that study the document retrieval problem, namely ORAM and PIR. We compare these primitives with the document retrieval strategy of SWiSSSE in the discussion below. We refer the reader to Table 3 in Section 6.4 for a more concrete comparison.

Comparison with ORAM. As mentioned earlier in the introduction, an ORAM allows realizing SSE with little (access pattern) leakage to an untrusted server. Typically, in an ORAM-based solution for document retrieval, the client would outsource the documents to an untrusted server, and subsequently retrieve them on at a time. There is a long line of research on improving the efficiency of (multi-server) ORAMs [8, 9, 49, 98, 102, 109]. However, as pointed out by Larsen et al. [75], the bandwidth lower bound for ORAMs (per document retrieval) is logarithmic in the number of documents. In addition, a typical ORAM scheme requires one round of interaction per document retrieved, leading to a linear (in the total number of documents being retrieved) round complexity and high latency. On the other hand, SWiSSSE achieves a constant bandwidth overhead for document retrieval and all documents are retrieved in a total of two rounds of interaction between the server and the client.

It is worth noting that there have been attempts to build SSE schemes from (multi-server) ORAM [10, 26, 34, 42]. Apart from being index-only schemes, these schemes have logarithmic bandwidth overheads which makes them less efficient than SWiSSSE. In addition, SWiSSSE supports significantly faster online query processing as compared to these ORAM-style SSE solutions. This is because the additional pseudorandom write-backs in SWiSSSE do not happen during online query processing. This contrasts with existing ORAM-style solutions, where read and write operations are executed entirely during online query processing. In particular, SWiSSSE is designed such that the latency of online operations (searches and updates) is not affected by the latency of write-back operations, which occur independently and periodically at pseudorandom time-stamps.

Comparison with PIR. We also present a comparison of SWiSSSE with solutions based on private information retrieval (PIR). Traditionally, the documents in a PIR scheme are assumed to be public. The PIR scheme only protects the privacy of the queries. However, a PIR scheme can be easily adapted to the encrypted document retrieval setting by encrypting the documents. There are many PIR schemes proposed in the single-server single-query (each query retrieves one document) setting [3, 5, 44, 69, 73, 80, 83], single-server batched-query setting [5, 58, 82], and multi-server setting [30, 38, 46, 48]. The main research focus of PIR is on reducing the communication overhead. On the other hand, all (computational) PIR schemes need to access all documents per document retrieval to hide the access pattern, leading to high computational overhead. In comparison, SWiSSSE guarantees that the number of documents accessed by the client is linear in the number of documents to be retrieved. This makes SWiSSSE significantly more efficient than PIR schemes in terms of computational overhead.

Worst-Case Padding and Frequency Smoothing. We note that certain previous works (e.g., [59]) suggest using "worst-case" padding, where the goal of the padding is to make all keywords in the database indistinguishable from each other in terms of frequency (i.e., if there are N keywords, then the attacker should not be able to guess the keyword underlying a query with probability better than $1/N$ from the number of matching outcomes). However, this padding strategy is inherently expensive as keywords typically follow a Zipf distribution in real-world databases. To reduce the overhead incurred by padding, we propose a "partial" padding strategy (namely, keyword bucketization) in SWiSSSE, where the attacker is allowed to guess the keyword corresponding to a query with probability higher than $1/N$, but not much better than that (as shown by our cryptanalysis experiments). We experimentally validate that our padding strategy achieves good security-efficiency trade-offs (see Section 6 and Appendix C.4 for experiments evaluating the performance and cryptanalysis-resistance of SWiSSSE for varying bucket sizes and number of buckets).

On a related note, we compare SWiSSSE with PANCAKE [50] – a recently introduced system that protects key-value stores from access pattern leakage attacks using a technique called *frequency smoothing* that transforms plaintext accesses into uniformly distributed encrypted accesses to an encrypted data store. Technically, PANCAKE suppresses access-pattern leakage in encrypted key-value stores by using fake queries and query batching. While SWiSSSE shares a similar leakage-suppression goal, the underlying techniques used, namely keyword bucketization and introduction of fake documents, differ significantly from those used in PANCAKE. We leave it as an interesting open question to design an end-to-end practically efficient SSE system with built-in protections against system-wide leakage cryptanalysis by using the same leakage-suppression techniques as in PANCAKE.

TEE-based Encrypted Search. An alternative line of works [6, 7, 12, 39, 81, 95, 108] has explored the use of trusted execution environments (TEEs)/secure enclaves (such as such as Intel SGX [79]) to design encrypted databases with advanced query capabilities. We note that these solutions crucially rely on the TEE behaving as a "black box" (an assumption that has been challenged by several recent attacks [15, 84, 96, 97, 106, 107]). In addition, while these solutions typically avoid the inefficiencies associated with the usage of cryptographic techniques for encrypted search, they tend to incur high overheads in practice because of the multitude of read and write operations required to fetch the encrypted data into the enclave pre-query execution, and to write back to the database post-query execution. In this paper, we focus on designing end-to-end SSE systems where the security guarantees are derived purely from standard mathematical/cryptographic assumptions, as opposed to the hardware-based assumptions in the TEE-based solutions.

SSE for Alternative Query Types. An alternative line of works investigates SSE schemes supporting point, range and substring queries [34–36, 41], as well as SSE schemes supporting join and group-by queries over encrypted relational databases [34, 60, 64].

We note here that SSE schemes supporting range queries have been cryptanalyzed extensively, notably in [51–53, 67, 74]. Similarly, there exists a large body of work on order-preserving and property-preserving encryption [1, 14, 68, 76, 93, 105] supporting a rich class of SQL queries over encrypted relational databases, many of which have also been broken by leakage-abuse and inference attacks [85]. Our goal in this paper is to design SSE schemes keyword search over encrypted document collections, and we do not consider range (or other classes of) queries in the present version.

Leakage Cryptanalysis in SSE. Starting with the seminal work of Islam et al. [59], leakage cryptanalysis has been studied extensively in the context of SSE for document collections [13, 21, 32, 55, 88, 91, 94]. Some of these works consider somewhat idealized and noise-free leakage profiles (such as the leakage inversion technique in [71]), while others consider noisy access-pattern leakage (such as [88, 91]). Our cryptanalysis of SWiSSSE works with noisy *system-wide* correlation leakage, and can be viewed as a highly refined version of the system-wide leakage cryptanalysis techniques of [54].

2 PRELIMINARIES AND BACKGROUND

In this section, we introduce notations and present preliminary background material.

2.1 Notation

Throughout the paper, λ is the security parameter and $\mathtt{negl}(\lambda)$ denotes a negligible function in λ. If X be a set or list, then we write $x \xleftarrow{\ell}_{\$} X$ to mean uniformly randomly sample ℓ elements from X without replacement. We omit ℓ from the notation when $\ell = 1$.

We note that the keys for cryptographic functions are sometimes omitted from our notation for readability. So, if F is a pseudorandom function we simply write $F(x)$ for the result of applying F to x – how and when the key is sampled will be obvious from context. We use a similar convention for symmetric encryption. For a pseudorandom function F we write $Adv^{PRF,t}_{F,\mathcal{A}}$ for the advantage of an adversary $\mathcal{A}$ in distinguishing between F and a truly random function with at most t queries.

2.2 Key-value Stores

The basic data structure used by our construction is a “key-value store". This data structure implements an associative array abstract data type that maps (non-cryptographic) *keys* to *values*. The data structure supports efficient execution of the following operations:

- **init** : initialises an empty key-value store.
- **get** : takes as input a key k and returns the associated value. We abuse the notation and use **get** for getting the values for a set of keys too.
- **put** : takes as input a key-value pair (k, v) and sets the value associated to k to v. We use **put** for putting a set of key-value pairs too.
- **contains** : takes as input a key k and returns a boolean indicating if k is one of the keys in the key-value store.
- **del**: takes as input a set of keys $\mathcal{I}$ and removes the key-value pairs with the keys in $\mathcal{I}$ from the key-value store.
- **pop**: takes as input a natural number n, selects n uniformly random key-value pairs from the store, removes them from the store, and returns them as the result.

We do not explicitly distinguish between the name of the data structure and its state (e.g., if S is a key-value store we write $S.\mathbf{get}(k)$ for the result returned by $\mathbf{get}(k)$ on the current state of S).

2.3 Databases

Throughout the paper, when we use the term “database”, we refer to a document collection, where each document is tagged with (one or more) keywords. We consider a setting where a client wants to outsource a database $\mathsf{DB} = \{d_i\}$ to a remote server, which it can later query using keywords. We omit a fully formal treatment of unencrypted databases, which is reasonably straightforward. We only make some conventions which are helpful to formalize our results. We write $\mathsf{DB}[i]$ to mean the i-th document in the database. We assume that each document d has a unique document identifier $id(d)$. For simplicity, we assume that the document identifiers run from 0 to $|\mathsf{DB}| - 1$. For each document d we write $W(d)$ for the set of keywords contained in the document d. We write $W\{\mathsf{DB}\}$ for the multiset of keywords in the collection of documents DB. We write $\mathsf{DB}(w)$ to mean the set of documents which contains keyword w.

2.4 System-Wide Definitions for SSE

We introduce new *system-wide* security definitions for (static) SSE schemes in this paper. Our definitions take into account the leakage from *both* the encrypted index and the document retrieval phase during searches. This is in direct contrast to prior *index-only* definitions used extensively in the SSE literature [24, 29, 31] that only take into leakage from the encrypted index component of the overall SSE system. Due to lack of space in the body of the paper, we defer the detailed treatment of our system-wide definition for static SSE to Appendix A. For completeness, we briefly summarize below the system-wide syntax for static SSE (looking ahead, we use the same syntax for our description of static SWiSSSE). An extension of our definitions to dynamic SSE appears in Appendix E.1.

A system-wide static SSE scheme Σ consists of two protocols **Setup** and **KWQuery** between a client and a server. The **Setup** protocol is used by the client to encrypt a plaintext database DB. The client outputs $(\mathsf{EDB}, \mathtt{st}_{\mathtt{Clt}})$, where EDB is an encrypted database (including the encrypted search index as well as the encrypted documents) which is offloaded to the server, while $\mathtt{st}_{\mathtt{Clt}}$ is some internal state that is retained by the client. The **KWQuery** is a client-server protocol between the client and server to support single-keyword queries of the form $q = (op, args)$ where op is the operation of the query, and $args$ are the additional arguments required for the query. The server takes as input the encrypted database EDB. The client and server interact with each other and by the end of the interaction, the client gets $(r, \mathtt{st}_{\mathtt{Clt}})$ where r is the result of the query (i.e., the actual documents matching the query) and $\mathtt{st}_{\mathtt{Clt}}$ is the new state of the client; the server state EDB may also be modified following the execution.

Note that our syntax for static SSE takes into account the actual encrypted documents (as part of the encrypted database EDB output by **Setup** and as part of the query response r output by **KWQuery**) in addition to the encrypted index. See Appendix A for the detailed syntax, as well as the formal definitions of correctness and security.

3 SWISSSE FOR “SIMPLE" STATIC DATABASES

To highlight some of the key techniques underlying SWiSSSE, we first consider a highly “simplified" static database in which each document contains precisely one keyword. We explain how to extend this approach to the general case of arbitrarily many keywords per document in Section 5.

Server Storage. Our SSE scheme offloads the storage of two encrypted data structures to the server − an encrypted *lookup table* that is indexed by the set of keywords in the database, and an encrypted *document array*, which is indexed by the documents. For each keyword, the corresponding entry in the lookup table stores (in encrypted form) pointers to the corresponding entries in the document array. For each document, the document array stores (again in encrypted form) the contents of the document, the list of keywords it contains, and some auxiliary information necessary for searches. Both indices are implemented as key-value stores where keys are calculated using a pseudorandom function, and values are encryptions under a symmetric key owned by the client.

Bucketization. To limit keyword frequency leakage (i.e. in how many documents a keyword appears), we use a frequency bucketization strategy, i.e., we pad the outsourced database with fake occurrences of keywords as well as with additional fake documents. For each entry in the keyword lookup index we store a mix of pointers pointing to “real" and “fake" documents in the document array. Bucketization inherently introduces a tradeoff between security and efficiency, since the work done by the server is now proportional to the “bucket frequency" as opposed to the true frequency of the keyword. We expand on this in Section 4.

Local Stash and Write-backs. If the server-side data structures use static addresses (meaning that a given address in the keyword lookup index and/or the document array always corresponds to a fixed keyword and/or document), then the scheme inherently suffers from “access pattern leakage". To prevent such leakage (and hence the known attacks exploiting it), we ensure that all accesses, even those involving the same keyword, “touch" different parts of the server state. We achieve this through a delayed, pseudorandom write-back technique: after each operation involving a keyword we update the addresses of the keyword and of all documents containing it across the encrypted data structures.

More concretely, we first “locally stash" all the information returned by a server in response to a keyword search query at the client. This information includes the list of documents, the number of times the keyword has been accessed, and the number of times each document containing the keyword has been accessed. Then, at a later time, the client issues a “write-back operation", wherein it flushes out this information from its stash onto the encrypted data structures at the server, using fresh encryptions and to a new, pseudorandomly generated set of addresses. The next time the client issues a search query involving the same keyword, the server will access the new set of addresses in both the lookup index and the document array, and will observe only the fresh encryptions of the same (or updated) content.

Note that addresses as described above correspond to keys in the server’s key-value stores. The client need only assume that the server provides a correct implementation of the key-value store and to avoid using colliding addresses/keys. We defer a formal description of the stashing and write-back procedure to Section 5. Note also that, intuitively, a larger client stash leads to a larger storage requirement on the client, but lowers the frequency with which the client needs to flush its stash and trigger write-back operations. This allows flexibility in trading-off client storage with bandwidth requirements.

Search Queries. A search query involving a keyword w proceeds as follows. The client first checks its private stash to see if the transcript of a previous query involving w already exists in its stash. If yes, it directly obtains the search result from the stash and does not initiate any further interaction with the server. Otherwise, the client sends a search token to the server and receives back the entry corresponding to w in the encrypted lookup index. Next, the client decrypts the entry received from the server and retrieves the list of pointers to addresses in the encrypted document array. It sends this list to the server. The server retrieves the corresponding entries from the encrypted document array and sends these back to the client. Upon receiving the encrypted entries from the server, the client decrypts them, discards any fake documents and retains the real ones. Finally, the client caches the whole transcript of the search operation in its local stash, so that its contents may be written-back at a later point in time.

4 BUCKETIZATION IN SWISSSE

As mentioned in Section 3, we mitigate volume leakage in SWiSSSE through a frequency bucketization strategy over the set of keywords in the document collection. At a high level, this entails dividing the set of all keywords across multiple buckets such that each keyword in the same bucket is “padded" to have the same frequency as the most frequent keyword in that bucket (also referred to as “bucket frequency"). The core aim of bucketization is to prevent generic volume/frequency leakage-based attacks on SSE [13, 21, 22, 59, 94].

Padding. Bucketization of keywords requires *padding* the original database with fake data. Our first padding strategy is to add “fake occurrences" of each keyword across the existing documents, such that the keyword frequencies grow to the desired bucket frequency. This approach incurs only moderate additional storage overheads at the server since the number of documents remains the same, and only the encrypted lookup index grows in size.

Note that we *do not* perform “worst-case padding" where every keyword is padded to have the same frequency. Although theoretically free of volume leakage, worst-case padding is extremely inefficient and either imposes a linear search overhead [63] or a quadratic storage complexity, which we wish to avoid. Hence we opt for strategies that, while theoretically less secure, offer significantly better leakage versus efficiency tradeoffs in practice.

Bucket Sizes. We propose bucketing keywords using buckets of two different sizes − B_{high} for the keywords with higher frequencies, and B_{low} for the keywords with lower frequencies. We pad the database with fake data so as to equalise the frequency of all keywords within a given bucket. Hence, the adversary can guess the queried keyword with probability at most $1/B$ for $B \in \{B_{\text{high}}, B_{\text{low}}\}$.

The tradeoff here is that a larger B (i.e. larger but fewer buckets) enjoys lower leakage but incurs more padding and so greater search overheads. Note that the guessing probability of the adversary is independent of the *real* frequency of the queried keyword.

Our bucketization strategy is fundamentally motivated by the fact that most real-world document collections follow a Zipf power distribution [77], wherein frequency of ranked keywords decays inversely. For document collections that follow a Zipf distribution, we recommend a bucketization strategy where only a small fraction of the buckets have size B_{high}, while most of the buckets have size B_{low}. This yields low storage overheads and good search efficiency in practice. Concretely, for SWiSSSE, we propose using $B_{\text{high}} = 400$ and $B_{\text{low}} = 200$. In its most general form, SWiSSSE is capable of handling any arbitrary vector of bucket sizes to partition the keywords. We choose two distinct bucket sizes because our experiments over real-world databases indicate that this yields a good tradeoff between efficiency and security in practice.

We refer to Section 6.2 (Table 1) for an empirical evaluation of the tradeoffs offered by a variety of bucketization strategies (in particular, by varying the number of buckets and the size of each bucket). Additionally, Appendix C empirically evaluates the security offered by our bucketization strategy against co-occurrence leakage-based cryptanalysis over the Enron email corpus and the English Wikipedia.[7] Our experiments over these real-world databases show that bucket sizes $B_{\text{high}} = 400$ and $B_{\text{low}} = 200$ suffice to thwart not only all existing leakage-based attacks [13, 21, 59], but also highly refined system-wide leakage cryptanalysis techniques [54], while retaining practically efficient searches over these real-world databases.

5 SWISSSE FOR GENERAL STATIC DATABASES

We now formally describe SWiSSSE for general static databases.

5.1 Changes from the Simplified Case

Auxiliary Write-Backs. At a high level, we opt for the following strategy in the case of general databases: whenever a document d_ℓ is scheduled to be flushed from the client's stash and written back to the encrypted document array, the client additionally schedules an *auxiliary* write-back for each keyword w_i occurring in d_ℓ.

To understand why auxiliary write-backs ensure search correctness, consider the following three scenarios:

(1) Suppose that a query on w_i is issued before d_ℓ is written back. At this point, the client can directly retrieve d_ℓ from its private stash.
(2) Next, suppose that a query on w_i is issued after d_ℓ has been written back but before the auxiliary write-back for w_i is executed. In this case, the pointer in the lookup index is invalid, but client can refer to its stash to check if an auxiliary write-back for w_i is scheduled. This allows it to recover the new pointer and use it for the search operation.
(3) Finally, suppose that a query on w_i is issued after the auxiliary write-back for w_i has been executed. This again does not affect search correctness because the entry for w_i in the keyword lookup index now points to the "new" address for d_ℓ in the document array.

[7] http://kopiwiki.dsd.sztaki.hu/

Moreover, these write-backs impose no extra bookkeeping overhead at the client, since they do not need to be executed in sync with the original write-back for the document.

Restructuring the Keyword Lookup Index. To support auxiliary write-backs, we restructure the keyword lookup index. For simplicity of presentation, we present a simplified version of the restructuring. This incurs some undesirable leakage, which we address subsequently.

Instead of storing a single entry for each keyword w_i, we now store an entry for each keyword-document pair (w_i, d_ℓ) such that d_ℓ contains w_i. The address for this entry is generated as $F(K, w_i||j||\mathsf{cnt}_{w_i})$, where F is a PRF with key K, j is a counter that runs from 0 to $|\mathsf{DB}(w_i)| - 1$ (where $\mathsf{DB}(w_i)$ denotes the set of documents containing keyword w_i), and cnt_{w_i} is a per-key word counter held in the client's stash which records how many times w_i has appeared in search queries. Each entry stores a ciphertext encrypting a *single* pointer to the address of some d_ℓ in the document array.

In keeping with our "frequency bucketization strategy", we also create and store in the lookup index additional "fake" entries of the form $(w_i, \widetilde{d_\ell})$, where $\widetilde{d_\ell}$ is a fake document. The address for each such fake entry is generated as $F(K, w_i||j'||\mathsf{cnt}_{w_i})$, where j' is a counter that runs from $|\mathsf{DB}(w_i)|$ to one less than the bucket size for w_i. Each such entry again stores a single ciphertext, now encrypting a pointer from w_i to the fake document $\widetilde{d_\ell}$.

This makes the lookup index amenable to auxiliary write-backs. In particular, the *auxiliary* write-back for a keyword w_i occurring in d_ℓ targets the address $F(K, w_i||j||\mathsf{cnt}_{w_i})$, and updates specifically the pointer from w_i to d_ℓ.

Auxiliary Addresses for Auxiliary Write-Backs. To prevent the server from correlating auxiliary write-backs to the last normal write-back involving the same keyword, we choose to dissociate the two sets of addresses. More concretely, we generate separate sets of addresses for normal and auxiliary write-backs involving the same keyword:

$$\mathsf{addr}_{\text{norm}}(w_i, d_\ell, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(2 * \mathsf{cnt}_{w_i})),$$
$$\mathsf{addr}_{\text{aux}}(w_i, d_\ell, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(2 * \mathsf{cnt}_{w_i} + 1)),$$

where j is again a counter that runs from 0 to $|\mathsf{DB}(w_i)| - 1$ and cnt_{w_i} is again the per-key word counter held in the client's stash which records how many times w_i has appeared in search queries. Similarly, for the fake documents associated with w_i, we generate separate sets of addresses for normal and auxiliary write-backs:

$$\mathsf{addr}_{\text{norm}}(w_i, \widetilde{d_\ell}, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(2 * \mathsf{cnt}_{w_i})),$$
$$\mathsf{addr}_{\text{aux}}(w_i, \widetilde{d_\ell}, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(2 * \mathsf{cnt}_{w_i} + 1)),$$

where j' is a counter that runs from $|\mathsf{DB}(w_i)|$ to one less than the bucket size for the keyword w_i.

Search Correctness. To ensure that a search query on the keyword w_i correctly takes into account all auxiliary write-backs involving w_i, the client now requests the server to access both sets of write-back addresses for w_i – normal and auxiliary – in the keyword lookup index. If both sets of addresses exist for a particular document, the client uses the pointer stored in the auxiliary write-back address; otherwise it uses the pointer stored in the normal write-back address.

Remark 5.1. For ease of understanding, we explain the distinction between normal and auxiliary write-backs in SWiSSSE using an example. Suppose that the client searches for keyword w and there is only one document d (stored in address addr in the encrypted document array) containing w. The search protocol in SWiSSSE has two stages. In the first stage, the client looks for the address of the document from the search index using a token computed from keyword w. In the second stage, the address can be used to retrieve the actual document. Note that in the write-back procedure, our goal is to flush document d to a completely new address addr′ in the encrypted document array. However, to retrieve the document d in future queries, we also need to update the search index to point to this new address addr′ in the encrypted document array. We summarize this distinction between normal and auxiliary write-backs below.

- **Normal write-back:** Updates the search index for the keyword w that has just been searched for. The entry corresponding to the searched keyword w in the search index now points to the new address addr′ after the update.
- **Auxiliary write-back:** A document d containing the keyword w may contain other keywords. Suppose, for simplicity, that d has one other keyword w'. We also need to update the address associated with the keyword w' in the search index to maintain consistency during future searches. Auxiliary updates help to achieve exactly that.

Remark 5.2. In SWiSSSE, post write-backs, the documents that are still stored in the old addresses can be safely deleted (since the encrypted lookup index consistently points to the latest updated address of a document in the encrypted document array). Since such deletions can be done periodically and the old addresses can be reused for future search operations, SWiSSSE does not suffer from ever-growing storage.

5.2 The Setup Procedure of SWiSSSE

Algorithm 1 describes SWiSSSE.**Setup** – the setup procedure of SWiSSSE. The description uses a symmetric encryption scheme ($\mathbf{KGen}_1, \mathbf{Enc}, \mathbf{Dec}$) and a pseudorandom function F with key generation algorithm $\mathbf{KGen}_2$. For readability, we omit explicitly describing the keys used by these cryptographic functions.

The server stores an encrypted lookup table Svr.EI (to store the map between the keywords and the document addresses in the encrypted form) and an encrypted document array Svr.EA (to store the map between the document addresses and the actual documents in encrypted form). The addressing mechanism and encrypted contents for these data structures are as described above.

Similarly, the client stores in its local stash a plaintext lookup index Clt.I (to store the map between the keywords and where they are in the encrypted document array before they are written back to the server), and a plaintext document array Clt.A (to store the map between the document identifiers and the documents). In particular, the plaintext document array is used to store the documents retrieved from the previous queries and randomly select an appropriate number of documents for auxiliary write-backs.

To implement the keyword bucketization strategy, the client creates a "padded version" DB′ of the original database DB before encrypting and offloading it to the server. More concretely, the

Algorithm 1 SWiSSSE.**Setup**

procedure Clt.Setup(1^λ, DB)
  /* *Key generation* */
  Clt.$sk_1 \leftarrow \mathbf{KGen}_1(1^\lambda)$
  Clt.$sk_2 \leftarrow \mathbf{KGen}_2(1^\lambda)$
  Generate map $\mathsf{G} : W \rightarrow \mathbb{N}$
  Clt.G $\leftarrow$ G
  /* *Generate fake documents* */
  DB′ $\leftarrow$ **Fake_Doc_Gen**(DB, Clt.G)
  Clt.$N \leftarrow |\mathsf{DB}'|$
  /* *Clt.N allows the client to locally maintain the size of the padded database. This is used subsequently in the keyword search algorithm.* */
  EI, EA $\leftarrow \{\}$
  for $i \in 1, \ldots, |\mathsf{DB}'|$ **do**
    /* *Get the set of keywords with counters* */
    $x \leftarrow \{(w, \mathsf{Clt.KWCtr}[w]) \mid w \in W(\mathsf{DB}'[i])\}$
    /* *Update the lookup index* */
    /* *In the following expression, $W(\mathsf{DB}'[i])$ denotes the set of keywords in the i-th document of the padded database.* */
    for $w \in W(\mathsf{DB}'[i])$ **do**
      $j \leftarrow \mathsf{Clt.KWCtr}[w]$
      /* *Insert the new lookup entry in the search index* EI */
      $\mathsf{EI} \leftarrow \mathsf{EI} \cup (F(w||j||0), \mathbf{Enc}(id(\mathsf{DB}'[i])))$
      /* *Note that the zero at the end of the PRF input essentially indicates that the number of search operations involving w is initially* 0 *at setup* */
      $\mathsf{Clt.KWCtr}[w] \leftarrow \mathsf{Clt.KWCtr}[w] + 1$
    /* *Insert the encrypted document in* EA */
    $\mathsf{EA} \leftarrow \mathsf{EA} \cup (F(i||0), \mathbf{Enc}(x||\mathsf{DB}'[i]))$
  /* *Reset the keyword counter* */
  for $w \in W(\mathsf{DB}')$ **do**
    $\mathsf{Clt.KWCtr}[w] \leftarrow 0$
  /* *Initialise the stash* */
  Clt.I.**init**()
  Clt.A.**init**()
  Send (EI, EA) to the server

procedure Svr.Setup(EI, EA)
  Svr.EI.**init**()
  Svr.EA.**init**()
  Svr.EI.**put**(EI)
  Svr.EA.**put**(EA)

client selects a map $\mathsf{G} : W \rightarrow \mathbb{N}$. This map assigns each keyword to a bucket, such that all keywords in the same bucket have the same frequency in the padded database DB′.

Let $|\mathsf{DB}(w)|$ and $|\mathsf{DB}'(w)|$ denote the frequency of the keyword w in the original and padded databases, respectively. The map G allows the client to determine a padding procedure **Fake_Doc_Gen**; the procedure pads the input database DB with "fake" documents to obtain a padded version DB′ such that $|\mathsf{DB}'(w)| = \mathsf{G}(w)$ for each keyword w. The client may use any padding strategy to achieve the desired keyword frequencies as specified by the function G.

Additionally, to suppress volume leakage, the document addresses and the document contents are padded to fixed lengths ℓ_0 and ℓ_1 respectively (both assumed to be public parameters) prior to encryption. Note, however, that we do not perform worst-case document padding, which would potentially incur huge storage

Algorithm 2 SWiSSSE.**KWQuery**: Address Retrieval Sub-Routine

procedure Clt.TokenGen(w)
  $L \leftarrow \{\}$
  /* *Generate all lookup addresses for keyword w and store them in L* */
  for $j \in 0, \ldots, \texttt{Clt.G}(w) - 1$ **do**
    $L \leftarrow L \cup \{F(w||j||2 * \texttt{Clt.KWCtr}[w])\}$
    $L \leftarrow L \cup \{F(w||j||2 * \texttt{Clt.KWCtr}[w] + 1)\}$
  /* *Roll forward the counter corresponding to w in preparation for the next query on w* */
  $\texttt{Clt.KWCtr}[w] \leftarrow \texttt{Clt.KWCtr}[w] + 1$
  Send L to the server

procedure Svr.Index_Lookup(L)
  Send Svr.EI.**get**(L) to the client

Algorithm 3 SWiSSSE.**KWQuery**: Encrypted Document Retrieval Sub-Routine

/* *EL is a list containing all encrypted document addresses retrieved at the end of Algorithm 2* */
procedure Clt.Document_Retrieval(w, EL)
  $L \leftarrow$ the latest addresses of the keywords from **Dec**(EL) if they are not in Clt.I
  Add random document identifiers between 0 and Clt.$N - 1$ that are not in the stash to L until $|L| = 2 \cdot \texttt{Clt.G}(w)$
  $M \leftarrow \{\}$
  for $id \in L$ **do**
    /* *Compute the document addresses and collect them in M* */
    $M \leftarrow M \cup F(id||\texttt{Clt.ArrCtr}[id])$
    /* *Increase the counters* */
    $\texttt{Clt.ArrCtr}[id] \leftarrow \texttt{Clt.ArrCtr}[id] + 1$
  Send M to the server

procedure Svr.Document_Retrieval(M)
  Send Svr.EA.**get**(M) to the client

and communication overheads. Instead we use a fragmentation strategy where each document is fragmented into sub-documents of size ℓ_1; so we only really perform padding for the last fragment in case it has size less than ℓ_1. We avoid these details in Algorithm 1 for the sake of readability.

5.3 The KWQuery Procedure of SWiSSSE

We now describe SWiSSSE.**KWQuery** – the keyword query procedure for SWiSSSE. For ease of presentation, SWiSSSE.**KWQuery** is broken up into three sub-routines described in Algorithms 2, 3 and 4. Again, for readability, we omit explicitly describing the keys used by the cryptographic functions in these algorithms.

These algorithms formally depict the following steps taken by the client during a keyword query:

- **Algorithm 2:** The client generates a search token to look up both the normal and auxiliary write-back addresses for w_i in the encrypted keyword lookup index, receives the corresponding encrypted entries from the server, and decrypts the results locally to identify the relevant entries in the encrypted document array. It also updates the counter keeping track of the number of search operations involving w_i (this helps generate the new write-back address for w_i in the encrypted keyword lookup index).
- **Algorithm 3:** The client then generates a search token and retrieves the corresponding encrypted documents from the document array at the server, and decrypts the results locally to filter out the fake documents. For each accessed document, the client updates the local counter keeping track of the number of times the document has been addressed (this helps generate the new write-back address for w_i in the encrypted document array).
- **Algorithm 4:** Finally, the client updates its local stash with the documents retrieved in the previous step. These will be written back to the encrypted document array at the server via normal auxiliary write-backs at a later point of time. Each write-back operation involves the client randomly sampling a certain proportion of the documents and the lookup indices from its local stash (created over multiple search operations in the past), and writing them back to the corresponding encrypted data structures at the server. For the specific instance described in Algorithm 4, this fraction is set to one-half; we justify the choice of the parameter in Section 6.2.

Algorithm 4 SWiSSSE.**KWQuery**: Auxiliary Write-Back Sub-Routine

/* *EM is a list containing all encrypted documents retrieved at the end of Algorithm 3* */
procedure Clt.Write_Back(EM, $\bar{w}$)
  $UA \leftarrow \{\}$
  /* *Get random documents from the stash, b_i is a bit to indicate if w_i was the leading keyword* */
  $D \leftarrow \texttt{Clt.A.}\textbf{pop}(\lfloor|\texttt{Clt.A}|/2\rfloor)$
  for $(\{(w_i, j_i, b_i)\}, d) \in D$ **do**
    /* *Encrypt the new documents* */
    $UA \leftarrow UA \cup \{(F(id(d)||\texttt{Clt.ArrCtr}[id(d)]), \textbf{Enc}(\{(w_i, j_i)\}||d))\}$
    /* *Update the stash for the lookup index* */
    for $(w, j, b) \in \{(w_i, j_i, b_i)\}$ **do**
      $\texttt{Clt.I.}\textbf{put}((F(w||j||2*\texttt{Clt.KWCtr}[w]+b), \textbf{Enc}(id(d))))$
  /* *Decrypt the documents retrieved at the end of Algorithm 3 and insert them into the document array* */
  Clt.A.**put**(**Dec**(EM))
  Send $(\texttt{Clt.I.}\textbf{pop}(\lfloor|\texttt{Clt.I}|/2\rfloor), UA)$

procedure Svr.Write_Back((UI, UA))
  Svr.EI.**put**(UI)
  Svr.EA.**put**(UA)

5.4 System-Wide Correctness

We state and prove the following theorem for the system-wide correctness of SWiSSSE.

Theorem 5.1 (**Correctness of** SWiSSSE). *Let* $|\mathsf{DB}|$ *and* $|W\{\mathsf{DB}\}|$ *denote the total number of documents and document-keyword pairs, respectively, in the database* DB, *and let* l *denote the output length of the PRF* F *used in SWiSSSE. Then the advantage of any adversary* $\mathcal{A}$, *which issues at most* k *search queries, in breaking the system-wide correctness of SWiSSSE over the database* DB *is at* most:

$$\frac{\left(|\mathsf{DB}|^2 + 4t_0|\mathsf{DB}| + |W\{\mathsf{DB}\}|^2 + 4t_1|W\{\mathsf{DB}\}|\right)}{2^{l+1}}$$
$$+Adv^{PRF,|\mathsf{DB}|+2t_0}_{F,\mathcal{B}} + Adv^{PRF,|W\{\mathsf{DB}\}|+2t_1}_{F,C},$$

where $t_0 = k \cdot \max_w |\mathsf{DB}(w)|$, $t_1 = k \cdot \max_w |w\{\mathsf{DB}(w)\}|$, and $\mathcal{B}$ and C denote probabilistic polynomial-time adversaries in independent security experiments against the PRF F.

Due to lack of space in the body of the paper, we defer a formal proof of this theorem to Appendix B.1.

5.5 System-Wide Leakage of SWiSSSE

We now formally describe the leakage profile for SWiSSSE with respect to static databases. Following the approach introduced by previous works on SSE (such as [31] and [24]), we use a simulation-based framework where a PPT adversary is required to distinguish between the real world (where the adversary interacts with a real execution of SWiSSSE that uses the secret key) and the ideal world (where the adversary interacts with a simulator that only has access to the described leakage profile for SWiSSSE). The enumeration is provably sound if no PPT adversary can distinguish between these two worlds with non-negligible advantage over a random guess.

Unlike most prior SSE constructions, the leakage function for SWiSSSE is stateful. This follows naturally from the fact that the search protocol in SWiSSSE makes abundant usage of random address accesses across the encrypted data structures at the server. We use a stateful definition to capture the leakage from such random accesses.

Leakage at Setup. As summarized in Section 5, at setup, the client offloads the encrypted lookup index and the encrypted document array to the server. These data structures are essentially key-value stores with pseudorandomly generated keys/addresses and values/entries that are encrypted under an IND-CPA secure encryption scheme. Hence, at setup, the server learns no information about the original database DB other than the number of documents in the padded database DB′ (including both real and fake documents), and the total number of keyword-document pairs post-bucketization. Formally, we have:

$$\mathcal{L}^{\mathbf{Setup}}_{\Sigma}(\mathsf{DB},\mathsf{G}) = (|\mathsf{DB}'|, |W\{\mathsf{DB}'\}|, \mathbf{St}_{\mathcal{L}}),$$

where Σ denotes a concrete instance of SWiSSSE. Note that the leakage function is stateful; it maintains in $\mathbf{St}_{\mathcal{L}}$ a realisation of the padded database (used later by the leakage function for searches).

Leakage during Searches and Write-Backs. As summarized in Section 5, each search query leaks the following to the server:

(1) The set of normal and auxiliary write-back addresses in the encrypted keyword lookup index corresponding to the queried keyword (but not precisely which of these are normal and which of these are auxiliary).
(2) The set of addresses in the encrypted document array for the real and fake documents containing the queried keyword (but not precisely the document identifiers, or even which of the addresses correspond to real documents and the fake documents, respectively).

Similarly, each write-back operation reveals to the server the set of addresses in the encrypted keyword lookup index and the encrypted document array that are written to using content from the stash.

We capture this leakage using a probabilistic and stateful leakage function $\mathcal{L}^{\mathbf{KWQuery}}_{\Sigma}$. Due to lack of space in the body of the paper, we defer the formal description of the leakage profile to Algorithm 5 (for an instance Σ of the SWiSSSE scheme) in Appendix B.2. The state of the leakage function contains a realisation of the padded database, everything in the stash of the client, and two data structures, namely the lookup history **IndHist** and the array-access history **ArrHist**, which we describe in Appendix 5.

The following theorem, which we prove in Appendix B.3, formalizes the security of SWiSSSE.

Theorem 5.2 (**Security of** SWiSSSE). *Let $\mathcal{L}^{Setup}_{\Sigma}$ and $\mathcal{L}^{KWQuery}_{\Sigma}$ be the leakage functions defined above. The instance Σ of* SWiSSSE *is $\left(\mathcal{L}^{Setup}_{\Sigma}, \mathcal{L}^{KWQuery}_{\Sigma}\right)$-secure.*

We give a more detailed discussion on the implications of the leakage in Appendix C.1.

5.6 System-Wide Leakage Cryptanalysis

We report in Appendix C a detailed cryptanalysis of the system-wide leakage of SWiSSSE on the Enron email corpus and the English Wikipedia dump. In particular, we show that for appropriate choices of bucket sizes (more precisely, 400 for the most frequent keywords and 200 for the remaining keywords), SWiSSSE is resilient to traditional leakage-abuse attack techniques [13, 21, 59, 88], as well as the system-wide leakage-based query recovery attacks proposed recently in [54]. The authors of [54] used their attacks to break all end-to-end SSE systems built in a natural way from a vast majority of the index-only SSE schemes in the literature [16, 19, 23, 24, 26, 29, 31, 41, 62, 64, 66, 86, 101], including those applying leakage-suppression techniques such as volume-hiding EMMs [65, 92] to the search index. To the best of our knowledge, SWiSSSE is the only end-to-end SSE system to be secure against these system-wide leakage-abuse attacks while supporting practically efficient searches. More details can be found in Appendix C.

6 EXPERIMENTAL EVALUATION

In this section, we describe a prototype implementation of SWiSSSE and report on its performance. We also present a detailed experimental comparison between the query performance of state-of-the-art SSE schemes with security against system-wide leakage, and the query performance of SWiSSSE. A theoretical performance evaluation of SWiSSSE can be found in Appendix D.

Overview of Experiments. As target database we chose the Enron email corpus. It contains over 500K emails and over 30M keyword-document pairs which makes it a perfect database to experiment with the scalability of our SSE scheme. We run experiments on sub-databases of different sizes ranging from 10K documents to 400K documents.

6.1 Experimental Setup

Choice of Primitives. We instantiate the PRF with HMAC-SHA-256 [72]. Only the first 16 bytes of the output are used as keys to reduce storage. We use AES-GCM [78, 87] for symmetric-key encryption.

Implementation. We implement the client and server in Java [104], using the Java Cryptography Extension[8] as the underlying cryptographic library. We choose to use a single-thread implementation as it provides the most accurate measurements of performance. The server we implemented serves as a proxy between the client and the actual storage system. It is responsible for translating the queries into standard key-value store queries. We used Redis[9] as the underlying key-value store. For comparison, we also implement a plaintext database in Java. The database uses an inverted index for fast lookup, where the keys are the keywords, and the values are lists of document identifiers associated to the keywords.

Document Preprocessing. To prepare the documents for insertion, we extract keywords from them with the Natural Language Toolkit.[10] The English stop words and the keywords with frequency higher than 5% are removed from the set of keywords for each email. Emails that are larger than 1KB are partitioned into chunks of 1KB, taking care to associate all the keywords from a given email with each chunk.

Experimental Environment. We run our experiments on an AMD Ryzen 9 5900X CPU clocked at 3.7 GHz (4.8 GHz boost clock) and 32 GB DDR4 memory clocked at 2400 MHz. For simplicity, the server and client are run on the same machine (so our results do not take into account network latency).

6.2 Parameter Selection for SWiSSSE

There are three tunable parameters in SWiSSSE that affect its security and performance. These parameters are: (1) the size of the buckets, (2) the fraction of documents written back in the write-back step, and (3) the fraction of lookup indices written back in the write-back step.

In this section, we investigate the relationship between the three parameters above and the performance and security of SWiSSSE experimentally. We give our recommended parameters at the end of the section.

Size of buckets. We perform thorough performance and cryptanalysis experiments to investigate the trade-offs between different bucketization strategies. These experiments are performed on the Enron email corpus and the write-back rate for the lookup indices and the documents are set to 50%. A full description of our cryptanalysis techniques and experimental results can be found in Appendix C.3. Table 1 summarises our experimental results. The query recovery rate shown in the table are for the most frequent keywords only.

It can be seen that bucket sizes smaller than 400 do not offer enough resilience against our new leakage cryptanalysis on the most frequent keywords. On the other hand, using a bucket size larger than 400 will incur significant overheads in terms of storage and communication (Figure 3 in Appendix C.3).

[8]https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
[9]https://redis.io/
[10]https://www.nltk.org/

Bucket size (# buckets)	Storage overhead (index)	Communication overhead	Query Recovery Rate
50 (668)	3.9%	0.9%	98.7%
100 (334)	8.1%	1.7%	84.9%
200 (167)	17.0%	3.6%	56.3%
400 (84)	36.1%	7.9%	26.5%

Table 1: Performance and cryptanalysis experiments on the size of the buckets.

Write-back rate. In this section, we investigate alternative parameters for the fraction of documents written back and the fraction of lookup indices written back in the write-back step. We note that for any write-back rate r, r document write-backs and r lookup index write-backs produce equivalent leakage (since the leakage comes retrieving documents/indices that have been just written back). Therefore, we focus on parameter selection for the fraction of documents.

We perform performance experiments and cryptanalysis experiments with 50% to 90% lookup indices and documents write-back (in steps of 10%). The cryptanalysis experiments (see Appendix C.3) are performed on the Enron email corpus only and uses bucket size of 400. We report the experimental results in Table 2.

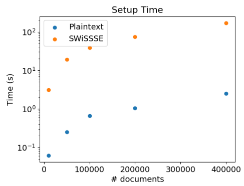
Write-back	Mean Write-back time (ms)	Mean Stash Size (MB)	Query Recovery Rate
50%	1571	0.978	26.5%
60%	1534	0.647	86.3%
70%	1584	0.422	91.4%
80%	1671	0.269	93.9%
90%	1690	0.179	94.5%

Table 2: Performance and cryptanalysis experiments on the write-back rate.

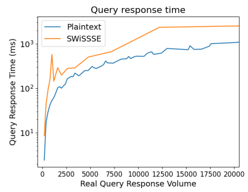
We observe that having a higher write-back rate does not help with the write-back time. This is because a higher write-back rate means more lookup indices and documents have to be written back in the write-back step. On the other hand, the mean stash size decreases significantly as the write-back rate increases, as expected. In terms of resilience to query reconstruction attacks, we observe that SWiSSSE is a lot more vulnerable to our highly refined leakage-abuse attack if we allow for a higher write-back rate. In particular, as soon as the write-back rate is set to 60%, our attack is able to recover 86.3% of the queries.

Recommended Parameters. We give our recommended parameters in this section. These parameters are used in the performance experiments in Section 6.3. We choose a bucket size of 400 for the most frequent 2400 keywords. For the other keywords, we find that a bucket size of 200 provides reasonable security and efficiency. A more detailed discussion of our bucketization strategy can be found in Appendix C.4. As for the write-back rate, we set it to 50%.

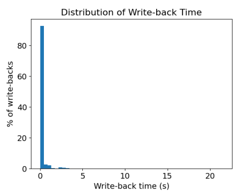
Remark 6.1. For the bucket size of 400 and a write-back rate of 50%, our highly refined system-wide query recovery attacks achieve a query recovery rate of 26.5% (the identical figures in Tables 1 and 2 result from using an identical experimental parameters in both cases). While this appears to be a concern at first glance, we point out that our leakage analysis is rather "pessimistic" and assumes



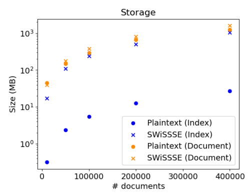
(a) Setup time of the plaintext database and SWiSSSE **(log scale).**



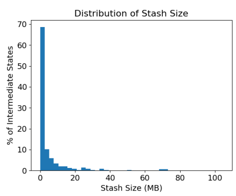
(b) Query response time of the plaintext database and SWiSSSE **on 400K documents (log scale).**



(c) Distribution of write-back time of SWiSSSE **on 400K documents.**



(d) Storage required by the plaintext database and SWiSSSE **(log scale).**



(e) Stash size of SWiSSSE **on 400K documents.**

Figure 1: Performance comparison between the plaintext database and SWiSSSE**.**

far more leakage than is actually leaked by a real implementation of SWiSSSE in practice. Hence, we recommend that using these parameters is safe from SWiSSSE in practical implementations. See Appendix C.3 for additional discussion.

6.3 Benchmarks

Setup Time. Figure 1a shows the setup time of the plaintext database and SWiSSSE. SWiSSSE is two orders of magnitude slower than a plaintext implementation which is expected due to its extensive use of encryption.

Query Response Time. Figure 1b shows the query response time of the plaintext database and SWiSSSE for the experiment with 400K documents. In each experiment, 1000 uniformly randomly picked keywords are queried. Here, real query response volume refers to the actual number of documents associated to the keywords and query response time is defined to be the time from the start of a query to the point of time for which the client obtains the plaintext documents. SWiSSSE is about 2-4 times slower than a plaintext database depending on the real frequency of the queried keyword and its padded frequency. This can be attributed to several factors. Firstly, SWiSSSE deploys a bucketization strategy which leads to more documents being retrieved than the real query response volume. Secondly, SWiSSSE uses the duplication technique on the inverted index, which means the amount of time required for index retrieval become linear in the bucket size of the queried keyword as opposed to a single query for the plaintext database. Finally, SWiSSSE has to search the stash to retrieve locally stored documents, and perform cryptographic operations.

We note that the query response time of SWiSSSE can be improved significantly with parallelisation and multi-threading. For example, the computation of search keys and decryption of documents can be parallelised. Computation of search tokens and decryption of documents can be separated from interactions with the server using different threads to reduce unnecessary blocking time between different commands.

Write-back Efficiency. We report write-back efficiency for the experiment with 400K documents in Figure 1c. As write-back time depends on the number of documents retrieved in previous queries, reporting an average value is not very informative. Here, we choose to show the distribution of write-back time for 1000 uniformly randomly distributed queries over the set of keywords. We observe that over 80% of the write-backs are completed in under 1 second; very long write-back times arise only occasionally.

The major bottle-neck of the write-back operation (over 70% of the execution time) comes from inserting the key-value pairs into the Redis database – a significant number of key-value pairs needs to be inserted in every write-back operation due to the use of duplication on the encrypted search index.

In practice, the client can simply transfer all the key-value pairs it wants to update and go offline; the server (with the help of the proxy if needed) can then insert these key-value pairs on its own. This yields a 5× reduction of the effective write-back time.

Server Storage. Storage of the plaintext database compared to SWiSSSE is reported in Figure 1d. The main source of overhead for SWiSSSE comes from the inverted index as can be seen clearly from the graph. This is because SWiSSSE uses duplication and many more keys need to be created for the inverted index. On the other hand, the overhead for document storage is minimal.

The Client Stash. The distribution of the stash size is shown in Figure 1e. The stash size was kept under 1 MB for over 80% of the queries. There were several occasions when the stash size grew to over 10 MB. This is due to queries on keywords with high frequencies. These can be expected to be rare in practice for typical query distributions. Furthermore, as half of the documents are written back to the server after each query, the stash size will only be high for a few queries. Therefore, we expect the stash size to stay reasonably small on the client most of the time. On the rare occasions where the stash size becomes large, the client can use data compression or can issue dummy queries on low frequency keywords causing the stash size to reduce more quickly.

We experimentally validate the data compression technique we proposed and report the following performance metrics. The compression algorithm used in our experiment is GZIP at compression level 6. In terms of query response time, the compression technique results in an 17.9% overhead (due to compression). In terms of the write-back time, the compression technique results in a 4.3% overhead (due to decompression). On the other hand, in terms of the size of the stash for the documents, the compression technique saves 27.6% of the storage. We recommend using data compression when the stash size is high.

Scheme	Storage (Client)	Storage (Server)	Computation (Client)	Communication (C→S S→C)
SWiSSSE	23.3 MB	1.4 GB	2.1K PRF 2.1K DEC	16 KB 1.0 MB
Duplication	-	42 GB (31×)	70K PRF 70K DEC	1.1 MB (67×) 68 MB (67×)
PRT-EMM [65]	-	1.1TB (830×)	35K PRF 35K DEC	0.53 MB (34×) 34 MB (34×)
VH-EMM [92]	-	110 GB (80×)	140K PRF 140K DEC	2.1 MB (130×) 140 MB (130×)
SealPIR [5]*	-	340 GB (250×)	70K ENC 70K DEC	4.3 GB (270, 000×) 17 GB (17, 000×)
Non-recursive Path ORAM [102]	42.7 MB	5.3 GB (3.9×)	1.5M ENC 1.5M DEC	3.0 GB (190, 000×) 3.0 GB (2, 900×)

Table 3: Comparison of different document retrieval techniques. In the experiments, we used $G = 522$ as the query response volume (mean keyword frequency in the Enron email corpus) for calculating the overheads. As all of the schemes except SWiSSSE require full padding, only the performance numbers of SWiSSSE will be affected by the query response volume. The numbers in the brackets indicate overheads beyond the baseline provided by SWiSSSE. *SealPIR also requires the server to perform 390 billion FHE operations per query. The client storage we report does not include cryptographic keys as they do not grow with the size of the database.

Optimizations and Extensions. While our experiments already demonstrate that SWiSSSE scales well in practice to very large databases, an implementation in a low-level language (such as C) should improve efficiency further. Another potential optimization is to switch to a specialised PRF such as SipHash [11] in place of HMAC-SHA-256. A larger client stash would also allow batching the write-backs together to further improve efficiency. We conducted our experiments locally rather than over a network; further work is needed to characterise the impact of network latency on query response times. However, recall that each search operation only involves two round trips. In addition, latency from communication over a WAN would be on the order of 50 milliseconds [20] and it is additive to the query processing time, which is typically on the order of 1k-10k milliseconds. Hence, network latency does not have a large impact on the performance.

6.4 Comparison to State-of-the-Art SSE Schemes

We now compare the performance of SWiSSSE with the state-of-the-art SSE schemes if they were made system-wide secure.[11] For ease of comparison, we ignore the index retrieval phase and focus on the document retrieval phase only (the cost of index retrieval is only a small fraction of the cost of document retrieval for all of the schemes under consideration).We consider the same system-wide secure solutions as in [54]: (1) duplicate the documents so that there is no co-occurrence leakage any more, (2) apply the state-of-the-art SSE schemes on the documents directly, and (3) use off-the-shelf data retrieval techniques such as private information retrieval (PIR) and oblivious random-access memory (ORAM). Concretely, we choose Pseudo-Random Transform from [65] (referred to as **PRT-EMM**) and the volume-hiding EMM from [92] (referred to as **VH-EMM**), as well as SealPIR [5] and non-recursive Path ORAM [102] as the PIR and ORAM schemes, respectively, for our comparison.

Parameter Choices. We use the following parameters in our experiments. All PRFs used in the experiments have 128-bit outputs. For **PRT-EMM** [65], we use $\alpha = 0.5$ as proposed in the original paper. For SealPIR [5], we pick the degree of ciphertext to be $N = 2048$, the size of the coefficients to be 60 bits and represent the database in $d = 2$ dimensions as those are used in the original paper. For Path ORAM, we assume each block has size 1 KB and there are 4 blocks per bucket. We pad query response volume from the duplication scheme, SealPIR and Path ORAM to the maximum query response length to suppress volume leakage. For Path ORAM, we assume that all documents are of equal size. Note that leaking the length of retrieved documents is undesirable and the standard practice in the SSE literature is to either pad each document to the same length or to divide each document into equal-sized chunks/sub-documents associated with separate identifiers. Hence, we pick a fixed document-size, which may be viewed as either all documents being padded to the same length, or all documents split into equal-sized chunks.

Concrete Comparison. For the comparison experiments, we use the same 400K (preprocessed) documents from the Enron email corpus. We report storage, computation and communication costs in Table 3. Storage and communication costs are measured in total volumes. Additional overheads arising from how the data is structured and packaged are ignored. It is clear from the table that SWiSSSE is significantly more efficient than all of the alternatives that we consider. This is because SWiSSSE uses delayed write-backs with fresh addresses to suppress access-pattern leakage. That is significantly more efficient than creating physical duplications of documents as used in the duplication scheme and the state-of-the-art index-only SSE schemes [65, 92]. Furthermore, SWiSSSE uses keyword bucketization to suppress volume leakage. The resultant query response volume is much smaller than the worst-case padding strategies adopted by the other solutions.

7 EXTENSION TO DYNAMIC DATABASES

We conclude with a brief discussion on extending SWiSSSE to dynamic databases. We chose to focus on the static version of SWiSSSE in the body of the paper for easy exposition of our core ideas. However, our approach naturally extends to the dynamic setting. In Appendix E, we detail a dynamic version of SWiSSSE and prove its security against system-wide leakage-abuse attacks. This two-phase approach of first focusing on static SWiSSSE before extending it to the dynamic setting is similar to much of the SSE literature, where the static version of a scheme is usually proposed and analyzed extensively before its dynamic counterpart is introduced.[12]

Our dynamic SWiSSSE scheme achieves a new system-wide security notion called "obliviousness of operations", which requires that search and update query operations should be computationally indistinguishable to the server. This brings several advantages. First of all, it naturally implies that search and update operations incur computationally indistinguishable leakage, which allows for a unified system-wide security definition with respect to searches and updates for dynamic SSE schemes, as opposed to the separate index-only definitions in prior work. Secondly, as a consequence of this property, dynamic SWiSSSE achieves *stronger forward and backward privacy guarantees* than state-of-the-art SSE constructions in the literature, *including* those based on volume-hiding EMMs or ORAM [16, 19, 26, 45]. We refer to Appendix E for the full details.

[11] Recall that, as presented in their original forms, they are not [54].

[12] For instance, volume hiding EMMs were first proposed in the static SSE setting [65, 92] before being extended to dynamic databases only recently [45].

ACKNOWLEDGMENTS

The work of Zichen Gui was supported by the Zurich Information Security and Privacy Center (ZISC).

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order-Preserving Encryption for Numeric Data. In *ACM SIGMOD 2004*. ACM, 563–574.

[2] Hime Aguiar e Oliveira Junior, Lester Ingber, Antonio Petraglia, Mariane Rembold Petraglia, and Maria Augusta Soares Machado. 2012. *Adaptive Simulated Annealing*. Springer Berlin Heidelberg, Berlin, Heidelberg. 33–62 pages. https://doi.org/10.1007/978-3-642-27479-4_4

[3] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies* 2016, 2 (April 2016), 155–174.

[4] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2023. Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption. *Proc. Priv. Enhancing Technol.* (2023).

[5] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 962–979. https://doi.org/10.1109/SP.2018.00062

[6] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *ACM SIGMOD 2020*. 1511–1525.

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *CIDR 2013*.

[8] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology – EUROCRYPT 2020, Part II (Lecture Notes in Computer Science, Vol. 12106)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, Germany, Zagreb, Croatia, 403–432. https://doi.org/10.1007/978-3-030-45724-2_14

[9] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. 2023. Oblivious RAM with Worst-Case Logarithmic Overhead. *Journal of Cryptology* 36, 2 (April 2023), 7. https://doi.org/10.1007/s00145-023-09447-5

[10] Léonard Assouline and Brice Minaud. 2023. Weighted Oblivious RAM, with Applications to Searchable Symmetric Encryption. In *Advances in Cryptology – EUROCRYPT 2023, Part I (Lecture Notes in Computer Science, Vol. 14004)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Heidelberg, Germany, Lyon, France, 426–455. https://doi.org/10.1007/978-3-031-30545-0_15

[11] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012: 13th International Conference in Cryptology in India (Lecture Notes in Computer Science, Vol. 7668)*, Steven D. Galbraith and Mridul Nandi (Eds.). Springer, Heidelberg, Germany, Kolkata, India, 489–508. https://doi.org/10.1007/978-3-642-34931-7_28

[12] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Trans. Knowl. Data Eng.* 26, 3 (2014), 752–765.

[13] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*. The Internet Society, San Diego, CA, USA.

[14] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Advances in Cryptology – CRYPTO 2011 (Lecture Notes in Computer Science, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 578–595. https://doi.org/10.1007/978-3-642-22792-9_33

[15] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security 2022: 31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 3917–3934.

[16] Raphael Bost. 2016. $\Sigma o\phi o\varsigma$: Forward Secure Searchable Encryption. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1143–1154. https://doi.org/10.1145/2976749.2978303

[17] Raphael Bost and Pierre-Alain Fouque. 2017. Thwarting Leakage Abuse Attacks against Searchable Encryption – A Formal Approach and Applications to Database Padding. Cryptology ePrint Archive, Report 2017/1060. https://eprint.iacr.org/2017/1060.

[18] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. 2016. Verifiable Dynamic Symmetric Searchable Encryption: Optimality and Forward Security. Cryptology ePrint Archive, Report 2016/062. https://eprint.iacr.org/2016/062.

[19] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 1465–1482. https://doi.org/10.1145/3133956.3133980

[20] Jake Brutlag. 2009. Speed Matters for Google Web Search. https://services.google.com/fh/files/blogs/google_delayexp.pdf

[21] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 668–679. https://doi.org/10.1145/2810103.2813700

[22] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2016. Leakage-Abuse Attacks Against Searchable Encryption. Cryptology ePrint Archive, Report 2016/718. https://eprint.iacr.org/2016/718.

[23] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *ISOC Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society, San Diego, CA, USA.

[24] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 353–373. https://doi.org/10.1007/978-3-642-40041-4_20

[25] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. 2022. Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions. In *USENIX Security 2022: 31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 2425–2442.

[26] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 1038–1055. https://doi.org/10.1145/3243734.3243833

[27] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. 2018. Perfectly Secure Oblivious Parallel RAM. In *TCC 2018: 16th Theory of Cryptography Conference, Part II (Lecture Notes in Computer Science, Vol. 11240)*, Amos Beimel and Stefan Dziembowski (Eds.). Springer, Heidelberg, Germany, Panaji, India, 636–668. https://doi.org/10.1007/978-3-030-03810-6_23

[28] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS 05: 3rd International Conference on Applied Cryptography and Network Security (Lecture Notes in Computer Science, Vol. 3531)*, John Ioannidis, Angelos Keromytis, and Moti Yung (Eds.). Springer, Heidelberg, Germany, New York, NY, USA, 442–455. https://doi.org/10.1007/11496137_30

[29] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *Advances in Cryptology – ASIACRYPT 2010 (Lecture Notes in Computer Science, Vol. 6477)*, Masayuki Abe (Ed.). Springer, Heidelberg, Germany, Singapore, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[30] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Milwaukee, Wisconsin, 41–50. https://doi.org/10.1109/SFCS.1995.492461

[31] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006: 13th Conference on Computer and Communications Security*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM Press, Alexandria, Virginia, USA, 79–88. https://doi.org/10.1145/1180405.1180417

[32] Marc Damie, Florian Hahn, and Andreas Peter. 2021. A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents. In *USENIX Security 2021: 30th USENIX Security Symposium*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 143–160.

[33] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. 2020. DORY: An Encrypted Search System with Distributed Trust. In *OSDI 2020*. 1101–1119.

[34] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *USENIX Security 2020: 29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2433–2450.

[35] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical Private Range Search Revisited. In *ACM SIGMOD 2016*. 185–198.

[36] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos N. Garofalakis, and Charalampos Papamanthou. 2018. Practical

Private Range Search in Depth. *ACM Trans. Database Syst.* 43, 1 (2018), 2:1–2:52.
[37] Ioannis Demertzis, Charalampos Papamanthou, and Rajdeep Talapatra. 2018. Efficient Searchable Encryption Through Compression. *Proc. VLDB Endow.* 11, 11 (2018), 1729–1741.
[38] Casey Devet, Ian Goldberg, and Nadia Heninger. 2012. Optimally Robust Private Information Retrieval. In *USENIX Security 2012: 21st USENIX Security Symposium*, Tadayoshi Kohno (Ed.). USENIX Association, Bellevue, WA, USA, 269–283.
[39] Saba Eskandarian and Matei Zaharia. 2017. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR* abs/1710.00458 (2017). http://arxiv.org/abs/1710.00458
[40] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient Dynamic Searchable Encryption with Forward Privacy. *PoPETs* 2018, 1 (2018), 5–20.
[41] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS 2015: 20th European Symposium on Research in Computer Security, Part II (Lecture Notes in Computer Science, Vol. 9327)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.). Springer, Heidelberg, Germany, Vienna, Austria, 123–145. https://doi.org/10.1007/978-3-319-24177-7_7
[42] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *Advances in Cryptology – CRYPTO 2016, Part III (Lecture Notes in Computer Science, Vol. 9816)*, Matthew Robshaw and Jonathan Katz (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 563–592. https://doi.org/10.1007/978-3-662-53015-3_20
[43] C. Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *ACM STOC'09*. 169–178.
[44] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *Automata, Languages and Programming*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 803–815.
[45] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *Advances in Cryptology – EUROCRYPT 2021, Part III (Lecture Notes in Computer Science, Vol. 12698)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, Heidelberg, Germany, Zagreb, Croatia, 370–396. https://doi.org/10.1007/978-3-030-77883-5_13
[46] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014 (Lecture Notes in Computer Science, Vol. 8441)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer, Heidelberg, Germany, Copenhagen, Denmark, 640–658. https://doi.org/10.1007/978-3-642-55220-5_35
[47] Eu-Jin Goh. 2003. Secure Indexes. Cryptology ePrint Archive, Report 2003/216. https://eprint.iacr.org/2003/216.
[48] Ian Goldberg. 2007. Improving the Robustness of Private Information Retrieval. In *2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA, USA, 131–148. https://doi.org/10.1109/SP.2007.23
[49] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.
[50] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores. In *USENIX Security 2020: 29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2451–2468.
[51] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 315–331. https://doi.org/10.1145/3243734.3243864
[52] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1067–1083. https://doi.org/10.1109/SP.2019.00030
[53] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 361–378. https://doi.org/10.1145/3319535.3363210
[54] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. 2023. Rethinking Searchable Symmetric Encryption. In *IEEE Symposium on Security and Privacy, SP 2023 (to appear)*. Available from https://eprint.iacr.org/2021/879.
[55] Zichen Gui, Kenneth G. Paterson, Sikhar Patranabis, and Bogdan Warinschi. 2020. SWiSSSE: System-Wide Security for Searchable Symmetric Encryption. Cryptology ePrint Archive, Report 2020/1328. https://eprint.iacr.org/2020/1328.
[56] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. 2019. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proc. Priv. Enhancing Technol.* 2019, 1 (2019), 172–191.
[57] Thang Hoang, Attila A. Yavuz, F. Betül Durak, and Jorge Guajardo. 2019. A multi-server oblivious dynamic searchable encryption framework. *J. Comput. Secur.* 27, 6 (2019), 649–676.
[58] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *36th Annual ACM Symposium on Theory of Computing*, László Babai (Ed.). ACM Press, Chicago, IL, USA, 262–271. https://doi.org/10.1145/1007352.1007396
[59] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society, San Diego, CA, USA.
[60] Charanjit S. Jutla and Sikhar Patranabis. 2022. Efficient Searchable Symmetric Encryption for Join Queries. In *Advances in Cryptology – ASIACRYPT 2022, Part III (Lecture Notes in Computer Science, Vol. 13793)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, Heidelberg, Germany, Taipei, Taiwan, 304–333. https://doi.org/10.1007/978-3-031-22969-5_11
[61] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. 2021. Cryptanalysis of Encrypted Search with LEAKER - A framework for LEakage AttacK Evaluation on Real-world data. Cryptology ePrint Archive, Report 2021/1035. https://eprint.iacr.org/2021/1035.
[62] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *Advances in Cryptology – EUROCRYPT 2017, Part III (Lecture Notes in Computer Science, Vol. 10212)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, Germany, Paris, France, 94–124. https://doi.org/10.1007/978-3-319-56617-7_4
[63] Seny Kamara and Tarik Moataz. 2018. Encrypted Multi-Maps with Computationally-Secure Leakage. Cryptology ePrint Archive, Report 2018/978. https://eprint.iacr.org/2018/978.
[64] Seny Kamara and Tarik Moataz. 2018. SQL on Structurally-Encrypted Databases. In *Advances in Cryptology – ASIACRYPT 2018, Part I (Lecture Notes in Computer Science, Vol. 11272)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer, Heidelberg, Germany, Brisbane, Queensland, Australia, 149–180. https://doi.org/10.1007/978-3-030-03326-2_6
[65] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *Advances in Cryptology – EUROCRYPT 2019, Part II (Lecture Notes in Computer Science, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, Germany, Darmstadt, Germany, 183–213. https://doi.org/10.1007/978-3-030-17656-3_7
[66] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *ACM CCS 2012: 19th Conference on Computer and Communications Security*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, Raleigh, NC, USA, 965–976. https://doi.org/10.1145/2382196.2382298
[67] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1329–1340. https://doi.org/10.1145/2976749.2978386
[68] Florian Kerschbaum. 2015. Frequency-Hiding Order-Preserving Encryption. In *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 656–667. https://doi.org/10.1145/2810103.2813629
[69] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. 2015. Optimal Rate Private Information Retrieval from Homomorphic Encryption. *Proceedings on Privacy Enhancing Technologies* 2015, 2 (April 2015), 222–243. https://doi.org/10.1515/popets-2015-0016
[70] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates. In *ACM CCS 2017: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 1449–1463. https://doi.org/10.1145/3133956.3133970
[71] Evgenios M. Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. 2022. Leakage Inversion: Towards Quantifying Privacy in Searchable Encryption. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 1829–1842. https://doi.org/10.1145/3548606.3560593
[72] H. Krawczyk, M. Bellare, and R. Canetti. 1997. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational). , 11 pages. https://doi.org/10.17487/RFC2104 Updated by RFC 6151.
[73] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Miami Beach, Florida, 364–373. https://doi.org/10.1109/SFCS.1997.646125
[74] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San

Francisco, CA, USA, 297–314. https://doi.org/10.1109/SP.2018.00002

[75] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, There is an Oblivious RAM Lower Bound!. In *Advances in Cryptology – CRYPTO 2018, Part II (Lecture Notes in Computer Science, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 523–542. https://doi.org/10.1007/978-3-319-96881-0_18

[76] Kevin Lewi and David J. Wu. 2016. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1167–1178. https://doi.org/10.1145/2976749.2978376

[77] Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing.* MIT Press.

[78] David A. McGrew and John Viega. 2004. The Security and Performance of the Galois/Counter Mode of Operation (Full Version). Cryptology ePrint Archive, Report 2004/193. https://eprint.iacr.org/2004/193.

[79] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP 2013*. 10.

[80] Samir Jordan Menon and David J. Wu. 2022. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 930–947. https://doi.org/10.1109/SP46214.2022.9833700

[81] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 279–296. https://doi.org/10.1109/SP.2018.00045

[82] M. Mughees and L. Ren. 2023. Vectorized Batch Private Information Retrieval. In *IEEE Symposium on Security and Privacy 2023*. 437–452.

[83] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2292–2306. https://doi.org/10.1145/3460120.3485381

[84] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[85] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 644–655. https://doi.org/10.1145/2810103.2813651

[86] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. 2014. Dynamic Searchable Encryption via Blind Storage. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 639–654. https://doi.org/10.1109/SP.2014.47

[87] National Institute of Standards and Technology Gaithersburg MD. 2001. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[88] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *USENIX Security 2021: 30th USENIX Security Symposium*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 127–142.

[89] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *USENIX Security 2021*. USENIX Association, 127–142.

[90] Simon Oya and Florian Kerschbaum. 2021. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. arXiv:2110.04180 [cs.CR]

[91] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *USENIX Security 2022: 31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 2407–2424.

[92] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 79–93. https://doi.org/10.1145/3319535.3354213

[93] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP 2011*. 85–100.

[94] David Pouliot and Charles V. Wright. 2016. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1341–1352. https://doi.org/10.1145/2976749.2978401

[95] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 264–278. https://doi.org/10.1109/SP.2018.00025

[96] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1852–1867. https://doi.org/10.1109/SP40001.2021.00020

[97] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS 2019: 26th Conference on Computer and Communications Security*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 753–768. https://doi.org/10.1145/3319535.3354252

[98] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *Advances in Cryptology – ASIACRYPT 2011 (Lecture Notes in Computer Science, Vol. 7073)*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer, Heidelberg, Germany, Seoul, South Korea, 197–214. https://doi.org/10.1007/978-3-642-25385-0_11

[99] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA, USA, 44–55. https://doi.org/10.1109/SECPRI.2000.848445

[100] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. 2018. Forward Private Searchable Symmetric Encryption with Optimized I/O Efficiency. Cryptology ePrint Archive, Report 2018/497. https://eprint.iacr.org/2018/497.

[101] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage. In *ISOC Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society, San Diego, CA, USA.

[102] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 299–310. https://doi.org/10.1145/2508859.2516660

[103] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 763–780. https://doi.org/10.1145/3243734.3243782

[104] SWiSSSE. 2020. *System-wide Security for Symmetric Searchable Encryption*. https://github.com/SWiSSSE-crypto/SWiSSSE.

[105] Anselme Tueno and Florian Kerschbaum. 2020. Efficient Secure Computation of Order-Preserving Encryption. In *ASIACCS 20: 15th ACM Symposium on Information, Computer and Communications Security*, Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese (Eds.). ACM Press, Taipei, Taiwan, 193–207. https://doi.org/10.1145/3320269.3384739

[106] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security 2018: 27th USENIX Security Symposium*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, Baltimore, MD, USA, 991–1008.

[107] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 339–354. https://doi.org/10.1109/SP40001.2021.00064

[108] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.

[109] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. 2014. SCORAM: Oblivious RAM for Secure Computation. In *ACM CCS 2014: 21st Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 191–202. https://doi.org/10.1145/2660267.2660365

[110] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security 2016: 25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, Austin, TX, USA, 707–720.

A SYSTEM-WIDE DEFINITIONS FOR SEARCHABLE SYMMETRIC ENCRYPTION

In this section, we present our first major contribution – new *system-wide* security definitions for (static) SSE schemes. Our definitions take into account the leakage from *both* the encrypted index and the document retrieval phase during searches. This is in direct contrast to prior *index-only* definitions used extensively in the SSE literature [24, 29, 31] that only take into leakage from the encrypted index component of the overall SSE system. An extension of our definitions to dynamic SSE appears in Appendix E.1.

A.1 System-Wide Syntax for Static SSE

We begin by formally defining our system-wide syntax for a static SSE scheme Σ consisting of two protocols **Setup** and **KWQuery** between a client and a server. A query q from the client to the server takes the form $(op, args)$ where op is the operation of the query, and $args$ are the additional arguments required for the query. We only focus on search queries over a single keyword w.

- **Setup**$(1^\lambda, \mathtt{DB})$ is a client-server protocol. The client takes as input a security parameter 1^λ and a database $\mathtt{DB}$, and outputs $(\mathtt{EDB}, \mathtt{st_{Clt}})$, where $\mathtt{EDB}$ is an encrypted database (including the encrypted search index as well as the encrypted documents) and $\mathtt{st_{Clt}}$ is the client's internal state. The client sends $\mathtt{EDB}$ to the server and the server runs $\mathtt{Svr}.\mathbf{Setup}(\mathtt{EDB})$ to setup the encrypted database.
- **KWQuery**$(sk, q, \mathtt{st_{Clt}}; \mathtt{EDB})$ is a client-server protocol to support single-keyword query. The query q in this case is of the form $(\mathbf{KWQuery}, w)$, where w is the target keyword of the query. The client takes as input a secret key sk, a search query q, and its internal state $\mathtt{st_{Clt}}$. The server takes as input the encrypted database $\mathtt{EDB}$. The client and server interact with each other and by the end of the interaction, the client gets $(r, \mathtt{st_{Clt}})$ where r is the result of the query (i.e., the actual documents matching the query) and $\mathtt{st_{Clt}}$ is the new state of the client; the server state $\mathtt{EDB}$ may also be modified following the execution.

Note that our syntax for static SSE takes into account the actual encrypted documents (as part of the encrypted database $\mathtt{EDB}$ output by **Setup** and as part of the query response r output by **KWQuery**) in addition to the encrypted index.

A.2 System-Wide Security for Static SSE

We now present our new system-wide security definitions for static SSE. As in prior index-only definitions for static SSE [24, 29, 31], we formulate system-wide security in terms of a *leakage* function which captures the information an adversary can unavoidably learn; however, in contrast to existing definitions, our definitions also capture the information leakage from the document retrieval phase. We say that a static SSE scheme is system-wide secure with respect to a leakage function $\mathcal{L} = (\mathcal{L}^{\mathbf{Setup}}, \mathcal{L}^{\mathbf{KWQuery}})$ if no (honest-but-curious) adversary can distinguish between the real execution and an execution simulated given only the leakage, where the real and simulated executions are outlined below.

Definition A.1 (System-Wide Security of Static SSE). Let $\Sigma = (\mathbf{Setup}, \mathbf{KWQuery})$ be a static SSE scheme and consider the following probabilistic experiment where $\mathcal{A}$ is a stateful, probabilistic polynomial time (PPT) honest-but-curious adversary, $\mathcal{S}$ is a stateful simulator and $\mathcal{L}$ is the leakage function.

$\underline{\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda):}$

(1) The adversary $\mathcal{A}$ selects a database $\mathtt{DB}$ and gives it to the challenger C.
(2) The challenger C generates a key sk, and encrypts the database as $\mathtt{EDB} \leftarrow \mathbf{Setup}(1^\lambda, sk, \mathtt{DB})$. The challenger C sends the encrypted database $\mathtt{EDB}$ to the adversary $\mathcal{A}$.
(3) The adversary picks a polynomial number of keyword search queries $q_1, \ldots, q_{\mathtt{poly}(\lambda)}$. For each query, the challenger C interacts with the adversary $\mathcal{A}$ to execute the search query protocol (including the final document retrieval phase), where the challenger plays the client and the adversary plays the server.
(4) Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

$\underline{\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda, \mathcal{L} = (\mathcal{L}^{\mathbf{Setup}}, \mathcal{L}^{\mathbf{KWQuery}})):}$

(1) The adversary $\mathcal{A}$ selects a database $\mathtt{DB}$ and gives $\mathcal{L}^{\mathbf{Setup}}(\mathtt{DB})$ to the simulator $\mathcal{S}$.
(2) Using $\mathcal{L}_{\mathbf{Setup}}(\mathtt{DB})$, the simulator $\mathcal{S}$ generates $\mathtt{EDB}$ and returns it to the adversary $\mathcal{A}$.
(3) The adversary picks a polynomial number of search queries $q_1, \ldots, q_{\mathtt{poly}(\lambda)}$. For each query, the simulator $\mathcal{S}$ computes the transcript of the query using $\mathcal{L}^{\mathbf{KWQuery}}$ (including the leakage from the document retrieval phase corresponding to the query), and sends it to the adversary $\mathcal{A}$.
(4) Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

We say that Σ is $\mathcal{L}$-secure if there exists a probabilistic polynomial-time (PPT) simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$,

$$\left|\Pr\left[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda, \mathcal{L}) = 1\right]\right| \leq \mathtt{negl}(\lambda).$$

Differences with Index-Only Security Definitions. Unlike prior (index-only) SSE definitions [24, 29], our security definition for static SSE takes into account the actual encrypted documents and the leakage from the document retrieval phase. This follows from the facts that: (a) the encrypted database $\mathtt{EDB}$ in both the real and ideal worlds includes the actual encrypted documents, and (b) the adversary $\mathcal{A}$ gains access to the leakage from the document retrieval phase in both the real and ideal worlds.

Comparison with [31]. The security definition for SSE in [31] does account for leakage during document retrieval. However, their definition assumes a pre-specified "noise-free" access-pattern leakage by default and fails to capture SSE schemes with noisy/suppressed leakage, including index-only SSE schemes (e.g., [34, 45, 65, 92]) and end-to-end SSE systems (e.g., SWiSSSE). Additionally, the concrete schemes proposed in [31] are index-only with no leakage suppression, no specification of document retrieval (in Definition 4.1 of [31], the Search algorithm only outputs "a set X of (lexicographically-ordered) document identifiers"), and no enumeration of system-wide leakage. One could, of course, naturally extend these index-only schemes into end-to-end SSE systems matching the security definition of [31]. However, such systems would leak the exact access pattern anyway, and would be broken trivially by system-wide attacks [54].

Our system-wide definition of SSE is significantly more general. It is parameterized by a generic leakage function (as opposed to pre-specified leakage in [31]), and allows modeling a larger class of (end-to-end) SSE schemes with leakage suppression and/or stateful/probabilistic leakage profiles. Moreover, in contrast to the index-only SSE schemes in [31], SWiSSSE precisely specifies how to retrieve documents, and the leakage from document-retrieval is enumerated explicitly in its system-wide leakage profile.

Adaptive vs Non-adaptive Adversaries. In the security definition above, we can distinguish between non-adaptive and adaptive adversaries, depending on how the adversary chooses its queries. We say that Σ is non-adaptively secure if the adversary $\mathcal{A}$ chooses all the queries before executing them. We say that Σ is adaptively secure if the adversary $\mathcal{A}$ can choose each of his queries based on the transcripts resulting from his previous queries.

A.3 System-Wide Correctness for Static SSE

To define correctness of static SSE in a system-wide setting, we build upon some of the formalism in our system-wide security definitions above. Concretely, consider a game between a challenger and a PPT adversary $\mathcal{A}$ very similar to the the game $\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda)$ described above with the adversary $\mathcal{A}$ issuing at most $k = \mathtt{poly}(\lambda)$ single-keyword queries during the game, except that at the end of the game, a PPT distinguisher $\mathcal{D}$ receives one of the following:

- the data obtained by executing the plaintext queries against the unencrypted/plaintext database (including the document retrieval phase), or
- the data obtained from the execution of the **KWQuery** protocol between the challenger C and the adversary $\mathcal{A}$ (once again, including the document retrieval phase).

Let $Adv^{corr,k}_{\Sigma,\mathcal{A},\mathcal{D}}(1^\lambda)$ denote the probability that the distinguisher $\mathcal{D}$ distinguishes between the aforementioned data distributions. We say that the SSE scheme Σ is correct from a system-wide perspective if for any such PPT adversary $\mathcal{A}$ and any such PPT distinguisher $\mathcal{D}$, we have $Adv^{corr,k}_{\Sigma,\mathcal{A},\mathcal{D}}(1^\lambda) \le \mathtt{negl}(\lambda)$.

Our correctness definition takes into account the actual encrypted documents returned by the query, as opposed to only the encrypted document identifiers resulting from querying the index considered by the prior (index-only) SSE definitions [24, 29, 31].

B FORMAL PROOF OF CORRECTNESS AND LEAKAGE PROFILE OF SWISSSE

In this section, we formally prove Theorem 5.1 and Theorem 5.2.

B.1 Proof of Theorem 5.1

We use standard game-hopping to reduce the correctness game G to finding a pair of collisions in a game where the adversary interacts with truly random functions. Let game G_2 be the game where the PRF used to generate the addresses for the encrypted documents is replaced by a truly random function. The number of addresses used for the encrypted documents with k queries is at most $|\mathsf{DB}| + 2k \cdot \max_w |\mathsf{DB}(w)|$, so the difference in the advantages between game G and G_2 is $Adv_F^{PRF,|\mathsf{DB}|+2k\cdot\max_w |\mathsf{DB}(w)|}$. For database DB, we use $|\mathsf{DB}|$ addresses to store the encrypted documents during initialisation; hence, the probability of a pair of collisions is upper-bounded by $|\mathsf{DB}|^2 \cdot 2^{-(l+1)}$.

In the subsequent write-backs, the number of active addresses in the encrypted document array is at most $|\mathsf{DB}|$ and the number of new addresses we generate is upper bounded by $2\max_w |\mathsf{DB}(w)|$. This means that there are at most $2|\mathsf{DB}| \cdot \max_w |\mathsf{DB}(w)|$ new potential pairs of collisions for each query. Using the birthday bound, the probability of finding a collision in the addresses of the encrypted document array in each write-back is upper bounded by $2|\mathsf{DB}| \cdot \max_w |\mathsf{DB}(w)| \cdot 2^{-l}$.

Similarly, we define game G_3 be the game where the PRF used to generate the addresses for the encrypted lookup table is replaced with a truly random function. The number of addresses used for the encrypted lookup table is at most $|W\{\mathsf{DB}\}| + 2k\max_w |\mathsf{DB}\{w\}|$, so the difference in the advantages between game G_2 and G_3 is $Adv_F^{PRF,W\{\mathsf{DB}\}+2k\max_w |\mathsf{DB}\{w\}|}$.

Using a similar argument as above, the probability of a pair of collision in the addresses for the encrypted lookup table during initialisation is at most $|W\{\mathsf{DB}\}|^2 \cdot 2^{-(l+1)}$, and the probability of a pair of collision for the encrypted lookup table for each query is at most $2|W\{\mathsf{DB}\}| \cdot \max_w |\mathsf{DB}\{w\}| \cdot 2^{-l}$.

Combining everything together with a union bound on all k queries, we conclude that the failure probability of the construction is at most

$$\left(|\mathsf{DB}|^2 + 4t_0|\mathsf{DB}| + |W\{\mathsf{DB}\}|^2 + 4t_1|W\{\mathsf{DB}\}|\right) \cdot 2^{-(l+1)} + Adv_F^{PRF,|\mathsf{DB}|+2t_0} + Adv_F^{PRF,W\{\mathsf{DB}\}+2t_1},$$

where $t_0 = k \cdot \max_w |\mathsf{DB}(w)|$ and $t_1 = k \cdot \max_w |w\{\mathsf{DB}(w)\}|$. This completes the proof of Theorem 5.1.

B.2 Formal Description of Leakage Profile for Static SWiSSSE

In this section, we formally describe the leakage functions $\mathcal{L}_\Sigma^{\mathbf{Setup}}$ and $\mathcal{L}_\Sigma^{\mathbf{KWQuery}}$ corresponding to leakage of static SWiSSSE during setup and keyword queries, respectively. We describe these probabilistic and stateful leakage functions in Algorithm 5 for an instance Σ of the SWiSSSE scheme. The state of the leakage function contains a realisation of the padded database, everything in the stash of the client, and two data structures, namely the lookup history **IndHist** and the array-access history **ArrHist**, which we describe below.

Lookup and Array-Access Histories. We define the lookup history **IndHist** and the array-access history **ArrHist** to be lists of records corresponding to operations on the encrypted keyword lookup index and the encrypted document array, respectively. These capture the access pattern leakage during searches.

Formally, each lookup (respectively, array-access) record is of the form (i, b, t), where i is a unique identifier associated with the index (respectively, array) address being operated on, b is a bit which indicates whether the entry corresponds to a read operation ($b = 0$) or a write operation ($b = 1$), and t is the timestamp of the query that resulted in the operation. We also overload notation and let $\mathbf{IndHist}\left[\{i_j\}\right]$ and $\mathbf{ArrHist}\left[\{i_j\}\right]$ be the sets of all records

Algorithm 5 SWiSSSE: Leakage Function for Searches and Write-Backs

procedure $\mathcal{L}_{\Sigma}^{\mathbf{KWQuery}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\mathbf{KWQuery}, w) \leftarrow query$
  $\mathtt{I}, \mathtt{A}, \mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr}, \mathbf{IndHist}, \mathbf{ArrHist} \leftarrow \mathbf{St}_{\mathcal{L}}$
  /* *Leakage from the query tokens* */
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(w, i, \mathtt{KWCtr}[w]), 0, k), \mathbf{T}(w, i, \mathtt{KWCtr}[w] + 1), 0, k) \mid i \in 1, \ldots, \mathtt{G}(w)\}$
  $\mathtt{KWCtr}[w] \leftarrow \mathtt{KWCtr}[w] + 2$
  /* *Leakage from document array access* */
  $L \leftarrow \mathtt{I}[w]$
  while $|L| < 2 \cdot \mathtt{Clt.G}(w)$ **do**
    $id \leftarrow \mathbf{Rand}(|\mathtt{A}|)$
    if $id \notin \{id(d) \mid d \in \mathtt{A}'\}$ **then**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

  /* *Leakage from write-back* */
  $UI \leftarrow \mathtt{I}'.\mathbf{pop}(|\mathtt{I}'|/2)$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(i, 1, k) \mid i \in UI\}$
  $UA \leftarrow \mathtt{A}'.\mathbf{Pop}(\lfloor |UA|/2 \rfloor)$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(id(d), \mathtt{ArrCtr}[id(d)]), 1, k) \mid (\{w_i, j_i, b_i\}, d) \in UA\}$

  /* *Update the stash for consistency* */
  $\mathtt{I}' \leftarrow \mathtt{I}' \cup \mathbf{Index}(UA, \mathtt{KWCtr})$
  $\mathtt{A}' \leftarrow \mathtt{A}' \cup \mathbf{Retrieve}(\mathtt{A}[L], w)$
  $\mathbf{St}_{\mathcal{L}} \leftarrow (\mathtt{I}, \mathtt{A}, \mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr}, \mathbf{IndHist}, \mathbf{ArrHist})$
  Return $(\mathbf{IndHist}, \mathbf{ArrHist}), \mathbf{St}_{\mathcal{L}}$

procedure $\textsc{Index}(UA, \mathtt{KWCtr})$
  $\mathtt{I} \leftarrow \{\}$
  for $(\{w_i, j_i, b_i\}, d) \in UA$ **do**
    $\mathtt{I} \leftarrow \mathtt{I} \cup \{\mathbf{T}(w, j, \mathtt{KWCtr}[w] + b) \mid (w, j, b) \in \{w_i, j_i, b_i\}\}$
  Return $\mathtt{I}$

procedure $\textsc{Retrieve}(\mathtt{A}, w)$
  $\mathtt{A}' \leftarrow \{\}$
  for $(\{w_i, j_i\}, d) \in \mathtt{A}$ **do**
    $\mathtt{A}' \leftarrow \mathtt{A}' \cup \{w_i, j_i, (w_i = w)\}, d)$
  Return $\mathtt{A}'$

corresponding to a given set $\{i_j\}$ of index/array addresses, i.e.,

$$\mathbf{IndHist}\left[\{i_j\}\right] = \left\{(i, b, t) \mid (i, b, t) \in \mathbf{IndHist} \wedge i \in \{i_j\}\right\}.$$

$$\mathbf{ArrHist}\left[\{i_j\}\right] = \left\{(i, b, t) \mid (i, b, t) \in \mathbf{ArrHist} \wedge i \in \{i_j\}\right\}.$$

B.3 Proof of Theorem 5.2

The proof proceeds with a hybrid argument. Let DB be a database and $q_1, \ldots, q_k$ be the set of single-keyword queries with the leading keywords $w_1, \ldots, w_k$. Let $Adv_F^{PRF,t}(\lambda)$ be the PRF advantage of the PRF F with at most t evaluations used in the construction and $Adv_{\Sigma'}^{IND-CPA}(\lambda)$ be the IND-CPA advantage of the scheme Σ' used for lookup address encryption and document content encryption. Finally, we assume the plaintext of the lookup addresses is padded to ℓ_0 and the plaintext of the document contents is padded to ℓ_1. The simulator has access to ℓ_0 and ℓ_1 as they are public parameters.

(**Game 0.**) Let the real execution of the scheme on the database DB with queries $q_1, \ldots, q_k$ be game G_0. Then we have that for any adversary $\mathcal{A}$,

$$\Pr\Big[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^{\lambda}) = 1\Big] = \Pr[G_0 = 1].$$

Algorithm 6 Game G_1. Only the setup step is changed.

procedure $\mathtt{Clt}.\textsc{Setup}(\mathtt{DB})$
  $(N, p, \mathbf{St}_{\mathcal{L}}) \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{Setup}}(\mathtt{DB}, \mathtt{G})$
  $\mathtt{EI}, \mathtt{EA} \leftarrow [\ ]$
  /* *Generate the encrypted documents* */
  for $i = 0, \ldots, N-1$ **do**
    $\mathtt{EA}.\mathbf{Insert}(\mathbf{RF}(i), \mathbf{Enc}(0^{l_0}))$
  /* *Generate the encrypted document addresses* */
  for $i = 0, \ldots, 2p$ **do**
    $\mathtt{EI}.\mathbf{Insert}(\mathbf{RF}(2i+1), \mathbf{Enc}(0^{l_1}))$
  Send $(\mathtt{EI}, \mathtt{EA})$ to the server

(**Game 1.**) We define game G_1 by letting the leakage function generate the padded database and replace the addresses in the encrypted lookup index and the encrypted documents with outputs generated from a truly random function **RF** with output length l. We omit the conversion between an integer to a string of appropriate length in the use of the random function for simplicity. In addition, the encryptions of the document addresses and the documents themselves are replaced with encryptions of zeros of length ℓ_0 and ℓ_1 respectively.

The number of addresses that needs to be generated in the initialisation step is equal to $t_0 = \sum_w \mathtt{G}(w) + |\mathtt{DB}|$. An equal number of encryptions need to be created, so the difference in advantages between G_0 and G_1 is upper-bounded by $Adv_F^{PRF,t_0} + t_0 \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$.

(**Game 2.**) In game G_2, we replace the single-keyword query algorithm with a simulator that has access to the output of the leakage function only. As before, the addresses are generated by applying the truly random function **RF** on the indices provided by the leakage function. The encrypted documents and the encrypted document addresses are generated with encryptions of zeros of appropriate length. The way the addresses are generated is consistent with game G_1 as the only difference between the two games is that the leakage function is responsible for randomising the write-backs.

The number of addresses the algorithm has to generate is upper-bounded by $t_1 = 2\sum_i \mathtt{G}(W(q_i)) + 2\sum_i |\mathtt{DB}(W(q_i))|$. The number of encryptions that need to be created is upper-bounded by the same t_1. This means the difference in advantages between G_1 and G_2 is upper-bounded by $Adv_F^{PRF,t_1} + t_1 \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$.

(**Conclusion.**) By combining the two games above, we see that the difference in advantages between G_0 and G_2 is at most Adv_F^{PRF,t_0+t_1} $+(t_0 + t_1) \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$.

C CRYPTANALYSIS OF THE LEAKAGE OF SWISSSE

Having established the precise leakage of SWiSSSE, we now turn to its cryptanalysis. We begin by giving a brief discussion on the leakage profile of SWiSSSE.

Algorithm 7 Game G_2.

procedure Clt.**KWQuery**(q)
  $(\mathbf{IndHist}, \mathbf{ArrHist}), \mathbf{St}_{\mathcal{L}} \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{KWQuery}}(\mathtt{DB}, q, \mathbf{St}_{\mathcal{L}})$

  /* *Encrypted document array address retrieval* */
  $L \leftarrow \{\}$
  $t' \leftarrow$ the number of single-keyword queries executed
  for $i \in \{i \mid (i, b, t) \in \mathbf{IndHist}, b = 0, t = t'\}$ **do**
    $L \leftarrow L \cup \mathbf{RF}(2i + 1)$
  Send L to the server

  /* *Encrypted document retrieval* */
  $L \leftarrow \{\}$
  for $i \in \{i \mid (i, b, t) \in \mathbf{ArrHist}, b = 0, t = t'\}$ **do**
    $L \leftarrow L \cup \mathbf{RF}(2i)$
  Send L to the server

  /* *Write-back* */
  $UI, UA \leftarrow \{\}$
  for $i \in \{i \mid (i, b, t) \in \mathbf{IndHist}, b = 1, t = t'\}$ **do**
    $UI \leftarrow UI \cup (\mathbf{RF}(2i + 1), \mathbf{Enc}(0^{l_0}))$
  for $i \in \{i \mid (i, b, t) \in \mathbf{ArrHist}, b = 1, t = t'\}$ **do**
    $UA \leftarrow UA \cup (\mathbf{RF}(2i), \mathbf{Enc}(0^{l_1}))$
  Send (UI, UA) to the server

We then briefly argue that SWiSSSE is resilient to most well-known attack techniques, including the query recovery attacks proposed in [13, 21, 59, 89], the document recovery attacks proposed in [21] and the file-injection attacks proposed in [110].

Finally, we analyze the security of SWiSSSE against a highly specialized query reconstruction attack based on system-wide leakage. The attack analysis is set up in a model that is as "unfriendly" for SWiSSSE as possible: (1) the attacker gets more information than is actually leaked from SWiSSSE; (2) the attacker observes *all* possible keyword queries; and (3) the auxiliary data used by the attacker is the *same* as the target database. The first two attack assumptions are, in fact, stronger than any of the previous attacks [13, 21, 59, 88], and are unlikely to be realizable in practice. We demonstrate that, despite the strong attack setting, SWiSSSE resists this specialized query reconstruction attack so long as the bucketization parameters are chosen appropriately. The authors of [54] actually used a refined version of this attack to break several state-of-the-art SSE schemes, albeit while relying on a significantly weaker attack model as compared to the one we use here (and as compared to existing attacks).

Remark C.1. Our cryptanalysis of SWiSSSE can be viewed as a highly refined form of the attacks proposed originally in [54] (which worked with more noisy versions of system-wide leakage). Our attacks assume a "worst-case" version of the (probabilistic) leakage of SWiSSSE, as well as very strong assumptions on the auxiliary information available to the adversary. In the real world, we expect the leakage from SWiSSSE to be significantly smaller than what we assume in our cryptanalysis, and hence the resistance of SWiSSSE to such attacks would be greater than reflected in our experiments. We opt for such an "unfriendly/pessimistic" leakage analysis to account for the worst-case leakage of SWiSSSE, but expect such leakage to occur with very small probability in practice.

C.1 Discussion of the Leakage Profile

Search Pattern Leakage. SWiSSSE leaks search pattern/query equality only probabilistically. Consider a query with response volume v at time t, the write-back after the query will contain about $v/2$ of the retrieved documents. If the same keyword is queried in the future, the server will be able to tell that about $v/4$ (the additional factor of $1/2$ comes from dummy retrievals) of the documents came from time t and guess that the two queries have the same underlying keywords. However, this attack only works probabilistically as: (1) the write-backs are pseudo-random and it is possible that none of the documents retrieved at time t are written back immediately, and (2) the documents that have been written back may be touched by dummy document retrievals, thereby breaking the traceability of these documents (and hence the traceability of the query itself).

Volume Leakage. The bucketization strategy allows us to choose a trade-off between volume leakage and search efficiency. An extreme instantiation of the bucketization strategy is to place every keyword in the same bucket. This is equivalent to worst-case padding [63] and inherently imposes a linear search overhead. Instead, we use a few well-spaced out buckets in the frequency spectrum to hide a significant fraction of the volume leakage, while retaining efficient searches in practice.

Access Pattern Leakage. As established formally in Theorem 5.2, the leakage function output for a search operation on keyword w does not reveal to the adversarial server the identifiers of documents containing w. In addition, all delayed pseudorandom write-backs allow the server to see freshly re-encrypted documents and pointers to documents. Hence, our scheme is free of access pattern leakage.

Co-occurrence Leakage. SWiSSSE effectively hides all forms of leakage that could allow the adversarial server to correlate queries executed by the client across time. Intuitively, the adversary learns the co-occurrence information of the keywords when queries for different keywords are made and there are documents returned in common between these queries. In SWiSSSE, any retrieved document is re-encrypted and written back to the server using a fresh address with a random delay from the time when the document was first retrieved, so only a small fraction of the the co-occurrence pattern can be inferred from the document access pattern at the best. Further, we update the encrypted document addresses for the auxiliary keywords of the documents written back, but this is done with fresh addresses so that the adversary cannot link these addresses to the previous queries. In other words, the adversary will find it hard to group these auxiliary keywords with the correct leading keywords to produce the co-occurrence information of the encrypted database.

C.2 Resistance to Traditional Leakage-Abuse Attacks

Traditional query recovery attacks (such as those proposed in [13, 21, 59, 88]) and document recovery attacks (such as those proposed in [21]) typically rely on a combination of three kinds of deterministic leakage – volume/frequency leakage, document access pattern leakage, and query equality/search pattern leakage. Through the use of keyword bucketization, SWiSSSE makes it possible to suppress volume leakage sufficiently to prevent these attacks (we subsequently discuss in detail the appropriate bucket-sizes needed). Similarly, the use of delayed pseudorandom write-backs corresponding to each query prevents the adversary from deterministically learning the document access patterns and the query equality patterns across multiple queries. In summary, existing cryptanalytic techniques for query and document recovery cannot be applied directly to cryptanalyze the leakage profile for SWiSSSE.

C.3 Resistance to System-Wide Leakage-Abuse Attacks

Since SWiSSSE resists the known cryptanalytic attacks, we tested its resistance against an even stronger query recovery attack in which we give the attacker access to a highly refined co-occurrence leakage. This setting strengthens that of [13, 21, 59], as well as the system-wide leakage-based attack setting considered in [54], where the attacker has access to leakage from both the index retrieval and document retrieval phases. We use this strong (and somewhat idealized) attack model to establish settings for various design parameters for SWiSSSE, and to compare the pros and cons of the various bucketization strategies discussed earlier.

Attack Assumption. Let $M : \{1, \ldots, |\mathsf{DB}|\}^2 \rightarrow \mathbb{N}$ to be a two-dimensional "co-occurrence" matrix that maps pairs of keywords to the number of documents containing them both. Formally, we have $M_{i,j} = \left|\mathsf{DB}(w_i) \cap \mathsf{DB}(w_j)\right|$. It is important to note that this matrix is defined with respect to the original database (before padding/bucketization). We assume here that at the beginning of the attack, the attacker has an exact copy of matrix M. This is a very strong assumption, because it is not obvious how the attacker might obtain this information.

We also simulate an "observed" co-occurrence matrix $\bar{M}$ as follows: let M' be the "co-occurrence" matrix for the padded version of the database. We simulate $\bar{M}_{i,j}$ as a value sampled according to a binomial distribution as follows:

$$\bar{M}_{i,j} \leftarrow \mathsf{Binom}(M'_{i,j}, 1/q),$$

where $1/q$ is a parameter of SWiSSSE denoting the fraction of the local stash flushed out by the client in each write-back operation (we use $q = 2$ for our cryptanalysis experiments in keeping with the formal description of SWiSSSE in Section 5). We assume that the attacker has access to a randomly permuted version of $\bar{M}$, which we abuse the notation $\bar{M}$ to denote it.

Again, this is a very strong assumption. Inferring the matrix $\bar{M}$ from the leakage profile of SWiSSSE is highly non-trivial, since the intermittent and pseudorandom nature of the write-back operations corresponding to each search query makes it very hard for the attacker to identify if the same document appears across multiple search queries.

Attack Strategy. The goal of the attacker is to identify the most likely assignment between the (unknown) queried keywords shown in $\bar{M}$ and the actual keywords. Note that the randomized nature of write-backs in SWiSSSE rules out the deterministic approaches in previous works, such as count-based approaches [21] and matrix/graph matching-based approaches [59, 94]. We opt for a simulated annealing-based approach [2] to search for the most likely keyword assignment following the attack strategy in [54], albeit using the somewhat idealized leakage model described above. We refer to [54] for the details of the attack procedure.

Cryptanalysis Results on Enron Email Corpus. We run the simulated-annealing based attack against the Enron email corpus.[13] We focus on the bucketization strategy in the experiments. We preprocess the documents in the same way we described in Section 6.1 and generate co-occurrence matrices from 400K emails, with keywords from different frequency ranges. Specifically, we arranged the keywords in decreasing order of frequency and chose 800 of the most frequent keywords and 800 of the keywords from the 95-th to 75-th percentile frequencies respectively. We constructed the co-occurrence matrices M and $\bar{M}$ with varying bucket sizes (in the range of 50 to 400) for each of these keyword sets. We repeated the attack for 100 times with freshly generated $\bar{M}$ matrices, and the average recovery rates are reported in Figure 2a.

We observe that the keyword recovery rates do not follow a linear trend. For the most frequent keywords, we see very high query recovery rates, but as soon as we get to 85-th to 90-th percentile, the query recovery rate drops to almost zero. But as the keyword frequency decreases even further into the 80-th percentile, we begin to see significant query recovery rate for small bucket sizes again.

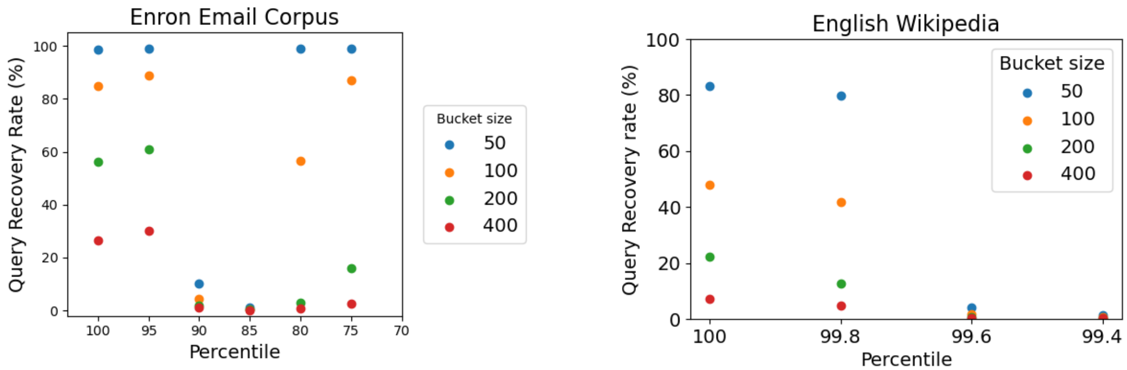
This phenomenon is likely due to the interaction between real and fake co-occurrence counts in different frequency ranges. For the Enron dataset with 400000 documents, the two most frequent keywords occurred 28919 and 28587 times respectively. So we expect about 2114 fake co-occurrence counts between these two keywords due to random padding. on the other hand, most of the real co-occurrence counts between these two keywords and other keywords are in the range of 4000 to 9000, and the amount of randomness in the fake co-occurrence counts is insufficient to hide the real ones.

For keywords with frequencies in the 85-th to 90-th percentile, the real frequencies are on the order of 10^3, and one can expect fake co-occurrence counts on the order of tens. The later is exactly the range of values one expect to find the co-occurrence counts in, which means that the noise is just right to mask the real co-occurrence counts.

Finally, for keywords with frequencies in the 80-th percentile and lower, the real frequencies are on the order of 10^2, and one does not expect any fake co-occurrence counts between these keywords. This means that the attack is essentially trying to match the real co-occurrence count to itself (with padded keyword frequencies of course), so it is not at all surprising to see high query recovery rates for small bucket sizes. We argue that this is not a weakness of our construction as a real-world adversary will likely receive a noisy auxiliary co-occurrence matrix as opposed to the perfect one. In that case, an attack on these keywords will be much harder as the co-occurrence counts with these keywords are very small and contain very little information. In fact, the attacks have shown that using larger bucket sizes on these keywords is already enough to create enough ambiguity.

Additional Cryptanalysis on English Wikipedia. We repeat our cryptanalysis on the English Wikipedia dump[14] from 2012. The articles in the English Wikipedia dump are much longer, so each

[13] https://www.cs.cmu.edu/~enron/
[14] http://kopiwiki.dsd.sztaki.hu/



(a) The experimental results on the Enron email corpus.

(b) Experimental results on the English Wikipedia dump.

Figure 2: Experimental results on the Enron email corpus and the English Wikipedia dump. The query recovery rate reported is averaged over 100 independent runs of the attack. Fixed bucket size is used for the experiment.

"document" in the database contains a lot more keywords. This should, in theory, generate richer co-occurrence information. We sort the keywords in decreasing order of frequency just as before, and choose 800 of the most frequent keywords and 800 keywords from the 99.8-th, 99.6-th and 99.4-th percentile frequencies respectively. Our choices of keywords are significantly different from those for the Enron email corpus, but that is due to the fact that the keyword universe of the English Wikipedia dump is two orders of magnitude larger than that of the Enron email corpus, and over 60% of the keywords only appear once in the whole dataset. We test our attack with varying bucket sizes and repeat it 100 times for each set of attack just as before. The average query recovery rates reported in Figure 2b establish the resistance of SWiSSSE against system-wide leakage-abuse attacks.

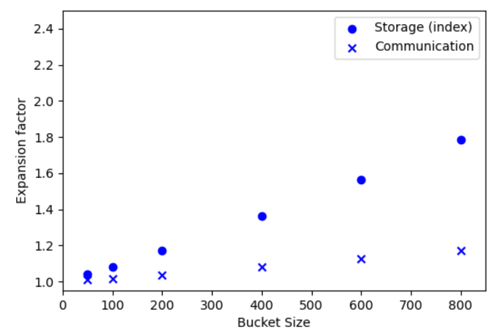


Figure 3: Overheads incurred by different bucket sizes on the Enron email corpus. The experiments are conducted with fixed bucket sizes.

C.4 Discussion

Interpreting the Experimental Results. Our cryptanalysis experiments assumed a very optimistic attack setting where the attacker has access to refined co-occurrence leakage. In practice, the leakage profile of SWiSSSE is significantly more "noisy"; it is not at all obvious how the attacker might obtain access to such refined leakage from a real implementation.

On the other hand, we acknowledge the need for further cryptanalysis of the leakage profile of SWiSSSE and welcome such studies from the community.

Security Versus Efficiency Tradeoffs for Bucketization. Our experiments reveal some interesting insights into the security versus efficiency trade-offs associated with choosing the bucket size. For example, we saw earlier that a bucket size of 50 for the most frequent keywords leads to almost 100% recovery, while a bucket size of 400 reduces this to around 30%. But what is the implication of using a larger bucket-size on the storage and bandwidth requirements for SWiSSSE?

In Figure 3, we demonstrate through concrete figures how variations in bucket sizes affect the storage and communication overheads of our construction. Here, the storage overhead only applies to the index as the number of documents is unaffected by bucketization in SWiSSSE. In general, the total number of (real and fake) keyword-document pairs grows essentially linearly with the bucket size. This implies that the search index overhead also grows linearly with bucket size.

Interestingly, the growth in overhead is more gradual when compared to the fall in recovery rate. When the initial bucket size varies from 50 to 400, the storage overhead varies between 1.04× and 1.36×. On the other hand, as demonstrated earlier, the keyword recovery rate falls from 100% to below 30%. This indicates that it is preferable to opt for a larger bucket size as long as the user can afford it, since it provides significantly stronger resistance to cryptanalysis while incurring only moderately larger overheads.

Parameter Selection. As discussed in Section 6.2, there are three tunable parameters in SWiSSSE that affect its security. These parameters are: (1) the size of the buckets, (2) the fraction of documents written back in the write-back step, and (3) the fraction of lookup indices written back in the write-back step. We provide extensive cryptanalysis and performance experiments on bucketization in this section. Given these experiments, we give recommended bucket sizes in Section 6.2. We also perform additional experiments on the write-back rate and give our recommended write-back rate in the same section.

D SWISSSE: ASYMPTOTIC PERFORMANCE EVALUATION

In this section, we provide an asymptotic performance analysis of SWiSSSE (summarized in Figure 4).

Size of the Stash. For search queries, recall that the half of the stash is flushed every iteration and filled with the response from the latest query. Since the number of documents retrieved by any query is less than $2 \cdot \max_w \mathsf{G}(w)$ (half of that comes from randomly generated document addresses), there are at most $4 \cdot \max_w \mathsf{G}(w)$ documents in the stash. The documents are padded to a constant size, which means the storage of the documents in the stash requires $O(\max_w \mathsf{G}(w))$ space.

The stash also stores a local lookup index. For a search query on keyword w, the number of lookup index locations that need to be updated is equal to the number of keyword-document pairs in the query response, or $|W\{\mathsf{DB}(w)\}|$. Since the number of keywords in the document is much smaller than $|W\{\mathsf{DB}(w_1)\}|$, it is reasonable to treat k as a constant in the asymptotic analysis. Recalling that half of the lookup index stored in the stash is flushed to the server after each query, it is not hard to see that the maximum number of lookup index locations stored by the client is $O(\max_w |W\{\mathsf{DB}(w)\}|)$.

In addition, the client needs to store two arrays of integers, namely an array for the groupings of the keywords and an array for the counters used to generate the document array addresses. These arrays are all small and of constant size, so they do not contribute to the asymptotic size of the stash. Combining everything, we get that the size of the stash is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$.

Size of the Encrypted Database. The server stores an encrypted lookup index and an encrypted document array. The size of the encrypted lookup index is proportional to the total number of keyword-document pairs whereas the size of the encrypted document array is proportional to the number of documents. Hence, the size of the encrypted database is $O(\sum_w \mathsf{G}(w) + |\mathsf{DB}|)$. Note that this order-of-magnitude calculation ignores the overhead from padding all documents to a constant size.

Time Complexity and Communication Volume of a Query. Suppose that the leading keyword for the query is w. The client first computes the encrypted lookup index addresses for the query. This involves $O(\mathsf{G}(w))$ computation and communication, as there are at most $2 \cdot \mathsf{G}(w)$ addresses involved. The server then takes $O(\mathsf{G}(w))$ time to retrieve the encrypted document array addresses and send them to the client. Upon receiving the $O(\mathsf{G}(w))$ encrypted document array addresses, the client processes them and retrieves $2 \cdot \mathsf{G}(w)$ encrypted documents from the server. The client decrypts the documents and filters the results locally to obtain the query response. The time complexity for the overall process is $O(\mathsf{G}(w))$. It is also straightforward to see that the communication volume and the time complexity for the server are both $O(\mathsf{G}(w))$.

Combining the analyses above, we conclude that the time complexity of a query for both the client and the server is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$, while the communication volume of a query is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$. We note that stash handling is not relevant to the retrieval of documents and it can be performed whenever the client is free.

E DYNAMIC SWISSSE

In this appendix, we extend SWiSSSE to handle dynamic databases. As in the case of static SWiSSSE, we design and analyze dynamic SWiSSSE while taking into account the leakage from both the (encrypted) index *and* the document retrieval component of a searchable encryption *system*. Our main technical contributions that are specific to dynamic SWiSSSE are summarized briefly below.

System-wide Security of Dynamic SSE. We propose new security definitions for dynamic SSE schemes that take into account the system-wide leakage during setup, searches, *and updates*. Once again, are for an honest-but-curious server and we use the real world/ideal world simulation paradigm with leakage profiles to model security. In contrast to previous analysis for dynamic SSE [16, 19, 26, 45], we take a system-wide view, including in our execution model and leakage profile the action of retrieving encrypted documents from the data storage component as well as the leakage from the encrypted search index. See Appendix E.1 for details.

Oblivious Operations. For dynamic SSE, we additionally introduce a new system-wide security notion called “obliviousness of operations", which requires that search and update query operations should be computationally indistinguishable to the server. This brings several advantages. First of all, it naturally implies that search and update operations incur computationally indistinguishable leakage. This allows for a unified security definition with respect to searches and updates for dynamic SSE schemes, as opposed to the separate definitions in prior work.

Recent works [16, 19, 26, 45] have put forth two new notions of security that any dynamic SSE scheme is expected to satisfy: (a) *forward privacy* (which requires that updating a document in the database should not reveal whether the updated document contains keywords that have been previously searched for) and (b) *backward privacy* (which requires that searching for a keyword should reveal no information about files previously containing this keyword that have subsequently been deleted from the database). It turns out that obliviousness of operations yields stronger notions of forward and backward privacy than considered in these recent works. For instance, it implies a notion of forward privacy that requires update operations (both inserts and deletes) to hide not only the content but also the size (in terms of number of keywords) of the document(s) being updated, something not addressed in previous work. Obliviousness of operations also implies a notion of backward privacy that requires searches to completely hide both the result pattern and the update history associated with the underlying keyword. This is a strong enough security guarantee to effectively rule out a powerful class of file injection attacks [110] that can otherwise compromise all existing notions of backward privacy in the literature. See Appendix E.2 for a more detailed overview, and Appendices E.3 and E.4 for rigorously formal definitions and analyses.

Storage	Stash	$O(\max_w \mathsf{G}(w) + \max_w \|W\{\mathsf{DB}(w)\})\|)$
	EDB/DB	$O(\sum_w \mathsf{G}(w) + \|\mathsf{DB}\|)$
Time complexity	Document retrieval	$O(\mathsf{G}(w))$
	Write-back	$O(\max_w \mathsf{G}(w) + \max_w \|W\{\mathsf{DB}(w)\}\|)$
Communication volume	Document retrieval	$O(\mathsf{G}(w))$
	Write-back	$O(\max_w \mathsf{G}(w) + \max_w \|W\{\mathsf{DB}(w)\}\|)$

Figure 4: A summary of the performance parameters. Here, w denotes the leading keyword of a query, $\mathsf{G}(w)$ is the bucket size of keyword w, $|W\{\mathsf{DB}\}|$ is the total number of keyword-document pairs, and $|W\{\mathsf{DB}(w)\}|$ is the total number of keyword-document pairs for the documents that contain w.

E.1 System-Wide Definitions for Dynamic SSE

System-Wide Syntax for Dynamic SSE. We begin by formally defining our system-wide syntax for a dynamic SSE scheme Σ consisting of *four* protocols: **Setup**, **KWQuery Insert** and **Delete** between a client and a server. A query q from the client to the server takes the form $(op, args)$ where op is the operation of the query, and $args$ are the additional arguments required for the query. We note that **Setup** and **KWQuery** are identical for static and dynamic SSE, but we repeat them below for the sake of completeness.

- **Setup**$(1^\lambda, \mathsf{DB})$ is a protocol between the client and the server. The client takes as input a security parameter 1^λ and a database DB, and outputs $(\mathsf{EDB}, \mathsf{st_{Clt}})$, where EDB is an encrypted database (including the encrypted search index as well as the encrypted documents) and $\mathsf{st_{Clt}}$ is the client's internal state. The client sends EDB to the server and the server runs $\mathsf{Svr}.$**Setup**(EDB) to setup the encrypted database.
- **KWQuery**$(sk, q, \mathsf{st_{Clt}}; \mathsf{EDB})$ is a protocol between the client and server to support single-keyword query. The query q in this case is of the form (**KWQuery**, w), where w is the target keyword of the query. The client takes as input a secret key sk, a search query q, and its internal state $\mathsf{st_{Clt}}$. The server takes as input the encrypted database EDB. The client and server interact with each other and by the end of the interaction, the client gets $(r, \mathsf{st_{Clt}})$ where r is the result of the query (i.e., the actual documents matching the query) and $\mathsf{st_{Clt}}$ is the new state of the client; the server state EDB may also be modified following the execution.
- **Insert**$(sk, q, \mathsf{st_{Clt}}; \mathsf{EDB})$ is a protocol between the client and server to support update on the database. An insertion query q takes the form (**Insert**, $\{w\}$, d), where d is the document to be inserted and $\{w\}$ are the keywords associated to the document. The client takes as input a secret key sk, an update query q, and his internal state $\mathsf{st_{Clt}}$. The server takes as input the encrypted database EDB. The client and server interact with each other and by the end of the interaction, the client gets $(r, \mathsf{st_{Clt}})$ where r is the response from the server and $\mathsf{st_{Clt}}$ is the new state of the client; the server gets EDB where EDB is the encrypted database after the insertion operation.
- **Delete**$(sk, q, \mathsf{st_{Clt}}; \mathsf{EDB})$ is a protocol between the client and server to support update on the database. An insertion query q takes the form (**Delete**, d), where d is the document to be deleted. The client takes as input a secret key sk, an update query q, and his internal state $\mathsf{st_{Clt}}$. The server takes as input the encrypted database EDB. The client and server interact with each other and by the end of the interaction, the client gets $(r, \mathsf{st_{Clt}})$ where r is the response from the server and $\mathsf{st_{Clt}}$ is the new state of the client; the server gets EDB where EDB is the encrypted database after the insertion operation.

Once again, note that unlike prior (index-only) dynamic SSE definitions [16, 19, 26, 45], our syntax for dynamic SSE takes into account the actual encrypted documents (as part of the encrypted database EDB output by **Setup** and as part of the query response r output by **KWQuery**, **Insert** and **Delete**) in addition to the encrypted search index.

System-Wide Security of Dynamic SSE. We now present our new system-wide security definitions for dynamic SSE. Similar to its static counterpart in Section A, we formulate system-wide security of dynamic SSE in terms of a *leakage* function. In contrast to existing definitions for dynamic SSE [16, 19, 26, 45], our definitions also capture the information leakage from the document retrieval phase. We say that a dynamic SSE scheme is system-wide secure with respect to a leakage function $\mathcal{L} = (\mathcal{L}^{\mathbf{Setup}}, \mathcal{L}^{\mathbf{KWQuery}}, \mathcal{L}^{\mathbf{Insert}}, \mathcal{L}^{\mathbf{Delete}})$ if no (honest-but-curious) adversary can distinguish between the real execution and an execution simulated given only the leakage, as outlined below.

Definition E.1 (System-Wide Security of Dynamic SSE). Let Σ = (**Setup**, **KWQuery**, **Insert**, **Delete**) be a dynamic SSE scheme and consider the following probabilistic experiment where $\mathcal{A}$ is a stateful, probabilistic polynomial time (PPT) honest-but-curious adversary, $\mathcal{S}$ is a stateful simulator and $\mathcal{L}$ is the leakage function.

$\underline{\mathbf{Real}^{\mathsf{Dynamic}}_{\Sigma,\mathcal{A}}(1^\lambda):}$

(1) The adversary $\mathcal{A}$ selects a database DB and gives it to the challenger C.
(2) The challenger C generates a key sk, and encrypts the database as $\mathsf{EDB} \leftarrow$ **Setup**$(1^\lambda, sk, \mathsf{DB})$. The challenger C sends the encrypted database EDB to the adversary $\mathcal{A}$.
(3) The adversary picks a polynomial number of queries $q_1, \ldots, q_{\mathsf{poly}(\lambda)}$ (where each query could be *either* a keyword search query or an *update query*). For each query, the challenger C interacts with the adversary $\mathcal{A}$ to execute the corresponding query protocol (including the final document retrieval phase), where the challenger plays the client and the adversary plays the server.
(4) Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

$\underline{\mathbf{Ideal}^{\mathsf{Dynamic}}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda, \mathcal{L} = (\mathcal{L}^{\mathbf{Setup}}, \mathcal{L}^{\mathbf{KWQuery}}, \mathcal{L}^{\mathbf{Insert}}, \mathcal{L}^{\mathbf{Delete}})):}$

(1) The adversary $\mathcal{A}$ selects a database DB and gives $\mathcal{L}^{\textbf{Setup}}(\mathsf{DB})$ to the simulator $\mathcal{S}$.
(2) Using $\mathcal{L}_{\textbf{Setup}}(\mathsf{DB})$, the simulator $\mathcal{S}$ generates EDB and returns it to the adversary $\mathcal{A}$.
(3) The adversary picks a polynomial number of queries $q_1, \ldots, q_{\mathtt{poly}(\lambda)}$ (where each query could be *either* a keyword search query or an *update query*). For each query, the simulator $\mathcal{S}$ computes the transcript of the query using the appropriate component of the leakage function $\mathcal{L}$ (including the leakage from the document retrieval phase corresponding to the query), and sends it to the adversary $\mathcal{A}$.
(4) Finally, the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.

We say that a dynamic SSE scheme Σ is $\mathcal{L}$-secure if there exists a probabilistic polynomial-time (PPT) simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, we have

$$\Big|\Pr\Big[\mathbf{Real}^{\mathsf{Dynamic}}_{\Sigma,\mathcal{A}}(1^\lambda) = 1\Big] - \Pr\Big[\mathbf{Ideal}^{\mathsf{Dynamic}}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda, \mathcal{L}) = 1\Big]\Big| \leq \mathtt{negl}(\lambda).$$

Differences with Index-Only Security Definitions. Unlike prior (index-only) dynamic SSE definitions [16, 19, 26, 45], our security definition for dynamic SSE takes into account the actual encrypted documents and the leakage from the document retrieval phase. This follows from the facts that: (a) the encrypted database EDB in both the real and ideal worlds includes the actual encrypted documents, and (b) the adversary $\mathcal{A}$ gains access to the leakage from the document retrieval phase of *both search and update queries* in the real and ideal worlds.

Adaptive vs Non-adaptive Adversaries. Finally, as in our system-wide definitions for static SSE, we can distinguish between non-adaptive and adaptive adversaries in the above definition, depending on how the adversary chooses its queries. We say that Σ is non-adaptively (resp. adaptively) secure if the adversary $\mathcal{A}$ chooses all the queries before executing them (resp. chooses each of his queries based on the transcripts resulting from his previous queries).

System-Wide Correctness of Dynamic SSE. Our system-wide correctness definition for dynamic SSE is very similar in flavor to the corresponding definition for static SSE, except that we now consider correctness for both keyword search queries and update queries (including the correctness of the document retrieval/update phase for both classes of queries). Concretely, consider a game between a challenger and a PPT adversary $\mathcal{A}$ very similar to the the game $\mathbf{Real}^{\mathsf{Dynamic}}_{\Sigma,\mathcal{A}}(1^\lambda)$ described above with the adversary $\mathcal{A}$ issuing at most $k = \mathtt{poly}(\lambda)$ keyword search or update queries during the game, except that at the end of the game, a PPT distinguisher $\mathcal{D}$ receives one of the following:

- the data obtained by executing the plaintext queries against the unencrypted/plaintext database (including the document retrieval phase), or
- the data obtained from the execution of the **KWQuery** protocol (for keyword search queries) between the challenger C and the adversary $\mathcal{A}$ (once again, including the document retrieval phase).
- the data obtained from the execution of the **Insert** and **Delete** protocols (for update queries) between the challenger C and the adversary $\mathcal{A}$ (including the corresponding updates to the actual encrypted documents).

Let $Adv^{corr,k,\mathsf{Dynamic}}_{\Sigma,\mathcal{A},\mathcal{D}}(1^\lambda)$ denote the probability that the distinguisher $\mathcal{D}$ distinguishes between the aforementioned data distributions. We say that the dynamic SSE scheme Σ is correct from a system-wide perspective if for any such PPT adversary $\mathcal{A}$ and any such PPT distinguisher $\mathcal{D}$, we have

$$Adv^{corr,k,\mathsf{Dynamic}}_{\Sigma,\mathcal{A},\mathcal{D}}(1^\lambda) \leq \mathtt{negl}(\lambda).$$

Once again, our correctness definition for dynamic SSE takes into account the actual encrypted documents returned/updated by the query, as opposed to only the encrypted document identifiers resulting from querying the index considered by the prior (index-only) dynamic SSE definitions [16, 19, 26, 45].

E.2 Overview of Dynamic SWiSSSE

In this section, we present an informal overview how we extend SWiSSSE to handle dynamic databases. See Appendix E.3 for the detailed formal description.

We consider two kinds of updates to the database – document insertion and document deletion; a document update can be simulated via: (a) a deletion operation on the old document, followed by (b) an insertion operation on the modified document. We first present a simple idea for handling document insertions. At a high level, we use a technique similar to the auxiliary write-backs used in our static construction. This incurs some undesirable leakage, which we address subsequently.

Handling Insertions—Simple Version. When a document d_ℓ is to be inserted, the client simply schedules: (a) a normal document write-back for d_ℓ targeting a set of "insert write-backs" for every keyword $w_i \in d_\ell$. As with auxiliary write-backs, insert write-backs target a separate set of addresses to avoid any correlation with prior write-backs (normal and auxiliary) corresponding to the same keyword. More concretely, we now generate three separate sets of addresses for normal, auxiliary and update write-backs involving the same keyword:

$$\mathsf{addr}_{\text{norm}}(w_i, d_\ell, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(3 * \mathsf{cnt}_{w_i})),$$
$$\mathsf{addr}_{\text{aux}}(w_i, d_\ell, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(3 * \mathsf{cnt}_{w_i} + 1)),$$
$$\mathsf{addr}_{\text{insert}}(w_i, d_\ell, \mathsf{cnt}_{w_i}) = F(K, w_i||j||(3 * \mathsf{cnt}_{w_i} + 2)),$$

where F is a PRF, j is a counter that runs from 0 to $|\mathsf{DB}(w_i)| - 1$ (where $\mathsf{DB}(w_i)$ denotes the set of documents containing keyword w_i), and cnt_{w_i} is a per-key word counter held in the client's stash which records how many times w_i has appeared in search *and* insertion queries.

In other words, during the time interval between the t^{th} and $(t+1)^{\text{th}}$ queries on keyword w_i, we use *three* sets of write-back addresses – the set $\{\mathsf{addr}_{\text{norm}}(w_i, d_\ell, j)\}$ for normal write-backs, the set $\{\mathsf{addr}_{\text{aux}}(w_i, d_\ell, j)\}$ for auxiliary write-backs, and the set $\{\mathsf{addr}_{\text{insert}}(w_i, d_\ell, j)\}$ for insert write-backs. The insert write-backs happen intermittently and can be randomly interspersed with normal and auxiliary write-backs involving other keywords and documents.

During a search query involving w_i, the client now requests the server to access all three sets of write-back addresses – normal, auxiliary and insert – in the keyword lookup index. The entries corresponding to the normal and insert write-back addresses allow the client to recover the pointers to already existing documents and freshly inserted documents, respectively, that contain w_i. The entries corresponding to the auxiliary write-back addresses allow the client to identify if any of these pointers have been updated subsequently due to searches involving other keywords. Thus, search correctness is ensured. Finally, as before, we use additional pointers to fake documents to hide the exact frequency of the keyword w_i, and reveal its bucket size instead.

Leakage. The solution outlined above leaks that a new document has been inserted: when the client executes a normal document write-back operation for the newly inserted document d_ℓ, the total number of actual and dummy addresses in the encrypted document array increases by one. While this leakage is currently incurred by all existing dynamic SSE schemes, it has some repercussions with respect to file injection attacks [110]. For this attack vector to work, the adversary needs to infer exactly when an insert operation corresponding to a maliciously constructed file occurs, as well as the effect of this insertion on subsequent keyword search operations.

This motivates hiding the occurrences of inserts from the server, and hence, masking the aforementioned leakage. We describe how to achieve this next.

Handling Insertions—Modified Version. An effective way to mask when a document is inserted is to avoid creating a fresh entry in the encrypted document array. Instead, we simply convert an (already existing) dummy entry into a real one.

Concretely, to insert a fresh document d_ℓ, the client first identifies a "leading keyword" w^* in d_ℓ. We assume without loss of generality that w^* is the keyword in d_ℓ with the smallest occurrence frequency in the database. Next, the client issues a search query on w^* and retrieves a list of pointers to real and dummy locations in the document array. To insert the new document, the client schedules a normal document write-back targeting one of the dummy addresses, as opposed to a newly generated address. The insert write-backs are scheduled exactly as described in the simple version above, except they now encapsulate a pointer to the dummy address as opposed to some newly generated address.

Handling Deletions. Finally, deletions are handled in a manner that is complementary to the insertion procedure described above. Namely, when a document is to be deleted, we convert the real entry corresponding to this document in the document array into a dummy entry with some garbage ciphertext. More concretely, the client again issues a search query on w^*, and schedules a dummy document write-back targeting the address corresponding to the document to be deleted. The insert write-backs are scheduled exactly as for the inserts, except they now encapsulate pointers to random addresses in the document array.

Note that in the above strategy, there is the possibility that we run out of dummy addresses in the document array after a certain number of insert operations. For simplicity of presentation and analysis, we implicitly assume a cap (determined at setup) on the maximum number of new document insertions supported by the system. We refer the reader to Appendix E.3 for a more detailed discussion on how to generalize the above proposal to support an uncapped number of insertions.

We refer the reader to Appendix E.3 for a formal analysis of how dynamic SWiSSSE achieves correctness of searches and updates while handling insertions and deletions as outlined above.

Oblivious Operations. Dynamic SWiSSSE naturally supports "oblivious operations". Both keyword searches and document updates involve reading a set of entries from the encrypted data structures, followed by delayed write-backs. The only functional differences between searches and updates are reflected in how the client locally manages/updates its stash. From the point of view of the server, the output of the leakage function at the point of query is simply the accesses made to the encrypted data structures, which is unconditionally indistinguishable for searches and updates. We formalize the notion of oblivious operations and prove that our dynamic scheme achieves this notion in Appendix E.4.

Forward Privacy. Dynamic SWiSSSE achieves stronger forward privacy guarantees than existing constructions in the literature, *including* those based on ORAM [16, 19, 26, 45]. Existing schemes satisfy a definition of forward privacy that only requires insertion and deletion operations to computationally hide the set of keywords in the target document. However, they do not hide the *number of keywords* an inserted/deleted document contains, which is potentially sensitive information. Our construction, on the other hand, achieves the stronger notion of forward privacy in which we also hide from the server the number of keywords in a document which is inserted/deleted. We formalize this in Appendix E.4.

Backward Privacy. Dynamic SWiSSSE also achieves stronger backward privacy guarantees than existing constructions in the literature. The strongest notion of backward privacy achieved thus far is called Type-1 backward privacy [19], and the only constructions to achieve it are based on full-fledged ORAM-style techniques [19, 26]. This notion allows the adversary to learn, for every search query, the corresponding result pattern and the timestamps at which the documents containing the queried keyword were inserted. We achieve a stronger notion of backward privacy that hides both these forms of leakage from the adversarial server.

To see why this is the case, recall that our construction computationally hides the result pattern for each query, since the adversarial server only sees encrypted documents and pointers to documents (in fact, the server sees fresh encryptions of these items for every write-back operation). Secondly, due to the delayed write-backs, the locations of keywords and documents change after every search and update operation. Therefore, it is difficult for the adversary to trace each encrypted document it accesses during a search query at timestamp t back to the timestamp $t' < t$ when the document was originally inserted. We formally describe this in Appendix E.4.

Resistance to Leakage Cryptanalysis. Finally, it turns out these stronger notions of forward and backward privacy makes SWiSSSE more resilient to known leakage-abuse and file-injection attacks as compared to existing dynamic schemes, including those based on ORAM-style techniques [19, 26]. We refer the reader to Appendix E.4 for a more detailed explanation.

Algorithm 8 Dynamic SWiSSSE.**Setup**

```
procedure Clt.Setup(DB)
    /* Generate fake documents */
    DB′ ← Fake_Doc_Gen(DB, Clt.G)
    Clt.N ← |DB′|
    EI, EA ← {}
    for i = 1, ..., |DB′| do
        /* Get the set of keywords with counters */
        x ← {(w, Clt.KWCtr[w]) | w ∈ W(DB′[i])}
        /* Update the lookup index */
        for w ∈ W(DB′[i]) do
            EI ← EI ∪ (F(w||Clt.KWCtr[w]||0), Enc(id(DB′[i])))
            Clt.KWCtr[w] ← Clt.KWCtr[w] + 1
        /* Insert the encrypted document */
        EA ← EA ∪ (F(i||0), Enc(x||DB′[i]))
    /* Reset the keyword counter */
    for w ∈ W(DB′) do
        Clt.KWCtr[w] ← 0
    /* Initialise the stash */
    Clt.I.init()
    Clt.A.init()
    Send (EI, EA) to the server

procedure Svr.Setup(EI, EA)
    Svr.EI.init()
    Svr.EA.init()
    Svr.EI.put(EI)
    Svr.EA.put(EA)
```

E.3 Dynamic SWiSSSE: Detailed Description

In this section, we present a formal description of the various protocols involved in dynamic SWiSSSE.

Setup. The setup procedure for the dynamic construction is very similar to the static one. The only differences are that the client has to initialise an array Clt.InsCtr to keep track of the number of insertions for each keyword since the last time they have been queried as the leading keywords.

SWiSSSE.{**KWQuery**, **Insert**, **Delete**}. We now describe the keyword query, insert and delete procedures for dynamic SWiSSSE. For ease of representation, these procedures broken up into smaller sub-routines described subsequently.

Encrypted Document Array Address Retrieval. This sub-routine is the same for a search query, document insertion or document deletion, and is described in Algorithm 9, and is very similar to the corresponding sub-routine for document array address retrieval in the static version of SWiSSSE.**KWQuery**, except that the client now fetches three sets of addresses - normal, auxiliary and insert. For document insertion, the client simply queries the first keyword in the document he wants to insert. For document deletion, the client queries the first keyword in the document he wants to delete. As the index for the inserted keywords are stored by the server, the client has to compute some additional virtual addresses to retrieve the documents.

Encrypted Document Retrieval. The sub-routine is again identical for a search query, document insertion and document deletion, and works in the same way as the corresponding sub-routine for the static version of SWiSSSE (see Algorithm 3 for the details of how this sub-routine works).

Algorithm 9 Dynamic SWiSSSE.{**KWQuery**, **Insert**, **Delete**}: Encrypted Document Array Address Retrieval

```
procedure Clt.TokenGen(w)
    L ← {}
    for j ∈ 0, ..., Clt.G(w) − 1 do
        L ← L ∪ {F(w||j||3 ∗ Clt.KWCtr[w])},
        L ← L ∪ {F(w||j||3 ∗ KWCtr[w] + 1)},
        L ← L ∪ {F(w||j||3 ∗ Clt.KWCtr[w]) + 2}.
    /* Roll forward the counter for the next query */
    Clt.KWCtr[w] ← Clt.KWCtr[w] + 1
    Send L to the server

procedure Svr.Index_Lookup(L)
    Send Svr.EI.get(L) to the client
```

Algorithm 10 Dynamic SWiSSSE.**KWQuery**: Write-Back Sub-Routine

```
procedure Clt.Write_Back_Keyword_Query(M, w̄)
    Replace the lookup addresses of the newly inserted documents which contain w̄ with the addresses used for the fake documents.
    UA ← {}
    /* Get random documents from the stash */
    D ← Clt.A.pop(|Clt.A|)
    for ({(w_i, j_i, b_i)}, d) ∈ UA do
        /* Encrypt the new document addresses and documents */
        UA ← UA ∪ {(F(id(d)||Clt.ArrCtr[id(d)]), Enc({(w_i, j_i)}||d))}
        /* Update the stash for the lookup index */
        for (w, j, b) ∈ {(w_i, j_i, b_i)} do
            Clt.I.put((F(w||j||Clt.KWCtr[w] + b), Enc(id(d))))
    /* Decrypt the documents retrieved and insert them into the document array */
    Clt.A.put(Dec(M))
    Send (Clt.I.pop(⌊|Clt.I|/2⌋), UA)

procedure Svr.Write_Back((UI, UA))
    Svr.EI.put(UI)
    Svr.EA.put(UA)
```

The final set of sub-routines are the write-back sub-routines corresponding to search queries, insertions and deletions. Unlike the previous sub-routines, write-backs are executed differently for each query type. We describe these next.

Write-Back for Search Query. The write-back sub-routine under dynamic SWiSSSE.**KWQuery** is described in Algorithm 10. Technically, it is very similar to that under static SWiSSSE.**KWQuery** (Algorithm 4), except that the client has to perform some maintenance on the lookup index for the queried keyword to relocate the addresses for the document insertions to the ones used for fake documents.

We explain this idea in greater detail. Recall that during the encrypted document array address retrieval phase, we have obtained all the normal write-back addresses of the form $F(w||j||3 * \texttt{KWCtr}[w])$, the auxiliary write-back addresses of the form $F(w||j||3 * \texttt{KWCtr}[w] + 1)$ and insertion write-back addresses of the form

Algorithm 11 Dynamic SWiSSSE.**Insert**: Write-Back Sub-Routine

procedure Clt.**Write_Back_Insertion**$(M, \{\bar{w}_j\}, \bar{d})$
  Replace the lookup addresses of the newly inserted documents which contain $\bar{w}$ with the addresses used for the fake documents.
  $UA \leftarrow \{\}$
  /* *Get random documents from the stash* */
  $D \leftarrow$ Clt.A.**pop**(|Clt.A|)
  for $(\{(w_i, j_i, b_i)\}, d) \in UA$ **do**
    /* *Encrypt the new document addresses and documents* */
    $UA \leftarrow UA \cup \{(F(id(d)||\texttt{Clt.ArrCtr}[id(d)]), \textbf{Enc}(\{(w_i, j_i)\}||d))\}$
    /* *Update the stash for the lookup index* */
    for $(w, j, b) \in \{(w_i, j_i, b_i)\}$ **do**
      Clt.I.**put**$((F(w||j||\texttt{Clt.KWCtr}[w] + b), \textbf{Enc}(id(d))))$
  /* *Decrypt the documents retrieved and insert them into the document array* */
  Clt.A.**Insert**(**Dec**(M))
  Insert document $\bar{d}$ with keywords $\{\bar{w}_j\}$ into Clt.A
  Send (Clt.I.**pop**$(\lfloor|\texttt{Clt.I}|/2\rfloor), UA)$

procedure Svr.**Write_Back**$((UI, UA))$
  Svr.EI.**put**(UI)
  Svr.EA.**put**(UA)

Algorithm 12 Dynamic SWiSSSE.**Delete**: Write-Back Sub-Routine

procedure Clt.**Write_Back_Deletion**$(M, \bar{d})$
  Replace the lookup addresses of the newly inserted documents which contain $\bar{w}$ with the addresses used for the fake documents.
  $UA \leftarrow \{\}$
  /* *Get random documents from the stash* */
  $D \leftarrow$ Clt.A.**pop**(|Clt.A|)
  for $(\{(w_i, j_i, b_i)\}, d) \in UA$ **do**
    /* *Encrypt the new document addresses and documents* */
    $UA \leftarrow UA \cup \{(F(id(d)||\texttt{Clt.ArrCtr}[id(d)]), \textbf{Enc}(\{(w_i, j_i)\}||d))\}$
    /* *Update the stash for the lookup index* */
    for $(w, j, b) \in \{(w_i, j_i, b_i)\}$ **do**
      Clt.I.**put**$((F(w||j||2*\texttt{Clt.KWCtr}[w]+b), \textbf{Enc}(id(d))))$
  /* *Decrypt the documents retrieved and insert them into the document array* */
  Clt.A.**put**(**Dec**(M))
  Turn $\bar{d}$ into a fake document in Clt.A
  Send (Clt.I.**pop**$(\lfloor|\texttt{Clt.I}|/2\rfloor), UA)$

procedure Svr.**Write_Back**$((UI, UA))$
  Svr.EI.**put**(UI)
  Svr.EA.**put**(UA)

$F(w||j||3 * \texttt{KWCtr}[w] + 2)$. Our goal is to remove the additional insertion addresses of the form $F(w||j||\texttt{KWCtr}[w] + 2)$ by making use of the fake documents that contain the keyword w. In terms of the documents, this means for each newly inserted document, we find a fake document that contains w, remove the keyword from the fake document, and allocate it to the newly inserted document. We omit the low-level details of the procedure for readability.

Write-Back for Document Insertion. The write-back sub-routine under dynamic SWiSSSE.**Insert** is described in Algorithm 11. Technically, it is essentially identical to the corresponding sub-routine under dynamic SWiSSSE.**KWQuery** except that the client has to insert the document locally. This is done by scanning the query response for fake documents, and replace one of them by the document that is to be inserted. The keyword pointers are updated so as to maintain correctness of future searches.

Write-Back for Document Deletion. The write-back sub-routine under dynamic SWiSSSE.**Delete** is described in Algorithm 12. Technically, it is again identical to the corresponding sub-routine under dynamic SWiSSSE.**KWQuery** except that the client has to overwrite the target document to a fake document in the stash.

Supporting Uncapped Number of Insertions. As one can clearly see from the bucketization strategy and the fake document generation procedure in our construction, there is a limit on how many documents the client can insert into the database. One possible work-around is to instantiate a new encrypted database every time the maximum quota is hit. This may not be practical for some systems as the client storage grows linearly in the number of instances of encrypted databases.

As an alternative, we can extend our dynamic construction to support uncapped document insertions at the cost of additional leakage. Without loss of generality, suppose that the client wants to store a documents more. He can simply insert a fake documents in the stash and redirect some of the pointers of the fake keywords (which he can obtain from normal queries) to these new fake documents. These new fake documents can then be written back to the server just like the normal documents. If the client wants to store a additional documents for a particular keyword w, he can make a search query on w to retrieve the documents associated to w, increase the address space of w by a keywords, and generate a fake documents and point the newly generated keyword pointers to the new fake documents. These pointers and documents can then be written-back to the server with normal write-back operations. On a side note, the client should choose a such that the new bucket size of w corresponds to the bucket size of some other keyword, so that the volume leakage does not trivially leak the identity of w in the future queries.

We leave it as an interesting future work to formalize the storage expansion process, and to analyze the additional leakage thereof.

Correctness. Similar to the static case, there is a possibility for our dynamic construction to fail if the client generates repeated addresses. We provide an upper bound of the failure probability of our dynamic construction with adversarially chosen queries below. As the proof is almost identical to the static case, we omit the proof from the paper.

Theorem E.1. [Correctness of Dynamic SWiSSSE]
Let $|\texttt{DB}|$ and $|W\{\texttt{DB}\}|$ denote the total number of documents and document-keyword pairs, respectively, in the database DB at any given point of time, and let l denote the output length of the PRF F used in static SWiSSSE. Then the advantage of any adversary $\mathcal{A}$, which issues at most k queries, in breaking the correctness of static

Algorithm 13 Dynamic SWiSSSE: Leakage Function for Keyword Queries

procedure $\mathcal{L}^{\textbf{KWQuery}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\textbf{KWQuery}, \bar{w}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$
  $\textbf{IndHist} \leftarrow \textbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1]), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 1), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w}_1) - 1\}$
  $\mathtt{KWCtr}[\bar{w}_1] \leftarrow \mathtt{KWCtr}[\bar{w}] + 3$
  $L \leftarrow \mathtt{I}[\bar{w}]$
  while $|L| < 2 \cdot \mathtt{Clt.G}(w)$ **do**
    $id \leftarrow \textbf{Rand}(|\mathtt{A}|)$
    if $id \notin \{id(d) \mid d \in \mathtt{A}'\}$ **then**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\textbf{ArrHist} \leftarrow \textbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

  $UI \leftarrow \mathtt{I}'.\textbf{pop}(|\mathtt{I}'|/2)$
  $\textbf{IndHist} \leftarrow \textbf{IndHist} \cup \{(i, 1, k) \mid i \in UI\}$
State $UA \leftarrow \mathtt{A}'.\textbf{Pop}(\lfloor |UA|/2 \rfloor)$
  $\textbf{ArrHist} \leftarrow \textbf{ArrHist} \cup \{(\mathbf{T}(id(d), \mathtt{ArrCtr}[id(d)]), 1, k) \mid (\{w_i, j_i, b_i\}, d) \in UA\}$
  $\mathtt{A}' \leftarrow \mathtt{A}' \cup \textbf{Merge\_Index}(\mathtt{A}[L], \bar{w})$
  $\mathbf{St}_{\mathcal{L}} \leftarrow (\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr})$
  Return $(\textbf{IndHist}, \textbf{ArrHist}), \mathbf{St}_{\mathcal{L}}$

SWiSSSE over the database DB is at most:

$$\frac{\left(|\mathsf{DB}|^2 + 4t_0|\mathsf{DB}| + 9|W\{\mathsf{DB}\}|^2 + 18t_1|W\{\mathsf{DB}\}|\right)}{2^{l+1}} + Adv_{F,\mathcal{B}}^{PRF,|\mathsf{DB}|+2t_0} + Adv_{F,C}^{PRF,2|W\{\mathsf{DB}\}|+2t_1},$$

where $t_0 = k \cdot \max_w |\mathsf{DB}(w)|$, $t_1 = k \cdot \max_w |w\{\mathsf{DB}(w)\}|$ and $\mathcal{B}$ and C denote probabilistic polynomial-time adversaries in independent security experiments against the PRF F.

E.4 System-Wide Leakage of Dynamic SWiSSSE

Setup. At setup, the client offloads the encrypted lookup index and the encrypted document array to the server. These data structures are essentially key-value stores with pseudorandomly generated keys/addresses and values/entries that are encrypted under an IND-CPA secure encryption scheme. Hence, at setup, the server learns no information about the original database DB other than the number of documents in the padded database DB′ (including both real and fake documents), and the total number of keyword-document pairs post-bucketization. Formally, we have:

$$\mathcal{L}_{\Sigma}^{\textbf{Setup}}(\mathsf{DB}, \mathsf{G}) = (|\mathsf{DB}'|, |W\{\mathsf{DB}'\}|, \mathbf{St}_{\mathcal{L}}).$$

Keyword Queries. As we have introduced virtual addresses for the inserted documents, the insertion history will be revealed by the keyword queries. As in the static case, we capture this leakage using a probabilistic and stateful leakage function, described formally in Algorithm 13.

Document Insertion. The leakage of a document insertion is identical to a single-keyword query except that the inserted document is processed in the state of the leakage. We capture this leakage using a probabilistic and stateful leakage function, described formally in Algorithm 14.

Algorithm 14 Dynamic SWiSSSE: Leakage Function for Insertion Queries

procedure $\mathcal{L}^{\textbf{Insert}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\textbf{Insert}, \{\bar{w}_i\}, \bar{d}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$
  $\textbf{IndHist} \leftarrow \textbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1]), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 1), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w}_1) - 1\}$
  $\mathtt{KWCtr}[\bar{w}_1] \leftarrow \mathtt{KWCtr}[\bar{w}_1] + 3$
  $L \leftarrow \mathtt{I}[w]$
  while $|L| < 2 \cdot \mathtt{Clt.G}(w)$ **do**
    $id \leftarrow \textbf{Rand}(|\mathtt{A}|)$
    if $id \notin \{id(d) \mid d \in \mathtt{A}'\}$ **then**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\textbf{ArrHist} \leftarrow \textbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

  $UI \leftarrow \mathtt{I}'.\textbf{pop}(|\mathtt{I}'|/2)$
  $\textbf{IndHist} \leftarrow \textbf{IndHist} \cup \{(i, 1, k) \mid i \in UI\}$
  $UA \leftarrow \mathtt{A}'.\textbf{Pop}(\lfloor |UA|/2 \rfloor)$
  $\textbf{ArrHist} \leftarrow \textbf{ArrHist} \cup \{(\mathbf{T}(id(d), \mathtt{ArrCtr}[id(d)]), 1, k) \mid (\{w_i, j_i, b_i\}, d) \in UA\}$
  $\mathtt{I}' \leftarrow \mathtt{I}' \cup \textbf{Index}(UA, \mathtt{KWCtr})$
  $M \leftarrow \textbf{Insert}(\mathtt{A}[L], \{\bar{w}_i\}, d))$
  $\mathtt{A}' \leftarrow \mathtt{A}' \cup \textbf{Merge\_Index}(M, \bar{w}_1)$
  $\mathbf{St}_{\mathcal{L}} \leftarrow (\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr})$
  Return $(\textbf{IndHist}, \textbf{ArrHist}), \mathbf{St}_{\mathcal{L}}$

Document Deletion. The leakage of a document deletion is identical to a single-keyword query except that the target document to be deleted is marked as fake in the state of the leakage.We capture this leakage using a probabilistic and stateful leakage function, described formally in Algorithm 15.

Finally, we are ready to state the security of our dynamic construction and prove it.

Theorem E.2 (Security of Dynamic SWiSSSE). *Let Σ be our proposed dynamic SSE scheme. Let $\mathcal{L}_{\Sigma}^{\textbf{Setup}}$ and $\mathcal{L}_{\Sigma}^{\textbf{KWQuery}}$, $\mathcal{L}^{\textbf{Insert}}$, and $\mathcal{L}^{\textbf{Delete}}$ be the leakage functions defined above, then Σ is ($\mathcal{L}_{\Sigma}^{\textbf{Setup}}$, $\mathcal{L}_{\Sigma}^{\textbf{KWQuery}}$, $\mathcal{L}^{\textbf{Insert}}$, $\mathcal{L}^{\textbf{Delete}}$)-secure.*

Proof. We use a game-based argument to prove the security of the dynamic construction.

(**Game 0**) Let the real execution of the scheme on the database DB with queries $q_1, \ldots, q_k$ be game G_0. Then we have that for any adversary $\mathcal{A}$,

$$\Pr\left[\mathbf{Real}_{\Sigma,\mathcal{A}}^{\mathsf{Dynamic}}(1^{\lambda}) = 1\right] = \Pr[G_0 = 1].$$

(**Game 1**) Let game G_1 be the same game as G_0 except that the execution of the setup step is replaced by the simulator. Clearly the simulator works the same way as the static case, so the difference in advantages between G_0 and G_1 is upper-bounded by $Adv_F^{PRF,t_0} + t_0 \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$, where $t_0 = 2\sum_w \mathsf{G}(w) + |\mathsf{DB}|$.

(**Game 2**) In game G_2, we replace the query algorithms with the simulator. The algorithms look the same for all query types so we

Algorithm 15 Dynamic SWiSSSE: Leakage Function for Deletion Queries

procedure $\mathcal{L}^{\textbf{DELETE}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\mathbf{Delete}, \{\bar{w}_i\}, \bar{d}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1]), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 1), 0, k), \mathbf{T}(\bar{w}, i, \mathtt{KWCtr}[\bar{w}_1] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w}_1) - 1\}$
  $\mathtt{KWCtr}[\bar{w}_1] \leftarrow \mathtt{KWCtr}[\bar{w}_1] + 3$
  $L \leftarrow \mathtt{I}[w]$
  while $|L| < 2 \cdot \mathtt{Clt.G}(w)$ **do**
    $id \leftarrow \mathbf{Rand}(|\mathtt{A}|)$
    if $id \notin \{id(d) \mid d \in \mathtt{A}'\}$ **then**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

  $UI \leftarrow \mathtt{I}'.\mathbf{pop}(|\mathtt{I}'|/2)$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(i, 1, k) \mid i \in UI\}$
  $UA \leftarrow \mathtt{A}'.\mathbf{Pop}(\lfloor |UA|/2 \rfloor)$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(id(d), \mathtt{ArrCtr}[id(d)]), 1, k) \mid (\{w_i, j_i, b_i\}, d) \in UA\}$
  $\mathtt{I}' \leftarrow \mathtt{I}' \cup \mathbf{Index}(UA, \mathtt{KWCtr})$
  $M \leftarrow \mathbf{Delete}(\mathtt{A}[L], \bar{d}))$
  $\mathtt{A}' \leftarrow \mathtt{A}' \cup \mathbf{Merge\_Index}(M, \bar{w}_1)$
  $\mathbf{St}_{\mathcal{L}} \leftarrow (\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr})$
  Return $(\mathbf{IndHist}, \mathbf{ArrHist}), \mathbf{St}_{\mathcal{L}}$

Algorithm 16 Game G_1 (dynamic construction). Only the setup step is changed.

procedure $\mathtt{Clt}.\textbf{Setup}(\mathtt{DB})$
  $(N, p, \mathbf{St}_{\mathcal{L}}) \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{Setup}}(\mathtt{DB}, \mathtt{G})$
  $\mathtt{EI}, \mathtt{EA} \leftarrow [\ ]$
  /* *Generate the encrypted documents* */
  for $i = 0, \ldots, N-1$ **do**
    $\mathtt{EA}.\mathbf{Insert}(\mathbf{RF}(2i), \mathbf{Enc}(0^{l_0}))$
  /* *Generate the encrypted document addresses* */
  for $i = 0, \ldots, 2p$ **do**
    $\mathtt{EI}.\mathbf{Insert}(\mathbf{RF}(2i+1), \mathbf{Enc}(0^{l_1}))$
  Send $(\mathtt{EI}, \mathtt{EA})$ to the server

only show the one for the single-keyword query. The simulator looks the same as game G_2 in the proof of security for the static case, but the lookup index tokens in the dynamic construction includes the addresses generated by the insertion queries too.

The number of addresses the algorithm has to generate is upper-bounded by $t_1 = 2\sum_i \mathtt{G}(W(q_i)) + 2\sum_i |\mathtt{DB}(W(q_i))|$, and the number of encryptions needs to be created is upper-bounded by the same t_1. This means the difference in advantages between G_1 and G_2 is upper-bounded by $Adv_F^{PRF,t_1} + t_1 \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$.

(**Conclusion**.) By combining the two games above, we see that the difference in advantages between G_0 and G_2 is at most $Adv_F^{PRF,t_0+t_1} + (t_0 + t_1) \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$.

Oblivious Operations. We introduce here a new notion of security for dynamic SSE schemes called "oblivious operations". Informally, a dynamic SSE scheme supports oblivious operations if document

Algorithm 17 Game G_2 (dynamic construction).

procedure $\mathtt{Clt}.\textbf{KWQuery}(q)$
  $(\mathbf{IndHist}, \mathbf{ArrHist}), \mathbf{St}_{\mathcal{L}} \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{KWQuery}}(\mathtt{DB}, q, \mathbf{St}_{\mathcal{L}})$

  /* *Encrypted document array address retrieval* */
  $L \leftarrow \{\}$
  $t' \leftarrow$ the number of single-keyword queries executed
  for $i \in \{i \mid (i, b, t) \in \mathbf{IndHist}, b = 0, t = t'\}$ **do**
    $L \leftarrow L \cup \mathbf{RF}(2i+1)$
  Send L to the server

  /* *Encrypted document retrieval* */
  $L \leftarrow \{\}$
  for $i \in \{i \mid (i, b, t) \in \mathbf{ArrHist}, b = 0, t = t'\}$ **do**
    $L \leftarrow L \cup \mathbf{RF}(2i)$
  Send L to the server

  /* *Write-back* */
  $UI, UA \leftarrow \{\}$
  for $i \in \{i \mid (i, b, t) \in \mathbf{IndHist}, b = 1, t = t'\}$ **do**
    $UI \leftarrow UI \cup (\mathbf{RF}(2i+1), \mathbf{Enc}(0^{l_0}))$
  for $i \in \{i \mid (i, b, t) \in \mathbf{ArrHist}, b = 1, t = t'\}$ **do**
    $UA \leftarrow UA \cup (\mathbf{RF}(2i), \mathbf{Enc}(0^{l_1}))$
  Send (UI, UA) to the server

updates and keyword searches are computationally indistinguishable to an adversarial server. The formal definition is presented below.

Definition E.2 (Oblivious Operations). *Let Σ be a dynamic SSE scheme. Let $\mathtt{DB}$ be a database, $\mathtt{G}$ be the bucketization parameter, $q_1, \ldots, q_{k-1}$ be a sequence of queries, and q_k and q'_k be two queries such that $W(q_k) = W(q'_k)$. Let $\ell_0, \mathbf{St}_{\mathcal{L}}^0 \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{Setup}}(\mathtt{DB})$ and $\ell_i, \mathbf{St}_{\mathcal{L}}^i \leftarrow \mathcal{L}_{\Sigma}^*(q_i, \mathbf{St}_{\mathcal{L}}^{i-1})$ for $0 < i \leq k$ where $\mathcal{L}_{\Sigma}^*$ is the appropriate leakage function for the query q_i, and $\ell'_k, \mathbf{St'}_{\mathcal{L}}^k \leftarrow \mathcal{L}_{\Sigma}^*(q'_k, \mathbf{St}_{\mathcal{L}}^{k-1})$.*

We say that Σ supports oblivious operations if ℓ_k is computationally indistinguishable from ℓ'_k for any choice of $\mathtt{DB}$, $\mathtt{G}$, $q_1, \ldots, q_k$ and q'_k.

Note that the definition of oblivious operations only requires the outputs of the leakage functions (at the point where the query is executed) to be indistinguishable. It does not, however, require the states of the leakage function to be indistinguishable. This makes sense because the leakage output is available to the adversary as soon as the corresponding operation is executed, which makes for an easy mapping task. On the other hand, the information contained in the state of the leakage function may be revealed to the adversary at a later point of time (for instance, via delayed pseudorandom write-backs in our scheme), and it is computationally hard for the adversary to map it back in time to the exact query it corresponds to.

Our dynamic SSE scheme naturally satisfies the aforementioned definition of oblivious operations. Both keyword searches and document updates involve reading a set of entries from the encrypted data structures, followed by delayed write-backs. The only functional differences between searches and updated are reflected in how the client locally manages/updates its stash. From the point of view of the server, the output of the leakage function at the point of query are simply the accesses made to the encrypted data structures, which is unconditionally indistinguishable for searches and updates. This allows us to state the following theorem.

Theorem E.3 (Oblivious Operations). *The dynamic variant* SWiSSSE *described above supports oblivious operations.*

Forward Privacy. Forward private SSE was introduced by Chang and Mitzenmacher in [28], and has been subsequently studied in [16, 18, 19, 40, 42, 70, 100, 101]. An SSE scheme is said to be forward private if insertion and deletion operations computationally hide the set of keywords in the underlying document. Forward privacy has received much attention in light of leakage-abuse and file injection attacks [21, 110], which are potentially devastating for SSE schemes that try to support updates without being forward private.

Observe that combining Theorems E.2 and E.3 allows us to claim that our dynamic SSE scheme achieves stronger forward privacy guarantees than existing constructions in the literature, *including* those based on ORAM [16, 19, 26, 42]. In particular, existing definitions of forward privacy do not hide the *number of keywords* an inserted/deleted document contains, which is potentially sensitive information. Our construction, on the other hand, achieves the stronger notion of forward privacy in which we also hide from the server the number of keywords in a document which is inserted/deleted.

We now present a more detailed argument. By Theorem E.3, our dynamic SSE scheme satisfies indistinguishability of operations. Hence, the output of the leakage function for updates is computationally indistinguishable from the output of the leakage function for keyword searches. Next, by Theorem E.2, the leakage function output for searches is the set of accesses made to the encrypted data structures at the server, which reveals no information to a computationally bounded adversary about the underlying keywords and documents Hence, at the point of an update operation, our dynamic scheme not only computationally hides the actual keywords in the target document, but also the number of keywords. As discussed later, this has important repercussions with respect to security against leakage-abuse and file-injection attacks.

Backward Privacy. The notion of backward privacy for dynamic SSE is comparatively more recent, and was first formalized by Bost *et al.* in [19]. Subsequently, Chamani *et al.* [26] and Sun *et al.* [103] proposed SSE schemes supporting single keyword search that are backward private under various leakage profiles. The strongest notion of backward privacy formalized in [19] is called Type-1 backward privacy [19]. A dynamic SSE scheme is said to be Type-1 backward private if a search query on a keyword w reveals no information to the adversary beyond result pattern for w and the timestamps at which the documents containing w were inserted into the database. The only constructions to achieve this strong notion of backward privacy adopt ORAM-style techniques and require polylogarithmically many communication rounds for searches [19, 26].

Once again, Theorem E.2 allows us to claim that our dynamic SSE scheme achieves stronger than Type-1 backward privacy guarantees. This is particularly notable given that our construction only require two rounds of communication between the client and the server for searches.

To begin with, observe that as per Theorem E.2 the leakage function output at the point of searches in our construction hides the result pattern and the update history for the underlying keyword from the server. If the adversary could monitor the state of the leakage function from the beginning of time up until the point of query, it could potentially learn the update history associated with a keyword. However, in the actual scheme, the adversary can only glean this through observing the delayed write-backs. However, given that the write-backs are mixed and matched and target pseudorandom locations, it is difficult for a computationally bounded adversary to trace each encrypted document it accesses during a search query at timestamp t back to the timestamp $t' < t$ when the document was originally inserted.

In the discussion below, we expand some more on delayed write-backs and their impact on the leakage of our dynamic scheme using an example.

File-Injection Attacks. Recall that file-injection attacks [110] are an extremely powerful class of query-recovery attacks where the adversary has the ability to additionally inject maliciously crafted files into the database. The adversary uses the occurrences of these files in query outputs to identify the keyword(s) underlying a given query. Once again, query recovery via file-injection relies crucially on the document access pattern leakage. In particular, it requires the adversary to identify which of the maliciously crafted files appear in the outcome of a given query (either from accesses to the search index or to the document array).

As was the case for static SWiSSSE, this leakage is also not available during search or update queries in dynamic SWiSSSE as the document identifiers matching a given query are never revealed in the clear, and the locations of documents in the encrypted document array change with every write-back operation (making it hard for the adversary to trace the occurrence of malicious documents across queries). Below, we explain in greater detail why the strong notion of backward privacy achieved by our scheme makes it more resilient to file-injection attacks than existing backward private SSE schemes. We illustrate this via a simple example.

Consider the following query-recovery attack strategy on a dynamic scheme: the adversary injects n maliciously created documents $\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_n$ at timestamps $t_1 < t_2 < \ldots < t_n$, and at a later point in time $t^* > t_n$ passively observes the outcome of a search operation involving an unknown keyword w. Also assume that it can identify (from some leakage source) which of the documents it injected earlier appeared in the search outcome. It can then exploit its knowledge of the keyword distributions across these documents to try to recover w.

To our knowledge, all existing dynamic SSE schemes are vulnerable to this attack despite their forward and backward security guarantees. In particular, a scheme that only satisfies Type-1 backward privacy is vulnerable to this attack, since the result pattern leakage trivially allows the server to learn which of the maliciously injected files appear in the search outcome. Perhaps surprisingly, dynamic SWiSSSE is able to thwart an attack as strong as this. In particular, by the arguments given above, our scheme makes it difficult for the adversary to map the outcome of the search operation

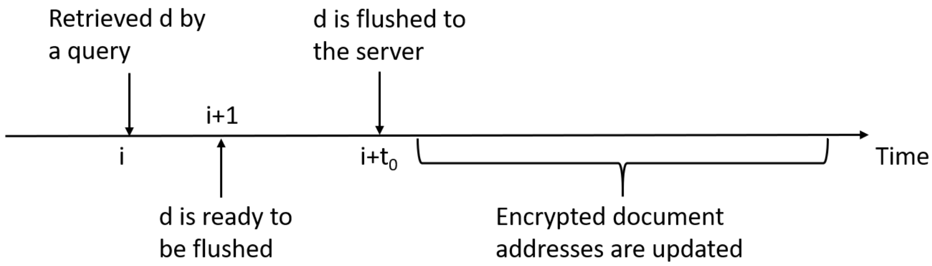


Figure 5: An illustration of the operations related to a document d in our construction. At time i, the document d is retrieved by a query. The document is in the stash and ready to be written back at time $i+1$. The document itself is written back at time $i+t_0$, where t_0 is a random delay due to randomised write-backs. The encrypted document addresses associated to d will be updated randomly in time later than $i+t_0$.

on w at time t^* to any of the malicious document injections at the prior timestamps $t_1, t_2, \ldots, t_n$.

Remark E.1. We *do not* claim that the above file-injection attack compromises *existing* notions/definitions of backward privacy. The above example simply illustrates that standard notions of backward privacy are not sufficient to rule out some very strong instances of file-injection attacks. This motivates considering even stronger notions of backward privacy that would also rule out such powerful attacks. Such a stronger notion of backward privacy (implied by obliviousness of operations) is achieved by dynamic SWiSSSE.

Encrypted document write-back. Without loss of generality, let DB be the database and $(q_1, \ldots, q_k)$ be the sequence of queries on the database. Let d be one of the documents retrieved in query q_i where $1 \le i < k$. We are interested in the distribution of t_0 for which query q_{i+t_0} triggers the write-back of document d. As half of the documents are written back from the stash after each query, t_0 clearly follows a shifted geometric distribution with parameter $\frac{1}{2}$, unless that there is a query q_{i+j} that retrieves d. In the latter case, the write-back of document d will happen at query q_{i+j+t_0} with t_0 following a shifted geometric distribution with parameter $\frac{1}{2}$.

Encrypted document address write-back. Under the same setting as above, let d be one of the documents retrieved in query q_i where $1 \le i < k$ and w be one of the keywords of d. We are interested in the distribution of t_1 for which query q_{i+t_1} triggers the write-back of the encrypted document address associated to keyword w. Recall that the write-backs for the encrypted document addresses are scheduled after the respective documents are written back to the server, and half of the encrypted document addresses are written back to the server in each query, this means that t_1 follows the sum of a shifted geometric distribution and a geometric distribution both parameterized by $\frac{1}{2}$, or equivalently, one plus a negative binomial distribution with parameter $(2, \frac{1}{2})$. As before, if the document d is retrieved by another query q_{i+j} before it is written back, then we will write the encrypted document address in query q_{i+j+t_1}.

Resistance to Cryptanalysis. Finally, for appropriate parameter choices (e.g., bucket sizes and bucketization strategies), dynamic SWiSSSE achieves strong enough backward privacy guarantees in practice to resist a wide range of cryptanalytic attacks based on system-wide leakage, such as access pattern and query equality pattern based attacks [21, 59], file injection attacks [110], and attacks based on highly refined leakage (such as the correlation-leakage based attack described in Section C). Also noteworthy is the fact that SWiSSSE achieves such strong guarantees without compromising significantly on query performance and communication overheads. This makes it an attractive candidate for deployment in typical applications involving outsourced databases.

E.5 Dynamic SWiSSSE: Asymptotic Performance Evaluation

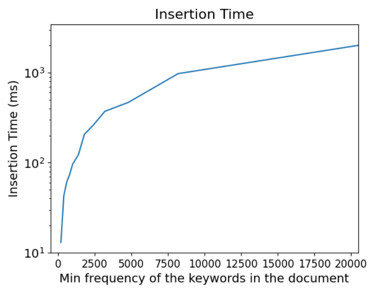
In this section, we revisit the asymptotic performance analysis of SWiSSSE from Appendix D, with focus on the dynamic version.

Size of the Stash. Since search queries are processed exactly as in static SWiSSSE, the corresponding stash size required remains unchanged, i.e. the space complexity is $O(\max_w \mathsf{G}(w))$. For document insertion queries, the documents to be inserted are processed with the responses, so the same analysis on the space complexity applies. Similarly, the size of the local lookup index also remains unchanged, and the client still needs to store $O(\max_w |W\{\mathsf{DB}(w)\}|)$ lookup index locations.

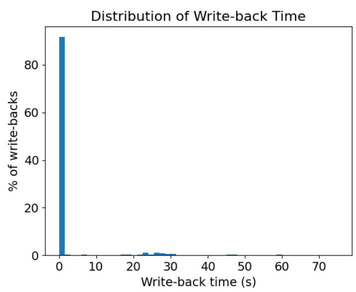
The main change from the static version is that, in dynamic SWiSSSE, the client needs to store *three* arrays of integers, namely an array for the groupings of the keywords, an array for the number of insertions of the keywords, and an array for the counters used to generate the document array addresses. However, these arrays are all small and of constant size, so they do not contribute to the asymptotic size of the stash. Combining everything together, we get that the size of the stash is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$, which is the same as static SWiSSSE.

Size of the Encrypted Database. As in static SWiSSSE, the server stores an encrypted lookup index and an encrypted document array, with combined size $O(\sum_w \mathsf{G}(w) + |\mathsf{DB}|)$. Note that this order-of-magnitude calculation ignores the overhead from padding all documents to a constant size.

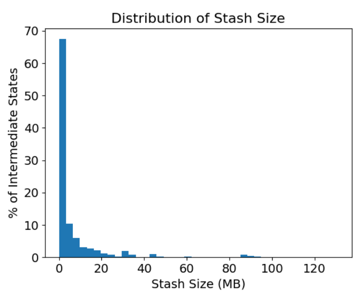
Time Complexity and Communication Volume of a Query. Asymptotically, the time complexity and communication volume of search and insertion/deletion queries in dynamic SWiSSSE are the same as that of a search query in static SWiSSSE. However, certain concrete constants differ due to the additional write-back addresses involved for handling updates, and we detail this for the sake of



(a) Insertion of dynamic SWiSSSE **on 400K documents (log scale).**



(b) Distribution of write-back time of dynamic SWiSSSE **on 400K documents.**



(c) Stash size of dynamic SWiSSSE **on 400K documents.**

Figure 6: Performance comparison between the plaintext database and dynamic SWiSSSE**.**

completeness. Suppose that the leading keyword for the query is w. In our construction, a query consists of three rounds of interaction. In the first round, the client computes the encrypted lookup index addresses for the query. This involves $O(\mathsf{G}(w))$ computation and communication, as there are at most $3 \cdot \mathsf{G}(w)$ addresses involved. The server then takes $O(\mathsf{G}(w))$ time to retrieve the encrypted document array addresses and send them to the client. This means that the overall communication volume is $O(\mathsf{G}(w))$ for the first round.

Upon receiving the $O(\mathsf{G}(w))$ encrypted document array addresses, the client processes them and retrieves $2 \cdot \mathsf{G}(w)$ encrypted documents from the server. The client decrypts the documents and filters the results locally to obtain the query response. The time complexity for the overall process is $O(\mathsf{G}(w))$. It is also straightforward to see that the communication volume and the time complexity for the server are both $O(\mathsf{G}(w))$.

Finally, after receiving the encrypted documents from the previous step, the client decrypts them in $O(\mathsf{G}(w))$ time. If the query is a document insertion query, the client has to do at most $O(\mathsf{G}(w))$ amount of work to turn one of the fake documents into the document intended for insertion. After that, the client randomly picks at most $2 \cdot \max_w \mathsf{G}(w)$ documents from the stash, encrypts them and uploads them to the server. He also randomly picks half of the lookup indices stored in the stash, encrypts them, and uploads them to the server. Using the analysis of the stash size above, we conclude that the time complexity and communication volume for this step is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$. This means the overall time complexity of this step for the client and the communication volume is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$. Similarly, we conclude that the time complexity of this step for the server is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$.

Combining the analyses above together, we conclude that the time complexity of a query for both the client and the server is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$, while the communication volume of a query is $O(\max_w \mathsf{G}(w) + \max_w |W\{\mathsf{DB}(w)\}|)$. We note that stash handling is not relevant to the retrieval of documents and it can be performed whenever the client is free. With regards to document retrieval only, the time complexity for the client and the server is $O(\mathsf{G}(w))$ and the communication volume is $O(\mathsf{G}(w))$.

E.6 Dynamic SWiSSSE: Experimental Evaluation

In this section, we provide experimental results on dynamic SWiSSSE. We use the same experimental setup as we used for static SWiSSSE in Section 6. The only difference between the two set of experiments is that we create as many placeholder documents (and keywords) as the real ones for insertions. This doubles the setup time and storage cost of dynamic SWiSSSE as compared to the static version.

Insertion and Query Response Time. Figure 6a shows the insertion time for dynamic SWiSSSE. As expected, the insertion time grows linearly with respect to the minimum frequency of the keywords in the document to be inserted. The time for an insertion query and the time for a search query are the same by design for dynamic SWiSSSE.

Write-back Efficiency. We report write-back efficiency for the experiment with 400K documents in Figure 6b. As before, the majority of the write-back queries are completed in under one second.

The Client Stash. The distribution of the stash size is shown in Figure 6c. The stash size is under 10 MB for over 90% of the time. It gets large occasionally due to consecutive queries on high frequency keywords, but we believe that this will rarely happen in real deployments; it is also possible for the client to issue dummy queries to reduce the stash size rapidly.