SWiSSSE: System-Wide Security for Searchable Symmetric Encryption

Zichen Gui
ETH Zürich

Kenneth G. Paterson
ETH Zürich

Sikhar Patranabis
ETH Zürich

Bogdan Warinschi
University of Bristol and Dfinity

October 22, 2020

Abstract

This paper initiates a new direction of research for searchable symmetric encryption (SSE). We provide comprehensive security models and notions for SSE in the simulation tradition that encompass leakage from the whole SSE system, including accesses to encrypted indices and the encrypted database documents themselves. We provide static and dynamic SSE constructions targeting our new notions. Our constructions involve a combination of novel techniques: bucketization to hide volumes of responses to queries; delayed, pseudorandom write-backs to disrupt access patterns; and indistinguishable search and update operations. The oblivious operations make it easy to establish strong versions of forward and backward security for our dynamic SSE scheme and rule out file-injection attacks. Our implementation of the dynamic SSE scheme demonstrates that it offers very strong security against general classes of leakage-abuse attack with moderate overhead. Our schemes scale smoothly to databases containing hundreds of thousand of documents and millions of keyword-document pairs.

1 Introduction

The advent of cloud computing potentially allows individuals and organizations to outsource storage and processing of large volumes of data to third party cloud servers. However, this ability to harness external storage and processing capabilities comes with privacy concerns. Ongoing threats from malicious insiders and external attackers had led to a situation where clients no longer trust cloud service providers to respect the confidentiality of their data.

A simple and efficient solution for ensuring the privacy of client data is to encrypt it using some conventional encryption algorithm (for instance, using the Advanced Encryption Standard (AES) block cipher in some appropriate mode) before offloading it to the cloud server. This solution guarantees privacy of data at rest and in transit. At the same time,

this solution renders the server incapable of performing computations on the data without decrypting it first.

Searchable Symmetric Encryption (SSE). The goal of Searchable Symmetric Encryption (SSE) [41, 17, 12, 9, 8] 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 [12, 9, 33], and *dynamic* SSE schemes that also allow for updates of encrypted document collections [11, 27, 8, 4, 24, 6, 10].

Leakage vs Efficiency. The main challenge in designing SSE schemes is ensuring an appropriate balance between information leakage and efficiency. For instance, the most general notion of SSE with optimal security guarantees (wherein little or no information is leaked to the server) can be achieved using techniques like fully homomorphic encryption (FHE) [16] and Oblivious RAM (ORAM) [18]. 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 [12, 9, 8, 24]. Types of leakage widely seen in the SSE literature include access pattern (which locations of a data structure are accessed for a given query), equality pattern (whether two queries correspond to the same underlying keyword), volume pattern (how many documents match a given keyword query), result pattern (which documents match a given keyword query) and intersection pattern (which documents are common to the result pattern of two or more queries). Given the current state-of-the-art in SSE, such leakage seems necessary in order to obtain reasonable performance

Leakage-Abuse Attacks. Leakage from SSE schemes (both static and dynamic) can be exploited by an adversarial server to infer sensitive client information. Islam *et al.* [23] and Cash *et al.* [7] gave query recovery attacks that exploit access pattern leakages and prior knowledge about the database. Zhang *et al.* [47] proposed active "file injection attacks", that completely break the query privacy guarantees of existing SSE schemes by exploiting their access pattern and result pattern leakages. SSE schemes supporting range queries can be broken by attacks exploiting either access pattern leakages [32, 20] or volume leakages [28, 19, 21].

Leakage Mitigation Challenges. Defending SSE schemes against leakage-abuse attacks while maintaining acceptable search performance is difficult. The state-of-the-art is represented by recent work [26, 13] giving SSE schemes that provably leak little or no information to the adversarial server. Kamara and Moataz [26] achieve this via a novel data structure called a volume-hiding encrypted multi-map. This provides better search performance compared to naïve padding. Demertzis *et al.* [13] propose a technique called "adjustable leakage" which uses a combination of ORAM-like techniques with padding to selectively hide access pattern and volume leakages, while maintaining reasonable search performance. 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 allowing the client to recover the document identifiers matching a given query.

Surprisingly, however, 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 omission is understandable when the leakage from the search index already subsumes the leakage from the final processing step, as is the case in many existing SSE schemes [12, 9, 8, 6, 33]. But it can no longer be ignored for more recent schemes in [26, 13] because they do succeed in reducing the encrypted search index leakage to the point where leakage from other system components becomes dominant. Concretely, it is straightforward to transfer existing leakage-abuse attacks based on co-occurrence patterns or volumes to the document level, thereby breaking the schemes in [26, 13] when properly viewed in the system context. It is currently unclear whether known leakage mitigation techniques that work for the encrypted search index can be scaled to the overall encrypted database. In particular, existing techniques tend to rely on introducing redundancies in the search index in order to reduce leakage. Such redundancy-based approaches are likely to be prohibitively expensive when applied to the entire encrypted database.

1.1 Our Contributions

These observations motivate a change of perspective. Instead of focussing on individual components of a system using SSE, such as 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 leakage-minimal yet efficient SSE schemes supporting keyword searches over both static and dynamic databases. Crucially, we include in our models, constructions and analysis for SSE the client's interactions with both the (encrypted) index and the data storage component of a searchable encryption *system*. We expand on these contributions below.

System-wide Security Definitions. We propose new security definitions for both static and dynamic SSE schemes that take into account the system-level leakage during setup, searches, and updates. As is standard, our definitions are for an honest-but-curious server and we use the real world/ideal world simulation paradigm with leakage profiles to model security. 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 only gains information from 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.

Oblivious Operations. For dynamic SSE, we 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 security notion has several advantages. First of all, it naturally implies that search and update operations incur computationally indistinguishable leakages. This allows for a unified security definition with respect to searches and updates for dynamic SSE schemes, as opposed to separate definitions in the existing literature.

Obliviousness of operations also yields stronger notions of forward and backward privacy than in [4, 6, 10]. For instance, it yields 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. It also rules out a very strong class of file injection attacks [47] that can otherwise compromise Type-1 [6] (and all other weaker notions of) backward privacy. In fact, obliviousness of operations 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 far stronger than *Type-1* backward privacy, the strongest notion of backward privacy achieved by existing SSE constructions.

New Construction Techniques. We propose new SSE constructions, giving both static and dynamic schemes (named static SWiSSSE and dynamic SWiSSSE). Dynamic SWiSSSE supports keyword searches and updates (insertions and deletions); both operations require *two* rounds of communication between the client and the server. We prove that static and dynamic SWiSSSE are adaptively secure under our new system-wide security definitions with respect to well-defined leakage profiles. At the heart of our new constructions are two techniques, *keyword frequency bucketization* and *delayed, pseudorandom write-backs*.

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 volume leakage. Unlike worst case padding which either imposes a linear search overhead [25] 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. The volume leakage incurred by our strategy is a function of the number of buckets used and the relative distribution of keywords across them. Indeed, there is an interplay between the bucketization strategy, the padding overhead, and the security that we obtain. We explore this in detail in the main body.

To mitigate access pattern leakage, we use a delayed, pseudorandom write-back strategy. After each search or update 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 it back to the server at regular intervals. In this way, data pertaining to multiple keywords gets mixed during the write-backs. This ensures that the server cannot correlate search and update operations through write-backs. Now, for correctness, search and update 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 schemes such that the latency of online operations (searches and updates) is not affected by the latency of write-back operations, which occur independently. All operations in our static and dynamic schemes require exactly two rounds of online communication. This contrasts with the polylogarithmic round complexity of previous SSE schemes using ORAM-style techniques [6, 10].

Of course, delayed write-backs have been used in the context of other cryptographic protocols such as ORAM [18], anonymous communication and anonymous blockchain transactions.

But this is, to our knowledge, their first use for leakage mitigation in SSE.

Leakage Analysis. We formally prove the security of static and dynamic SWiSSSE with respect to our system-wide security definitions and specific leakage profiles. This leaves the question: what is the impact of that leakage? To answer that question, we perform a detailed cryptanalysis. In particular, we show that the known leakage abuse attacks fail against our schemes. We also introduce a significantly more powerful version of the IKK attack [23] and show that it fails, provided we choose an appropriate bucketization strategy. We have made the cryptanalysis of our own schemes as "unfriendly" as possible but of course welcome further analysis from the community.

Implementation. We provide a Java implementation of dynamic SWiSSSE and evaluate its performance. We used Redis [1] as the underlying database system. For comparability with previous research in SSE, we use the Enron email corpus [46] 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 50 MB and the storage used by the server is about 3 GB; the throughput of our scheme is 1.77 documents per millisecond (ms) for setup, 2.85 documents per ms for search queries, and 1.14 documents per second for an average insertion or deletion.[1]

Organization. Section 2 provides essential preliminaries and background material on SSE. Section 3 introduces the core ideas behind static SWiSSSE; Sections 4 and 5 develop these into our full proposal for static SWiSSSE and prove its security. Section 6 presents the cryptanalysis of the proven leakage of static SWiSSSE. Section 7 presents the dynamic version of SWiSSSE and presents a detailed discussion of its security properties, including its forward and backward privacy guarantees. Section 8 presents an asymptotic performance analysis of SWiSSSE in both the static and dynamic cases. Finally, Section 9 describes our prototype implementation of SWiSSSE and its evaluation. Additional details on our cryptanalysis experiments and on the database used for our performance evaluation can be found in the appendices.

2 Preliminaries and Background

In this section, we introduce notations and preliminary background material. We present definitions of basic cryptographic primitives used in the paper, as well as background material on SSE.

[1] The query response time for insertions and deletions depends on the minimum keyword frequency of the document; here we are reporting the average case.

2.1 Notation

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

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 all our constructions 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** : takes no input and initialises an empty key-value store.
- **get** : takes as input a key k and returns the value associated to the key. 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 abuse the notation and use **put** for putting a set of key-value pairs too.
- **contains** : takes as input a key k and returns a boolean which indicates if k is one of the keys in the key-value store.
- **del**: takes as input a set of keys $\mathcal{I}$ and remove the key-value pairs with the keys in $\mathcal{I}$ from the key-value store.
- **pop**: takes as input a natural number n. It selects n uniformly random key-value pairs from the store, removes them from the store and returns them as the result of the call.

We do not explicitly distinguish between the name of the data structure and its state. For example, 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 Basic Cryptographic Primitives

This section presents the definitions and security notions for various cryptographic primitives used in the paper.

Pseudorandom functions. A pseudorandom function (PRF) is a polynomial-time computable function

$$F : \{0,1\}^{\lambda} \times \{0,1\}^{\ell} \longrightarrow \{0,1\}^{\ell'},$$

such that for all PPT algorithms $\mathcal{A}$, we have

$$\left|\Pr\left[\mathcal{A}^{F(K,\cdot)}=1\right]-\Pr\left[\mathcal{A}^{G(\cdot)}=1\right]\right|\leq \mathsf{negl}(\lambda),$$

where $K \xleftarrow{R} \{0,1\}^\lambda$ and G is uniformly sampled from the set of all functions that map $\{0,1\}^\ell$ to $\{0,1\}^{\ell'}$.

Symmetric-Key Encryption. A symmetric-key encryption scheme SKE consists of the following polynomial-time algorithms:

- **KGen**(λ): A probabilistic algorithm that takes the security parameter λ as input and outputs a secret-key sk.
- **Enc**(sk, x): A probabilistic algorithm that takes as input a key sk and a plaintext x. Outputs a ciphertext c.
- **Dec**(sk, c): A deterministic algorithm that takes as input a key sk and a ciphertext c. Outputs the decrypted plaintext x.

A symmetric-key encryption scheme is said to be CPA-secure if for all PPT algorithms $\mathcal{A}$ and any two *arbitrary* plaintext messages x_0 and x_1, we have

$$|\Pr\left[\mathcal{A}\left(\mathbf{Enc}(sk, x_0)\right)=1\right]-\Pr\left[\mathcal{A}\left(\mathbf{Enc}(sk, x_1)\right)=1\right]|\leq \mathsf{negl}(\lambda),$$

where $sk \leftarrow \mathbf{KGen}(\lambda)$.

2.4 Searchable Symmetric Encryption

We recall the syntax of searchable symmetric encryption (SSE) below. We consider a setting where a client wants to outsource a database $\mathtt{DB} = \{d_i\}$ 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 $\mathtt{DB}[i]$ 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 runs from 0 to $|\mathtt{DB}|-1$. For each document d we write $W(d)$ for the set of keywords contained in the document d. We write $W\{\mathtt{DB}\}$ for the multiset of keywords in the collection of documents $\mathtt{DB}$. We write $\mathtt{DB}(w)$ to mean the set of documents which contains keyword w, so a search for keyword w should return $\mathtt{DB}(w)$. The formal treatment below covers the case of dynamic databases where documents can be added and/or removed. The case of static databases is a particular case.

Syntax. An SSE scheme Σ consists 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 say that Σ is static if only **Setup** and **KWQuery** are supported, and dynamic if all four operations are supported.

- $\mathbf{Setup}(1^\lambda, \mathtt{DB}) = (\mathtt{Clt}.\mathbf{Setup}(1^\lambda, \mathtt{DB}), \mathtt{Svr}.\mathbf{Setup}(\mathtt{EDB}))$ is a protocol between the client and the server. The client takes as input a security parameter 1^λ and a database $\mathtt{DB}$, and outputs $(\mathtt{EDB}, \mathtt{st}_{\mathtt{Clt}})$, where $\mathtt{EDB}$ is an encrypted database and $\mathtt{st}_{\mathtt{Clt}}$ 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.

- $\mathbf{KWQuery}(sk, q, \mathtt{st}_{\mathtt{Clt}}; \mathtt{EDB}) = (\mathtt{Clt}.\mathbf{KWQuery}(sk, q, \mathtt{st}_{\mathtt{Clt}}), \mathtt{Svr}.\mathbf{KWQuery}(\mathtt{EDB}))$ is a protocol between the client and server to support single-keyword query. The query q in this case looks like $(\mathbf{KWQuery}, w)$, where w is the target keyword of the query. The client takes as input a secret key sk, a search query q, and his internal state $\mathtt{st}_{\mathtt{Clt}}$. 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 $(r, \mathtt{st}_{\mathtt{Clt}})$ where r is the result of the single-keyword query and $\mathtt{st}_{\mathtt{Clt}}$ is the new state of the client; the server state $\mathtt{EDB}$ may also be modified following the execution.

- $\mathbf{Insert}(sk, q, \mathtt{st}_{\mathtt{Clt}}; \mathtt{EDB}) = (\mathtt{Clt}.\mathbf{Insert}(sk, q, \mathtt{st}_{\mathtt{Clt}}),$ $\mathtt{Svr}.\mathbf{Insert}(\mathtt{EDB}))$ is a protocol between the client and server to support update on the database. An insertion query q takes the form $(\mathbf{Insert}, \{w\}, d)$, 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 $\mathtt{st}_{\mathtt{Clt}}$. 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 $(r, \mathtt{st}_{\mathtt{Clt}})$ where r is the response from the server and $\mathtt{st}_{\mathtt{Clt}}$ is the new state of the client; the server gets $\mathtt{EDB}$ where $\mathtt{EDB}$ is the encrypted database after the insertion operation.

- $\mathbf{Delete}(sk, q, \mathtt{st}_{\mathtt{Clt}}; \mathtt{EDB}) = (\mathtt{Clt}.\mathbf{Delete}(sk, q, \mathtt{st}_{\mathtt{Clt}}),$ $\mathtt{Svr}.\mathbf{Delete}(\mathtt{EDB}))$ is a protocol between the client and server to support update on the database. An insertion query q takes the form $(\mathbf{Delete}, d)$, 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 $\mathtt{st}_{\mathtt{Clt}}$. 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 $(r, \mathtt{st}_{\mathtt{Clt}})$ where r is the response from the server and $\mathtt{st}_{\mathtt{Clt}}$ is the new state of the client; the server gets $\mathtt{EDB}$ where $\mathtt{EDB}$ is the encrypted database after the insertion operation.

Correctness and Security. We now define correctness and security for SSE schemes. We start with the latter, and then use some of the formalism to define correctness. As usual, we formulate the security of an SSE scheme in terms of a *leakage* function which captures the information an adversary can unavoidably learn. A scheme is secure with respect to a leakage function if no adversary can distinguish between the real execution of the scheme and an execution simulated given only the leakage of the real execution. It is helpful to distinguish between the different places where leakage can occur, so we formally define a leakage function as a quadruple $\mathcal{L} = (\mathcal{L}^{\mathbf{Setup}}, \mathcal{L}^{\mathbf{KWQuery}}, \mathcal{L}^{\mathbf{Insert}}, \mathcal{L}^{\mathbf{Delete}})$.

The security definition below is parametric in the type of queries the adversary is allowed: if these are only search queries then we are in the static setting whereas if they also include update queries then we are in the dynamic scenario. Furthermore, we can distinguish between non-adaptive and adaptive adversaries, depending on how they chose their queries. In both cases the adversary attempts to distinguish between a real world (where it interacts

with the real scheme) and an ideal world (where it interacts with a simulator which can only access the leakage from the real execution).

Definition 1 (Security of SSE schemes)**.** *Let* $\Sigma = (\boldsymbol{Setup}, \boldsymbol{KWQuery}, \boldsymbol{Insert}, \boldsymbol{Delete})$ *be an SSE scheme and consider the following probabilistic experiment where* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator and* $\mathcal{L}$ *be the leakage function.*

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

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

$\boldsymbol{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(1^\lambda)$:

1. *The adversary* $\mathcal{A}$ *selects a database* $\mathtt{DB}$ *and gives* $\mathcal{L}^{\boldsymbol{Setup}}(\mathtt{DB})$ *to the simulator* $\mathcal{S}$*.*
2. *Using* $\mathcal{L}_{\boldsymbol{Setup}}(\mathtt{DB})$*, the simulator* $\mathcal{S}$ *generates an encrypted database* $\mathtt{EDB}$ *and return it to the adversary* $\mathcal{A}$*.*
3. *The adversary picks a polynomial number of queries* $q_1, \ldots, q_{\mathtt{poly}(\lambda)}$*. For each query, the simulator* $\mathcal{S}$ *computes the transcript of the query using the appropriate component of the leakage function* $\mathcal{L}$*, and sends it to the client.*
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 adversary* $\mathcal{A}$*,*

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

We say that Σ is non-adaptively secure if the adversary $\mathcal{A}$ chooses the queries before executing them. We say that Σ is adaptively secure if the adversary $\mathcal{A}$ can choose his queries based on the past transcripts.

Correctness. To define correctness, we use the game $\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda)$ above, which models the execution of the protocol against an arbitrary adversary $\mathcal{A}$. We define correctness by considering the same execution but only against the class of honest-but-curious adversaries who execute the protocol honestly. We say that such an adversary $\mathcal{A}$ wins the correctness game, if at any point during the execution of a single-keyword search query q for which the data returned to the client is not the same as the data obtained by executing the queries against the unencrypted database. For an adversary $\mathcal{A}$ which makes at most k queries, we define its advantage in breaking correctness, $Adv^{corr,k}_{\Sigma,\mathcal{A}}(1^\lambda)$, as the probability of this event.

Forward and Backward Privacy. One of the drawbacks of the definition above is that security with respect the notion above only guarantees security against passive adversaries. In a recent line of work, powerful file-injection attacks have been discovered [47, 38] which are devastating if the attacker is able to insert a set of documents with keywords of his choice. On a high level, the attacks work by abusing the information leaked from the maliciously inserted documents. For example, if the adversary inserts a document with keywords w_1 and w_2 and the token of keyword w_1 can be linked to the other tokens of the same keyword, then the adversary learns the number of documents associated to keyword w_1. Even worse, if a single-keyword query is executed on keyword w_1 and the list of document identifiers associated to w_1 is leaked by the scheme, then the adversary learns the exact set of documents that contains w_1.

In the light of this powerful attack, forward [43, 15, 5, 4] and backward private SSE schemes [6] are proposed to limit the exposure of the schemes to file-injection attacks. Intuitively, a scheme is forward private if an update on the database only leaks the number of keywords inserted but not the keywords themselves. This means the first attack described above does not work against forward-private schemes. Backward privacy is a bit more involved, but on a high level, a scheme is said to satisfy backward privacy if a single-keyword search with keyword w on the database only leaks some information about keyword w itself and no the other keywords. This means the second attack described above still works, but at least information regarding keyword w_2 is not leaked at that point in that.

3 Static SWiSSSE: Warm-Up

To highlight some of the key techniques underlying SWiSSSE, we start by considering a highly simplified setting where we have a static database in which each document contains precisely one searchable 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 more

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 them), 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 stored at the server.

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 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.

Garbage Collection. Observe that each write-back operation uses up newly generated addresses (keys) in the encrypted keyword lookup index and the encrypted document array. The corresponding "old addresses" are never used in the future and can be garbage-collected by the server after handling a query. This avoids the required server-side storage growing linearly with the number of queries. Of course, at this point, the data read is already stored in the stash of the client, so there is no data loss on account of the garbage collection.

Search Queries. A search query involving a keyword w proceeds as follows:

1. 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.

2. Next, the client decrypts the entry received from the server and retrieves the list of pointers to addresses in the encrypted document array, which it sends across to the server. The server retrieves the corresponding entries from the encrypted document array and sends these back to the client.

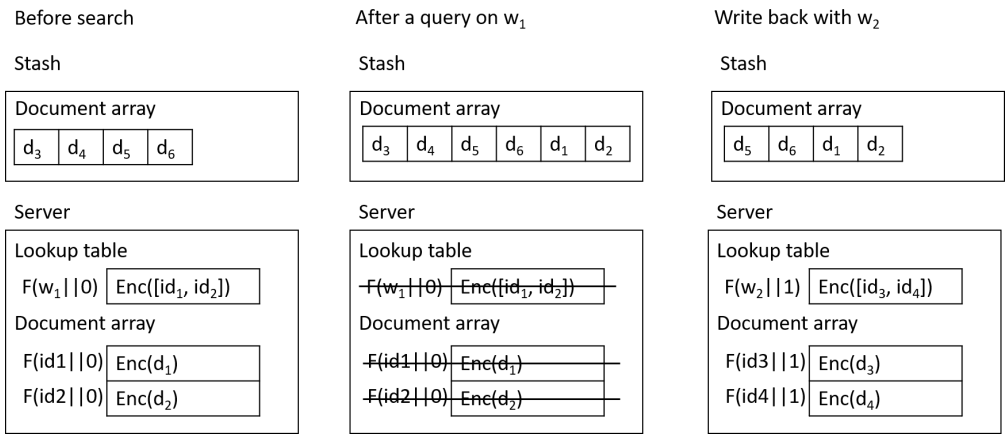

Figure 1: An illustration of the operations related to keyword search and delayed write-backs corresponding to two keywords w_1 (which occurs in documents d_1 and d_2) and w_2 (which occurs in documents d_3 and d_4). The query on w_2 is executed before the query on w_1. The figure on the left shows the status of the client stash and the encrypted data structures before the query on w_1. As soon as the query on w_1 is made, the corresponding entries in the encrypted data structures are deleted by the server, while the client stash gets updated with the same information. The figure in the middle reflects this. The figure on the right illustrates a delayed write-back operation for w_2.

3. Upon receiving the encrypted entries from the server, the client decrypts them, discards any fake documents and retains the real ones.

4. 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.

Figure 1 illustrates with an example how keyword searches and delayed write-backs affect the client stash and the state of the encrypted data structures at the server. Note that write-backs are only triggered intermittently by the client whenever its private stash is filled to capacity and needs to be flushed. Hence, the search latency is not affected by the latency of write-back operations.

System-Wide Leakages. There are two kinds of system-wide leakages potentially incurred by SWiSSSE – access-pattern leakages and write-back leakages. We provide here an informal overview of what the server can infer from these leakages. We defer the formal leakage profile enumeration and its cryptanalysis to Sections 5 and 6.

Observe that across both encrypted data structures, the server accesses any given address at most twice, once during a write-back and once during a subsequent read operation (the entry is subsequently deleted and the corresponding address is never re-used). Hence, unlike most existing SSE schemes, including those in [12, 9, 8, 24, 33], the access-pattern leakage in our simplified schemes does not allow the server to immediately infer whether the same keyword was queried twice. It also does not immediately reveal if the same document identifier appears across searches pertaining to different keyword queries.

The other potential source of leakage is the write-back operations. In order to make it hard to correlate write-back operations relating to the same search operation, we use a "randomized"

write-back strategy. More specifically, write-backs corresponding to multiple queries are randomly interspersed in time and do not follow any specific ordering: what is written back during a particular write-back operation is pseudorandomly selected from everything sitting in the stash. Thus write-backs "mix and match" across multiple queries from the client's stash. Hence, each individual write-back that occurs at time t may correspond to more than one query executed prior to time t, and an adversarial server will find it hard to correlate a write-back operation to the query to which it corresponds.

Intuitively, the ease with which the leakage can be exploited grows with the frequency of write-back operations, and hence inversely with the size of the local stash at the client. This provides yet another source of security-efficiency tradeoff that we discuss in detail subsequently.

4 Bucketization in SWiSSSE

As mentioned in Section 3, we mitigate volume leakages in SWiSSSE through a frequency bucketization strategy over the set of keywords in the document collection. At a high level, this entails dividing the list 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. A prominent example is [23], wherein Islam *et al.* exploited the correlations across frequency leakage from multiple queries to launch query recovery attacks on SSE schemes.

In this section, we detail the bucketization strategy used in our constructions, and its impact on the efficiency of our construction. Section 6, presents the results of cryptanalysis showing the effectiveness of our bucketization approaches.

Worst-case padding. Note that a trivial bucketization strategy is "worst-case padding" where every keyword is padded to have the same frequency. Although theoretically free of volume leakage, this approach is extremely inefficient and either imposes a linear search overhead [25] 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.

Fixed and Variable Bucket Sizes. We propose two strategies for bucketing keywords using buckets of fixed and variable sizes, respectively. In the first strategy, we arrange the keywords in decreasing order of frequency and divide them into equal-sized buckets (of size B say). We then pad with fake data so as to equalise the frequency of all keywords in each bucket. Hence, the adversary can guess the queried keyword with probability at most $1/B$. The tradeoff here is that a larger B (i.e. larger but fewer buckets) incurs lower leakage but more padding and so greater search overheads. This strategy yields good search efficiency in practice for document collections following a Zipf power distribution [34], wherein frequency of ranked keywords decays exponentially. The guessing probability of the adversary is independent of the frequency of the queried keyword.

In the second strategy, we place the keywords (arranged in decreasing order of frequency) in increasingly larger buckets; in particular, each bucket is double the size of the previous bucket. Note that the adversary's probability of guessing a keyword correctly decays exponentially with decreasing frequency of the queried keyword. Since in practice, most of the queried keywords typically have low frequency, this strategy achieves lower leakage in practice as compared to the first strategy. However, due to the larger bucket sizes, it is typically less efficient for queries involving low frequency keywords.

We provide in Section 6 an empirical evaluations of the security offered by these strategies against co-occurrence leakage-based cryptanalysis over real-world databases, as well as their impacts on the storage and communication overheads of SWiSSSE.

Padding Strategies. Bucketization of keywords requires padding the original database with fake data. A natural 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. Indeed this suffices for the static version of SWiSSSE.

However, we choose to additionally insert fake documents containing uniformly random keywords into the database, subject to the constraints imposed by the bucketization strategy. While this requires more storage on the server, it supports dynamic databases more elegantly, since once can perform document insertion by "transforming" the slot for a fake document into a slot for a real one in a computationally indistinguishable manner (see the dynamic version of SWiSSSE in Section 7 for details).

Parameter Choices. In SWiSSSE, we used fixed bucket sizes, and pad with fake keyword occurrences such that the minimum ratio between the padded keyword frequency and the original keyword frequency is 2. We also insert as many fake documents as real documents. With respect to our dynamic construction (described in Section 7), this effectively allows for doubling the number of documents in the database via document insertions (by incrementally transforming fake documents into real ones), and for (at most) doubling the number of documents containing a given keyword w without changing its bucket frequency. In Section 6, we experimentally show that for these choices of parameters, the success rate of a highly refined query recovery attack remains low against SWiSSSE, *even* in the presence of true co-occurrence information of the padded database as auxiliary leakage.

5 Static SWiSSSE: Detailed

In this section, we formally describe SWiSSSE for general but static databases, where each document in the database contains multiple searchable keywords. The main technical challenge compared to the simplified case described in Section 3 is that each time we update a document address (during a write-back) we must also update all pointers to that new address for the multiple keywords that appear in that document. We handle this using auxiliary write-backs.

We begin with an overview of the key changes from the simplified case, and then provide the formal description. The formal description adheres to the definitions for SSE in Section 2.4.

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.

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 $|\mathtt{DB}(w_i)| - 1$ (where $\mathtt{DB}(w_i)$ 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 $|\mathtt{DB}(w_i)|$ 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 $\widetilde{d_\ell}$.

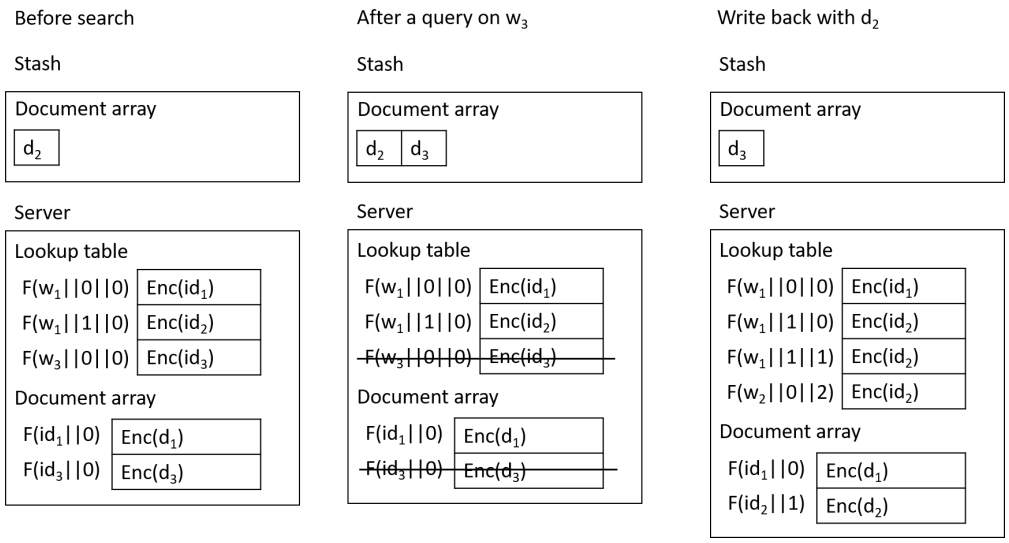

Figure 2: An illustration of the operations related to keyword search and delayed normal/auxiliary write-backs corresponding to keyword w_3 (which occurs in the document d_3) and document d_2 (which contains multiple keywords, including w_1 and w_2), respectively. The figure on the left shows the client stash and the encrypted data structures before the query on w_3. As soon as the query on w_3 is made, the corresponding entries in the encrypted data structures are deleted by the server, while the client stash gets updated with the same information. The figure in the middle reflects this. Finally, the figure on the right illustrates a delayed auxiliary write-back operation for d_2. Observe that d_2 gets written back (in encrypted form) to a pseudorandomly generated location in the document array. Additionally, as part of the auxiliary write-backs associated with d_2, the keyword-document pairs (w_1, d_2) and (w_2, d_2) get written back (in encrypted form) to pseudorandomly generated locations in the keyword lookup index.

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.

Leakage to the Server. Note that each auxiliary write-back corresponding to a keyword-document pair (w_i, d_ℓ) causes the server to *overwrite an existing entry* in the keyword lookup index. This is never the case for "normal" write-back operations, since they target freshly generated pseudorandom addresses. This leaks some information to the server. In particular, the server is able to correlate each auxiliary write-back involving a certain keyword w_i with the last normal write-back involving w_i (since both sets of write-back operations target the same addresses in the lookup index). This leaks the following information: (a) whether w_i was the subject of a previous query, and if yes, (b) that w_i must have been queried at some point in time prior to the corresponding normal write-back operation.

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:

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

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

where j is again a counter that runs from 0 to $|\mathtt{DB}(w_i)| - 1$ 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 $|\mathtt{DB}(w_i)|$ to one less than the bucket size for w_i.

Ensuring Search Correctness. Finally, 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.

Figure 2 illustrates with an example how keyword searches and delayed normal/auxiliary write-backs affect the client stash and the state of the encrypted data structures at the server in the general version of static SWiSSSE.

5.2 Formal Description of Static SWiSSSE

We now turn the aforementioned ideas into a formal description of static SWiSSSE.

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

Note that 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" $\mathtt{DB}'$ of the original database `DB` before encrypting and offloading it to the server. More concretely,

Algorithm 1 Static SWiSSSE.**Setup**

procedure $\texttt{Clt}.\textbf{Setup}(1^\lambda, \texttt{DB})$
  /* *Key generation* */
  $\texttt{Clt}.sk_1 \leftarrow \textbf{KGen}_1(1^\lambda)$
  $\texttt{Clt}.sk_2 \leftarrow \textbf{KGen}_2(1^\lambda)$
  Generate map $\texttt{G} : W \to \mathbb{N}$
  $\texttt{Clt.G} \leftarrow \texttt{G}$
  /* *Generate fake documents* */
  $\texttt{DB}' \leftarrow \textbf{Fake\_Doc\_Gen}(\texttt{DB}, \texttt{Clt.G})$
  $\texttt{Clt}.N \leftarrow |\texttt{DB}'|$
  /* $\texttt{Clt}.N$ *allows the client to locally maintain the size of the padded database. This is used subsequently in the keyword search algorithm.* */
  $\texttt{EI}, \texttt{EA} \leftarrow \{\}$
  for $i \in 1, \ldots, |\texttt{DB}'|$ **do**
    /* *Get the set of keywords with counters* */
    $x \leftarrow \{(w, \texttt{Clt.KWCtr}[w]) \mid w \in W(\texttt{DB}'[i])\}$
    /* *Update the lookup index* */
    /* *In the following expression,* $W(\texttt{DB}'[i])$ *denotes the set of keywords in the* i*-th document of the padded database.* */
    for $w \in W(\texttt{DB}'[i])$ **do**
      $j \leftarrow \texttt{Clt.KWCtr}[w]$
      $\texttt{EI} \leftarrow \texttt{EI} \cup (F(w||j||0), \textbf{Enc}(id(\texttt{DB}'[i])))$
      /* *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* */
      $\texttt{Clt.KWCtr}[w] \leftarrow \texttt{Clt.KWCtr}[w] + 1$
    /* *Insert the encrypted document* */
    $\texttt{EA} \leftarrow \texttt{EA} \cup (F(i||0), \textbf{Enc}(x||\texttt{DB}'[i]))$
  /* *Reset the keyword counter* */
  for $w \in W(\texttt{DB}')$ **do**
    $\texttt{Clt.KWCtr}[w] \leftarrow 0$
  /* *Initialise the stash* */
  $\texttt{Clt.I}.\textbf{init}()$
  $\texttt{Clt.A}.\textbf{init}()$
  Send $(\texttt{EI}, \texttt{EA})$ to the server

procedure $\texttt{Svr}.\textbf{Setup}(\texttt{EI}, \texttt{EA})$
  $\texttt{Svr.EI}.\textbf{init}()$
  $\texttt{Svr.EA}.\textbf{init}()$
  $\texttt{Svr.EI}.\textbf{put}(\texttt{EI})$
  $\texttt{Svr.EA}.\textbf{put}(\texttt{EA})$

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

Let $|\texttt{DB}(w)|$ and $|\texttt{DB}'(w)|$ denote the frequency of the keyword w in the original and padded databases, respectively. The map $\texttt{G}$ allows the client to determine a padding procedure

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

procedure Clt.**TokenGen**(w)
  $L \leftarrow \{\}$
  /* *Generate all the possible addresses for the keyword* */
  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 $\texttt{Svr.EI.get}(L)$ to the client

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

procedure Clt.**Document_Retrieval**(w, EL)
  $L \leftarrow$ the latest addresses of the keywords from $\textbf{Dec}(EL)$ if they are not in Clt.I
  Add random document identifiers between 0 and $\texttt{Clt.}N - 1$ 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* */
    $M \leftarrow M \cup F(id||\texttt{Clt.ArrCtr}[id])$
    /* *Increase the counters* */
    $\texttt{Clt.ArrCtr}[id] \leftarrow \texttt{Clt.ArrCtr}[id] + 1$
  Send M to the server

procedure Svr.**Document_Retrieval**(M)
  Send $\texttt{Svr.EA.get}(M)$ to the client

Fake_Doc_Gen; the procedure pads the input database DB with "fake" documents to obtain a padded version $\texttt{DB}'$ such that $|\texttt{DB}'(w)| = \texttt{G}(w)$ for each keyword w. The client may use any padding strategy to achieve the desired keyword frequencies as specified by G.

Additionally, to suppress volume leakages, 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 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.

SWiSSSE.**KWQuery**. We now describe the keyword query procedure for SWiSSSE. For ease of representation, 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 encrypted 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 half of 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; however this can also be a parameter input to the write-back sub-routine.

5.3 Correctness

The scheme as described does not have perfect correctness: the addresses where various information is stored are generated with a PRF and so the client may generate repeated addresses unintentionally and overwrite the encrypted document addresses or the encrypted documents. Worse still, it is possible that the queries are adversarially controlled to maximize the failure probability.

The following theorem establishes an upper bound on the advantage of any adversary against the correctness of our scheme. The probability is upper-bounded by negligible terms which account for collisions in the outputs of the PRF used in the construction, and the security of the PRF itself.

Theorem 1. [Correctness of Static SWiSSSE] Let $|\mathtt{DB}|$ and $|W\{\mathtt{DB}\}|$ denote the total number of documents and document-keyword pairs, respectively, in the database $\mathtt{DB}$, 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 SWiSSSE

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

procedure $\texttt{Clt}$.**Write_Back**$(M, \bar{w})$
  $UA \leftarrow \{\}$
  /* *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**
      $\texttt{Clt.I}.\textbf{put}((F(w||j||\texttt{Clt.KWCtr}[w] + b), \textbf{Enc}(id(d))))$
  /* *Decrypt the documents retrieved and insert them into the document array* */
  $\texttt{Clt.A}.\textbf{put}(\textbf{Dec}(M))$
  Send $(\texttt{Clt.I}.\textbf{pop}(\lfloor|\texttt{Clt.I}|/2\rfloor), UA)$

procedure $\texttt{Svr}$.**Write_Back**$((UI, UA))$
  $\texttt{Svr.EI}.\textbf{put}(UI)$
  $\texttt{Svr.EA}.\textbf{put}(UA)$

over the database $\texttt{DB}$ is at most:

$$\frac{\left(|\texttt{DB}|^2 + 4t_0|\texttt{DB}| + 4|W\{\texttt{DB}\}|^2 + 8t_1|W\{\texttt{DB}\}|\right)}{2^{l+1}} + Adv^{PRF,|\texttt{DB}|+2t_0}_{F,\mathcal{B}} + Adv^{PRF,2|W\{\texttt{DB}\}|+2t_1}_{F,\mathcal{C}},$$

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

Proof. 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 $|\texttt{DB}| + 2k \cdot \max_w |\texttt{DB}(w)|$, so the difference in the advantages between game G and G_2 is $Adv^{PRF,|\texttt{DB}|+2k\cdot\max_w|\texttt{DB}(w)|}_F$.

In our static construction, for database $\texttt{DB}$, we use $|\texttt{DB}|$ addresses to store the encrypted documents during initialisation, which means the probability of a pair of collisions is upper-bounded by $|\texttt{DB}|^2 \cdot 2^{-(l+1)}$.

In the subsequent write-backs, the number of active addresses in the encrypted document array is at most $|\texttt{DB}|$ and the number of new addresses we generate is upper bounded by $\max_w |\texttt{DB}(w)|$. This means that there are at most $|\texttt{DB}| \cdot \max_w |\texttt{DB}(w)|$ 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 $|\texttt{DB}| \cdot \max_w |\texttt{DB}(w)| \cdot 2^{-l}$.

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 $2|W\{\mathtt{DB}\}| + 2\max_w |\mathtt{DB}\{w\}|$, so the difference in the advantages between game G_2 and G_3 is $Adv_F^{PRF, 2W\{\mathtt{DB}\}+2\max_w |\mathtt{DB}\{w\}|}$.

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 $4|W\{\mathtt{DB}\}|^2 \cdot 2^{-(l+1)}$, and the probability of a pair of collision for the encrypted lookup table for each query is at most $4|W\{\mathtt{DB}\}| \cdot \max_w |\mathtt{DB}\{w\}| \cdot 2^l$.

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

$$\Big(|\mathtt{DB}|^2 + 4t_0|\mathtt{DB}| + 4|W\{\mathtt{DB}\}|^2 + 8t_1|W\{\mathtt{DB}\}|\Big) \cdot 2^{-(l+1)} + Adv_F^{PRF, |\mathtt{DB}|+2t_0} + Adv_F^{PRF, 2W\{\mathtt{DB}\}+2t_1},$$

where $t_0 = k \cdot \max_w |\mathtt{DB}(w)|$ and $t_1 = k \cdot \max_w |w\{\mathtt{DB}(w)\}|$. □

5.4 Formal Leakage Description

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 [12] and [9]), 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). We say that the enumeration is provably sound if no PPT adversary can distinguish between these two worlds with non-negligible advantage over a random guess.

We formally describe the leakage of static SWiSSSE. 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 leakages from such random accesses.

Leakages at 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 $\mathtt{DB}$ other than the number of documents in the padded database $\mathtt{DB}'$ (including both real and fake documents), and the total number of keyword-document pairs post-bucketization. Formally, we have:

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

where Σ denotes a concrete instance of static SWiSSSE.

Note that the leakage function is stateful; it maintains in $\mathbf{St}_{\mathcal{L}}$ a realisation of the padded database which is used later by the leakage function for the search queries.

Leakages during Searches and Write-Backs. At a high level, each search query reveals 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 these leakages using a probabilistic and stateful leakage function, described formally in Algorithm 5, again for an instance Σ of the static 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 formally 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 information leakaged during searches in SWiSSSE.

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

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

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

The following theorem formalizes the security of SWiSSSE.

Theorem 2 (Security of Static SWiSSSE)**.** *Let $\mathcal{L}_{\Sigma}^{\mathbf{Setup}}$ and $\mathcal{L}_{\Sigma}^{\mathbf{KWQuery}}$ be the leakage functions defined above. Then the instance Σ of the static* SWiSSSE *scheme is $\left(\mathcal{L}_{\Sigma}^{\mathbf{Setup}}, \mathcal{L}_{\Sigma}^{\mathbf{KWQuery}}\right)$-secure.*

Proof. 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 $Adv_F^{PRF,t}(\lambda)$

Algorithm 5 Static 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* */
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 2 * \mathtt{KWCtr}[\bar{w_1}]), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 2 * \mathtt{KWCtr}[\bar{w_1}] + 1), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathtt{KWCtr}[w] \leftarrow \mathtt{KWCtr}[w] + 1$
  /* *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 $\mathbf{Index}(UA, \mathtt{KWCtr})$
  $\mathtt{I} \leftarrow \{\}$
  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 $\mathbf{Retrieve}(\mathtt{A}, w)$
  $\mathtt{A}' \leftarrow \{\}$
  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 $\mathtt{A}'$

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.

Algorithm 6 Game G_1 (static construction). Only the setup step is changed.

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

Algorithm 7 Game G_2 (static construction).

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

  /* *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

(**Game 0.**) Let the real execution of the scheme on the database $\texttt{DB}$ with queries $q_1, \ldots, q_k$ be game G_0. Then we have that for any adversary $\mathcal{A}$, $\Pr\left[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^{\lambda}) = 1\right] = \Pr[G_0 = 1]$.

(**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 $\mathbf{RF}$ 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 $t_0 = 2\sum_w \mathtt{G}(w) + |\mathtt{DB}|$, and 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 $t_1 = 2\sum_i \mathtt{G}(W(q_i)) + 2\sum_i |\mathtt{DB}(W(q_i))|$, 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 $Adv_F^{PRF,t_0+t_1} + (t_0+t_1) \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$. □

6 Cryptanalysis of the Leakage

Having established the precise leakage of static SWiSSSE, we now turn to its cryptanalysis. For a corresponding analysis of dynamic SWiSSSE, see Section 7. We begin by briefly arguing that SWiSSSE is resilient to most well-known cryptanalytic techniques in the literature, including the query recovery attacks proposed in [23, 7], the document recovery attacks proposed in [7] and the file-injection attacks proposed in [47]. We then present a highly refined cryptanalytic technique based on system-level leakages with a much stronger attack model than assumed by these existing attacks, and demonstrate that for appropriately chosen bucketization parameters, SWiSSSE manages to resist even this attack.

Query and Document Recovery Attacks. Existing query recovery attacks (such as those proposed in [23] and [7]) and document recovery attacks (such as those proposed in [7]) typically rely on a combination of three kinds of deterministic leakage – volume/frequency leakage (i.e., the adversary deterministically recovers the number of documents matching a given query), document access pattern leakage (i.e., the adversary deterministically recovers which (encrypted) document identifiers are accessed across two or more queries), and query equality leakage (i.e., the adversary recovers whether two or more queries correspond to the same underlying keyword).

Through the use of keyword bucketization, SWiSSSE makes it possible to suppress volume leakages 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 doc-

ument 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.

File-Injection Attacks. File-injection attacks [47] are an extremely powerful class of query recovery attacks in which the adversary has the power to 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 encrypted document array).

In SWiSSSE, this leakage is not available during search queries 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). Hence, file-injection attacks cannot be applied directly to cryptanalyze the leakage profile for SWiSSSE.

Stronger Attacks from Refined Leakages. 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 [23]. Although it is unlikely that such leakages would be available from a real-world implementation of SWiSSSE we use this strong 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 $M : W \times W \to \mathbb{N}$ to be a two-dimensional "co-occurrence" matrix that maps pairs of keywords to the number of documents containing them both. Formally, we have

$$M[w_i, w_j] = |\mathtt{DB}(w_i) \cap \mathtt{DB}(w_j)|.$$

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 infer this information from just the leakage profile of SWiSSSE.

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 $\bar{M}[w_i, w_j]$ as a value sampled according to a binomial distribution as follows:

$$\bar{M}[w_i, w_j] \leftarrow \mathsf{Binom}(M'[w_i, w_j], 1/q),$$

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 description in Section 5.2). We assume that the attacker has access to a randomly permuted version of $\bar{M}$, which we denote as $\widetilde{\bar{M}}$.

Again, this is a very strong assumption, because inferring the matrix $\widetilde{\bar{M}}$ from the leakage profile of SWiSSSE is highly non-trivial. As already discussed, 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. Even in an ideal scenario where the client flushes out its stash after every q search queries, the attacker can guess any given entry $\bar{M}$ with probability at most $1/q$. In an actual implementation, the client only partially flushes its stash each time and writes-back documents from multiple search queries in a single batch, making such inferences even harder.

Attack Strategy. The goal of the attacker is to identify the most likely assignment of keywords to rows in $\widetilde{\bar{M}}$. Note that the randomized nature of write-backs in SWiSSSE rules out the deterministic approaches in previous works, such as count-based approaches [7] and matrix/graph matching-based approaches [23, 39]. We opt for a simulated annealing-based approach [2] to search for the most likely keyword assignment. See Section **??** for the details of this approach.

Cryptanalysis Results. We run our simulated-annealing based attack against the Enron email corpus [46]. Incidentally, this was also the target of cryptanalysis by the authors of [23]. We generated a sample co-occurrence matrix with representative keywords from different frequency ranges. Specifically, we arranged the keywords in decreasing order of frequency and (randomly) chose 800 of the most frequent keywords and 800 of the keywords with the 75-th, 50-th and 25-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 as reported in Figure 3.

We observed that it was easier to recover the keywords with higher frequencies and smaller bucket sizes. The keywords with the highest frequency could be recovered with nearly 100% accuracy when a bucket size of 50 was used, but the recovery rate degraded steadily to around 20% once the bucket size increased to 400. Interestingly, our results indicate that a larger bucket size should be used for the keywords in the 50-th percentile, which we believe is specific to the Enron database. We re-iterate that these recovery rates hold under an extremely strong attacker model; the recovery rate for SWiSSSE is expected to be significantly lower.

Implications for Bucketization Strategy. It is natural ask what these cryptanalysis results tell us about the bucketization strategies discussed earlier in Section 4. There we considered bucketing keywords using buckets of either fixed or variable sizes. For either bucketization strategy, our cryptanalysis experiments seem to indicate that we should choose an initial bucket size that is "safe" for the most frequent keywords (e.g., 400 with the Enron database). In the fixed bucketization strategy, we would use this bucket size for all keywords, which is also "safe" as our experiments revealed that for the same bucket size, keywords with lower frequency are generally less vulnerable to the recovery attack. In the variable bucketization strategy, we gradually double the bucket size, which is obviously more

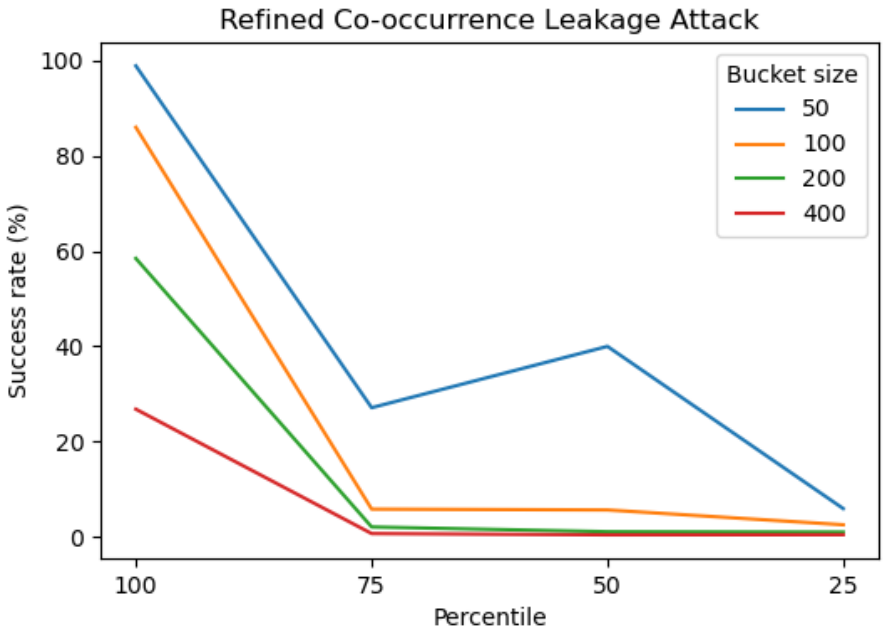

Figure 3: The average recovery rate over 100 executions of our simulated-annealing based cryptanalytic attack for keywords in various frequency percentiles. The recovery rate is measured as the fraction of keywords guessed correctly across all buckets in a single execution of the attack. We use the fixed bucket size strategy.

secure but incurs potentially greater storage requirements and communication bandwidth requirements.

Security Versus Efficiency Tradeoffs. Our experiments also reveal some interesting insights into the security versus efficiency tradeoffs associated with choosing the initial 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 20%. But what is the implication of using a larger bucket-size on the storage requirements and communication bandwidth requirements for SWiSSSE?

In Figure 4, we demonstrate through concrete figures how variations in bucket sizes affect the storage and communication overheads of our construction. The results are presented for both the fixed and variable bucket-size based strategies discussed in Section 4. In general, the total number of (real and fake) keyword-document pairs grows essentially linearly with the bucket size. This in turn implies that the storage overheads also grow linearly with the bucket size. The second bucketization strategy, where the bucket size increases as the keyword frequency decreases, naturally incurs greater overheads as compared to the fixed-size bucketization strategy.

Interestingly, for increasing bucket sizes, the growth in overhead is more gradual when

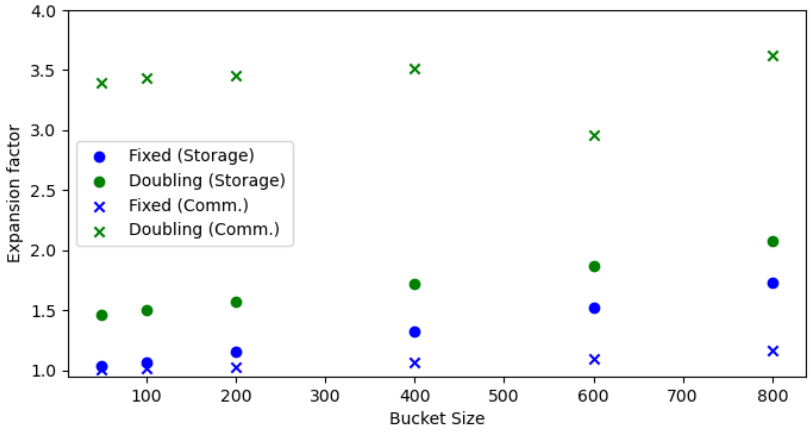

Figure 4: Overheads incurred by different bucketization strategies and bucket sizes. For the doubling bucket strategy, "bucket size" refers to the size of the first (smallest) bucket.

compared to the fall in recovery rate. For example, in the fixed-size bucketization strategy, when the initial bucket size varies from 50 to 800, the storage overhead varies between $1\times$ and $1.75\times$. On the other hand, as demonstrated earlier, the keyword recovery rate falls from 100% to below 20%. A similar observation holds for the variable-size bucketization strategy as well. This indicates that it is preferable to opt for a larger initial bucket size, since it provides significantly stronger resistance to cryptanalysis while incurring only moderately larger overheads.

Setting Parameters in Practice.Our cryptanalysis experiments seem to suggest that for a given database, it is preferable to carefully optimise the bucketization strategy and the bucket size parameters for SWiSSSE to attain a desirable tradeoff between security and efficiency. In practice, this is not easily deployable. Recall however that we 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. Hence, we suggest that a pragmatic choice of bucket size 400 should be adequate for typical applications.

At the same time, we acknowledge the need for further cryptanalysis of the leakage profile of SWiSSSE and welcome such studies from the community.

7 Dynamic SWiSSSE

7.1 Overview of Our Approach

In this section, we present an overview of how we extend SWiSSSE to handle dynamic databases. We refer the reader to Section 7.2 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 with key K, j is a counter that runs from 0 to $|\mathtt{DB}(w_i)| - 1$ (where $\mathtt{DB}(w_i)$ 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 – $\{\mathsf{addr}_{\text{norm}}(w_i, d_\ell, j)\}$ for normal write-backs, $\{\mathsf{addr}_{\text{aux}}(w_i, d_\ell, j)\}$ for auxiliary write-backs, and $\{\mathsf{addr}_{\text{insert}}(w_i, d_\ell, j)\}$ 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 [47]. 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 insert operations, 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 Section 7.2 for a more detailed discussion on how to generalize the above proposal to support an uncapped number of insertions.

7.2 Detailed Description

We now translate the informal presentation of our approach into 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

Algorithm 8 Dynamic SWiSSSE.**Setup**

```
1: procedure Clt.Setup(DB)
2:     /* Generate fake documents */
3:     DB′ ← Fake_Doc_Gen(DB, Clt.G)
4:     Clt.N ← |DB′|
5:     EI, EA ← {}
6:     for i = 1, ..., |DB′| do
7:         /* Get the set of keywords with counters */
8:         x ← {(w, Clt.KWCtr[w]) | w ∈ W(DB′[i])}
9:         /* Update the lookup index */
10:        for w ∈ W(DB′[i]) do
11:            EI ← EI ∪ (F(w||Clt.KWCtr[w]||0), Enc(id(DB′[i])))
12:            Clt.KWCtr[w] ← Clt.KWCtr[w] + 1
13:        /* Insert the encrypted document */
14:        EA ← EA ∪ (F(i||0), Enc(x||DB′[i]))
15:    /* Reset the keyword counter */
16:    for w ∈ W(DB′) do
17:        Clt.KWCtr[w] ← 0
18:    /* Initialise the stash */
19:    Clt.I.init()
20:    Clt.A.init()
21:    Send (EI, EA) to the server

22: procedure Svr.Setup(EI, EA)
23:    Svr.EI.init()
24:    Svr.EA.init()
25:    Svr.EI.put(EI)
26:    Svr.EA.put(EA)
```

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.

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

procedure Clt.**TokenGen**(w)
  $L \leftarrow \{\}$
  for $j \in 0, \ldots,$ Clt.G(w) $- 1$ **do**
    $L \leftarrow L \cup \{F(w||j||3 * \texttt{Clt.KWCtr}[w])\}$,
    $L \leftarrow L \cup \{F(w||j||3 * \texttt{KWCtr}[w] + 1)\}$,
    $L \leftarrow L \cup \{F(w||j||3 * \texttt{Clt.KWCtr}[w]) + 2\}$.
  /* *Roll forward the counter for the next query* */
  Clt.KWCtr$[w] \leftarrow$ 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**($M, \bar{w}$)
  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* */
    $UA \leftarrow UA \cup \{(F(id(d)||\texttt{Clt.ArrCtr}[id(d)]),$ **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**$((F(w||j||\texttt{Clt.KWCtr}[w] + b),$ **Enc**$(id(d))))$
  /* *Decrypt the documents retrieved and insert them into the document array* */
  Clt.A.**put**(**Dec**(M))
  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)

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 in Section 5.2 for the details of how this sub-routine works).

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**

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 */*
        $UA \leftarrow UA \cup \{(F(id(d)||\texttt{Clt.ArrCtr}[id(d)]), \mathbf{Enc}(\{(w_i, j_i)\}||d))\}$
        / Update the stash for the lookup index */*
        for $(w, j, b) \in \{(w_i, j_i, b_i)\}$ **do**
            Clt.I.**put**$((F(w||j||\texttt{Clt.KWCtr}[w] + b), \mathbf{Enc}(id(d))))$
    / 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)

is described in Algorithm 10. Technically, it is very similar to that under static SWiSSSE.**KWQuery** (Algorithm 4, Section 5.2), 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.

More explicitly, 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 $F(w||j||3 * \texttt{KWCtr}[w] + 2)$. 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 essentially identi-

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

procedure $\texttt{Clt}.\textbf{Write\_Back\_Deletion}(M, \bar{d})$
    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 */*
    $D \leftarrow \texttt{Clt.A}.\textbf{pop}(|\texttt{Clt.A}|)$
    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**
            $\texttt{Clt.I}.\textbf{put}((F(w||j||\texttt{Clt.KWCtr}[w] + b), \textbf{Enc}(id(d))))$
    / Decrypt the documents retrieved and insert them into the document array */*
    $\texttt{Clt.A}.\textbf{put}(\textbf{Dec}(M))$
    Turn $\bar{d}$ into a fake document in $\texttt{Clt.A}$
    Send $(\texttt{Clt.I}.\textbf{pop}(\lfloor|\texttt{Clt.I}|/2\rfloor), UA)$

procedure $\texttt{Svr}.\textbf{Write\_Back}((UI, UA))$
    $\texttt{Svr.EI}.\textbf{put}(UI)$
    $\texttt{Svr.EA}.\textbf{put}(UA)$

cal 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 leakages. 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 leakages thereof.

7.3 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 3. [Correctness of Dynamic SWiSSSE]

Let $|\mathtt{DB}|$ and $|W\{\mathtt{DB}\}|$ denote the total number of documents and document-keyword pairs, respectively, in the database $\mathtt{DB}$ 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 SWiSSSE over the database $\mathtt{DB}$ is at most:

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

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

7.4 Security

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 $\mathtt{DB}$ other than the number of documents in the padded database $\mathtt{DB}'$ (including both real and fake documents), and the total number of keyword-document pairs post-bucketization. Formally, we have:

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

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 these leakages 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

Algorithm 13 Dynamic SWiSSSE: Leakage Function for Keyword Queries

procedure $\mathcal{L}^{\mathbf{KWQuery}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\mathbf{KWQuery}, \bar{w}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}]), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 1), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathtt{KWCtr}[\bar{w_1}] \leftarrow \mathtt{KWCtr}[\bar{w}] + 1$
  $L \leftarrow \mathtt{I}[\bar{w}]$
  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**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

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

these leakages using a probabilistic and stateful leakage function, described formally in Algorithm 14.

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 these leakages 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 4 (Security of Dynamic SWiSSSE)**.** *Let Σ be our proposed dynamic SSE scheme. Let $\mathcal{L}_{\Sigma}^{\boldsymbol{Setup}}$ and $\mathcal{L}_{\Sigma}^{\boldsymbol{KWQuery}}$, $\mathcal{L}^{\boldsymbol{Insert}}$, and $\mathcal{L}^{\boldsymbol{Delete}}$ be the leakage functions defined above, then Σ is $(\mathcal{L}_{\Sigma}^{\boldsymbol{Setup}}, \mathcal{L}_{\Sigma}^{\boldsymbol{KWQuery}}, \mathcal{L}^{\boldsymbol{Insert}}, \mathcal{L}^{\boldsymbol{Delete}})$-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 $\mathtt{DB}$ with queries $q_1, \ldots, q_k$ be game G_0. Then we have that for any adversary $\mathcal{A}$, $\Pr\left[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^{\lambda}) = 1\right] = \Pr[G_0 = 1]$.

(**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 $t_0 = 2\sum_w \mathtt{G}(w) + |\mathtt{DB}|$.

Algorithm 14 Dynamic SWiSSSE: Leakage Function for Insertion Queries

procedure $\mathcal{L}^{\mathbf{Insert}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\mathbf{Insert}, \{\bar{w_i}\}, \bar{d}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}]), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 1), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathtt{KWCtr}[\bar{w_1}] \leftarrow \mathtt{KWCtr}[\bar{w_1}] + 1$
  $L \leftarrow \mathtt{I}[w]$
  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**
      $L \leftarrow L \cup id$
  $\mathtt{ArrCtr}[L] \leftarrow \mathtt{ArrCtr}[L] + 1$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(l, \mathtt{ArrCtr}[l]), 0, k) \mid l \in L\}$

  $UI \leftarrow \mathtt{I}'.\mathbf{pop}(|\mathtt{I}'|/2)$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(i, 1, k) \mid i \in UI\}$
  $UA \leftarrow \mathtt{A}'.\mathbf{Pop}(\lfloor |UA|/2 \rfloor)$
  $\mathbf{ArrHist} \leftarrow \mathbf{ArrHist} \cup \{(\mathbf{T}(id(d), \mathtt{ArrCtr}[id(d)]), 1, k) \mid (\{w_i, j_i, b_i\}, d) \in UA\}$
  $\mathtt{I}' \leftarrow \mathtt{I}' \cup \mathbf{Index}(UA, \mathtt{KWCtr})$
  $M \leftarrow \mathbf{Insert}(\mathtt{A}[L], \{\bar{w_i}\}, d))$
  $\mathtt{A}' \leftarrow \mathtt{A}' \cup \mathbf{Merge\_Index}(M, \bar{w_1})$
  $\mathbf{St}_{\mathcal{L}} \leftarrow (\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr})$
  Return $(\mathbf{IndHist}, \mathbf{ArrHist}), \mathbf{St}_{\mathcal{L}}$

(**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 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 $Adv_F^{PRF,t_0+t_1} + (t_0 + t_1) \cdot Adv_{\Sigma'}^{IND-CPA}(\lambda)$. □

7.5 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 updates and keyword searches are computationally indistinguishable to an adversarial server.

Algorithm 15 Dynamic SWiSSSE: Leakage Function for Deletion Queries

procedure $\mathcal{L}^{\mathbf{Delete}}(q, \mathbf{St}_{\mathcal{L}})$
  $(\mathbf{Delete}, \{\bar{w_i}\}, \bar{d}) \leftarrow q$
  $\mathtt{I}', \mathtt{A}', \mathtt{KWCtr}, \mathtt{ArrCtr} \leftarrow \mathbf{St}_{\mathcal{L}}$

  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}]), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 1), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathbf{IndHist} \leftarrow \mathbf{IndHist} \cup \{(\mathbf{T}(\bar{w}, i, 3 * \mathtt{KWCtr}[\bar{w_1}] + 2), 0, k) \mid i \in 0, \ldots, \mathtt{G}(\bar{w_1}) - 1\}$
  $\mathtt{KWCtr}[\bar{w_1}] \leftarrow \mathtt{KWCtr}[\bar{w_1}] + 1$
  $L \leftarrow \mathtt{I}[w]$
  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 $\mathtt{Clt}.\mathbf{Setup}(\mathtt{DB})$
  $(N, p, \mathbf{St}_{\mathcal{L}}) \leftarrow \mathcal{L}_{\Sigma}^{\mathbf{Setup}}(\mathtt{DB}, \mathtt{G})$
  $\mathtt{EI}, \mathtt{EA} \leftarrow [\,]$
  /* *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

The formal definition is presented below.

Definition 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 $\ell_0, \boldsymbol{St}^0_{\mathcal{L}} \leftarrow \mathcal{L}_{\Sigma}^{\boldsymbol{Setup}}(\mathtt{DB})$ and $\ell_i, \boldsymbol{St}^i_{\mathcal{L}} \leftarrow \mathcal{L}^*_{\Sigma}(q_i, \boldsymbol{St}^{i-1}_{\mathcal{L}})$ for $0 < i \leq k$ where $\mathcal{L}^*_{\Sigma}$ is the appropriate leakage function for the query q_i, and $\ell'_k, \boldsymbol{St}'^k_{\mathcal{L}} \leftarrow \mathcal{L}^*_{\Sigma}(q'_k, \boldsymbol{St}^{k-1}_{\mathcal{L}})$.*

Algorithm 17 Game G_2 (dynamic construction).

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

  /* *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 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 5 (Oblivious Operations)**.** *The dynamic variant* SWiSSSE *described above supports oblivious operations.*

7.6 Forward and Backward Privacy of Dynamic SWiSSSE

In this subsection, we describe the notions of forward and backward privacy achieved by dynamic SWiSSSE, and compare these with the notions of forward and backward privacy achieved by existing SSE constructions. In particular, we stress the fact that SWiSSSE is the first dynamic SSE scheme in the literature to achieve strong backward privacy guarantees against system-wide leakages.

Forward Privacy. Forward private SSE was introduced by Chang and Mitzenmacher in [11], and has been subsequently studied in [43, 5, 4, 15, 29, 6, 14, 42]. 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 [7, 47], which are potentially devastating for SSE schemes that try to support updates without being forward private.

Observe that combining Theorems 4 and 5 allows us to claim that our dynamic SSE scheme achieves stronger forward privacy guarantees than existing constructions in the literature, *including* those based on computation/communication-intensive techniques such as ORAM [15, 4, 6, 10]. 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 5, 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 4, 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 [6]. Subsequently, Chamani *et al.* [10] and Sun *et al.* [44] proposed SSE schemes supporting single keyword search that are backward private under various leakage profiles. The strongest notion of backward privacy formalized in [6] is called Type-1 backward privacy [6]. 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

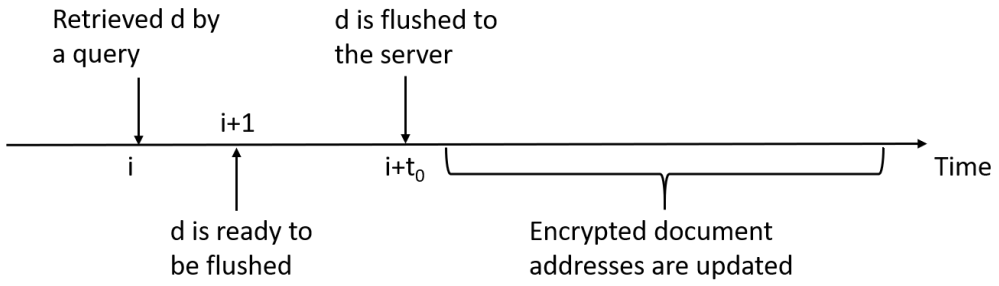

Figure 5: An illustration of the operations related to a document d in our construction. At time i, the document d is retrieved by a query. The document is in the stash and ready to be written back at time $i+1$. The document itself is written back at time $i+t_0$, where t_0 is a random delay due to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

many communication rounds for searches [6, 10].

Once again, Theorem 4 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 4 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. An illustration of the same is presented in Figure 5.

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 $1 \leq i < k$. 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

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

Figure 6: A summary of the performance parameters. Here, w denotes the leading keyword of a query, $\mathtt{G}(w)$ is the bucket size of keyword w, $|W\{\mathtt{DB}\}|$ is the total number of keyword-document pairs, and $|W\{\mathtt{DB}(w)\}|$ is the total number of keyword-document pairs for the documents that contain w.

one of the documents retrieved in query q_i where $1 \leq i < k$ 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 leakages, such as access pattern and query equality pattern based attacks [23, 7], file injection attacks [47], and attacks based on highly refined leakages (such as the correlation-leakage based attack described in Section 6). Also noteworthy is the fact that SWiSSSE achieves such strong guarantees without compromising significantly on search/update performance and communication overheads. This makes it an attractive candidate for deployment in typical applications involving outsourced databases.

8 SWiSSE: Performance Evaluation

In this section, we provide an asymptotic performance analysis of SWiSSSE in both the static and dynamic cases. A summary of the key performance characteristics can be found in Figure 6 and some concrete numbers are presented subsequently.

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 $2 \cdot \max_w \mathtt{G}(w)$ (half of that comes from randomly generated document addresses), there are at most $4 \cdot \max_w \mathtt{G}(w)$ documents in the stash. The documents are padded to a constant size, which means the storage of the documents in the stash requires $\mathcal{O}(\max_w \mathtt{G}(w))$ space. For document insertion queries, the documents to be inserted are processed with the responses, so the same analysis on the space complexity

applies.

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 $|W\{\mathtt{DB}(w)\}|$. Consider a document insertion query with document d and keywords $\{w_1, \ldots, w_k\}$. Without loss of generality, assume that w_1 is the leading keyword. Then the number of lookup index locations that need to be updated is at most $|W\{\mathtt{DB}(w_1)\}| + k - 1$. Since the number of keywords in the document is much smaller than $|W\{\mathtt{DB}(w_1)\}|$, 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 $\mathcal{O}(\max_w |W\{\mathtt{DB}(w)\}|)$.

In addition, 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. 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 $\mathcal{O}(\max_w \mathtt{G}(w) + \max_w |W\{\mathtt{DB}(w)\}|)$.

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 $\mathcal{O}(\sum_w \mathtt{G}(w) + |\mathtt{DB}|)$. 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. 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 $\mathcal{O}(\mathtt{G}(w))$ computation and communication, as there are at most $3 \cdot \mathtt{G}(w)$ addresses involved. The server then takes $\mathcal{O}(\mathtt{G}(w))$ time to retrieve the encrypted document array addresses and send them to the client. This means that the overall communication volume is $\mathcal{O}(\mathtt{G}(w))$ for the first round.

Upon receiving the $\mathcal{O}(\mathtt{G}(w))$ encrypted document array addresses, the client processes them and retrieves $2 \cdot \mathtt{G}(w)$ 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 $\mathcal{O}(\mathtt{G}(w))$. The communication volume and the time complexity for the server are straight-forwardly $\mathcal{O}(\mathtt{G}(w))$.

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

Combining the analyses above together, we conclude that the time complexity of a query for both the client and the server is $\mathcal{O}(\max_w \mathtt{G}(w) + \max_w |W\{\mathtt{DB}(w)\}|)$, while the communication volume of a query is $\mathcal{O}(\max_w \mathtt{G}(w) + \max_w |W\{\mathtt{DB}(w)\}|)$. 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 $\mathcal{O}(\mathtt{G}(w))$ and the communication volume is $\mathcal{O}(\mathtt{G}(w))$.

Alternative Parameters. There are three parameters we have fixed when we first introduced the construction, namely the number of fake documents, the number of dummy documents retrieved per query and the write back rate. These parameters can be changed to achieve different trade-off between efficiency and security.

The number of fake documents we initialise the database with affects the number of insertion queries we can make on the database. By using a larger number, the database can support more insertion queries, but as the fake documents have to be initialised with dummy documents during the setup, the time required to initialise the database is longer and the storage on the server takes more space. Maybe unexpectedly, more fake documents implies less noise in the co-occurrence leakage, as each document contains fewer keywords. This may mean that a more aggressive padding parameter or padding strategy has to be used to prevent attacks based on co-occurrence leakage.

Recall that in our construction, every time when the client wants to retrieve some documents, he retrieves an equal number of dummy documents by generating addresses at random. This helps to introduce noise in the co-occurrence leakage and prevents an attacker from identifying the set of retrieved documents exactly. However, if the user is willing to leak the document access pattern then he can set the number of dummy document retrievals to zero, as long as the resultant keyword co-occurrence leakage does not lead to a query recovery attack.

In the write-back phase, we set the number of keyword-document pairs and documents to be written back to half of the size of the stash. The client can opt to write back a different fraction of the stash, whenever the stash is full, or after a random number of queries each time. The choice affects the client storage, the probability of leaking the search pattern and the level of noise in the keyword co-occurrence leakage. For example, if the client decides to flush everything in the stash every 4 iterations, the worst-case client storage doubles, the probability of leaking the search pattern is upper-bounded by 25%, and the signal-to-noise ratio in the keyword co-occurrence leakage quarters.

9 Experimental Evaluation

In this section, we benchmark a prototype implementation of SWiSSSE that supports both static and dynamic databases. The dynamic version of SWiSSSE additionally supports

insert and delete in a manner that makes them indistinguishable from search queries. Full details of dynamic SWiSSSE are presented in Appendix 7. An asymptotic performance evaluation of SWiSSSE can be found in Appendix 8.

Overview. As target database we chose the Enron email corpus [46]. 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 schemes. Details of the database metadata and the pre-processing procedure we used on the database can be found in Appendix B. We run experiments on sub-databases of different sizes ranging from 50K documents to 400K documents. For each database, we test the performance of search queries by querying 1000 keywords in the database, and we test the performance of document insertion by initialising a database with 100K documents that is full of fake documents and then inserting 1000 real documents.

Experimental Setup. We instantiated the PRF in our construction with HMAC-SHA-256 [31] and the encryption function with AES-GCM [36, 35]. Hence our keys for the key-value stores are 32-byte values. We implemented the client in Java [45], using the Java Cryptography Extension [37] as the underlying cryptographic library; AES-NI was enabled in our implementation. We chose to use a single-thread implementation as it provides the most accurate measurements of performance. We implemented the server in Java with Redis [1] as the underlying database system.

We ran our experiments on an Intel i7-7700K CPU clocked at 4.7 GHz, 32 GB DDR4 memory clocked at 2400 MHz, and 1 TB NAND SSD with 660 MBps drive transfer rate. For all of our experiments, we used a fixed bucketization strategy with bucket size equal to 200. We ran the server and client on the same machine over localhost to simulate a LAN environment.

Benchmarks. We benchmarked keyword search queries over databases with 50K documents to 400K documents and query the first 1000 keywords in alphabetical order. We report the setup time of the databases and the average write-back time for the queries in Table 1 and the average document retrieval time in Figure 7. The setup time naturally grows linearly with the size of the database, while the online query response timegrows linearly in the response size. The offline write-back time also grows linearly in the response size (of previously retrieved documents).

To benchmark insertions, we inserted 1000 documents in a database with 100K documents. The performance of an insertion query is on par with that of a search query. As a concrete example, the latency for keyword search is 0.0250 ms, while the latency for insertion is 0.105 ms.

The relatively large response sizes for insertion queries are due to the initial read operations performed by the client. We view this as a tradeoff that allows us to achieve strong security guarantees – in particular, oblivious operations (indistinguishable searches and updates; see Appendix 7 for details).

Comparison with Plaintext database. We have also implemented a simple plaintext database and compared its performance to its SSE counterpart. The performance on search

Raw		Padded		Storage (MB)		Performance (ms)	
Documents	Storage (MB)	Documents	KDP	Stash	Server	Setup	Write-back
50000	132.1	139634	6755996	10.8 (0.18)	449.9	266044	2304
100000	338.0	282708	13587084	15.4 (0.36)	906.7	597279	5204
200000	590.3	578738	23480196	21.5 (0.53)	1657.7	1010530	6901
400000	1321.6	1125590	42801316	47.0 (0.88)	3092.8	2254622	12192

Table 1: Experimental results for search queries. Size of the stash is shown as the worst-case size, with the average-case size in brackets. KDP denotes the number of keyword-document pairs. The write-back times reported are averaged over all queries.

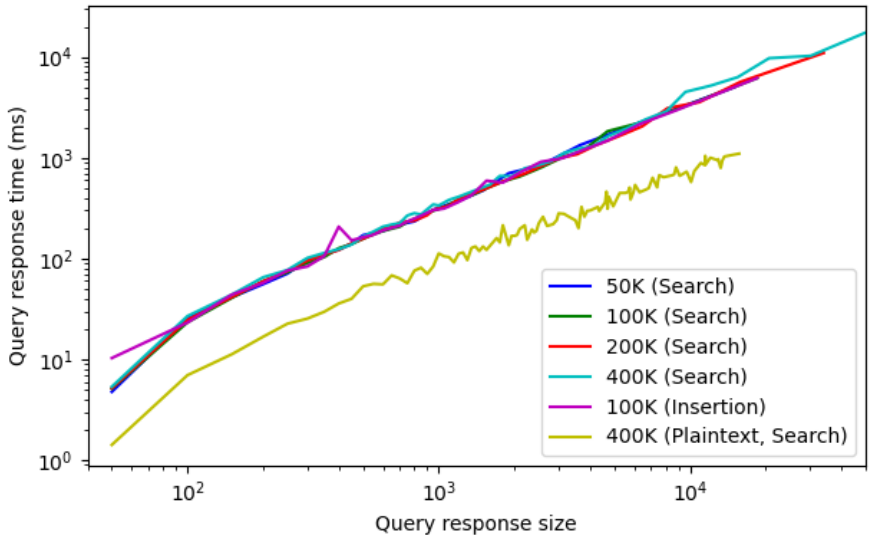

Figure 7: Query response times for search queries and insertion queries on databases and responses of different sizes. Query response time is defined to be the time between the start of the query to when the client receives the unencrypted documents. Write-backs are not included in the timing. Both axes are log scale; there is a clear linear relationship between query response time and response size, independent of database size.

queries can be seen in Figure 7. Our SSE scheme is around 10x slower than the plaintext implementation.

Discussion. 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 significantly improve efficiency. Other potential optimizations include switching to AES-CTR in place of AES-GCM and to a specialised PRF such as SipHash [3] 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, we remind the reader that each serach/insertion operation only involves two round trips; in addition, our experiments do take into account latencies due to communication through localhost.

References

[1] Redis. `https://redis.io/`. Accessed: 2020-10-15.

[2] Hime Aguiar e Oliveira Junior, Lester Ingber, Antonio Petraglia, Mariane Rembold Petraglia, and Maria Augusta Soares Machado. *Adaptive Simulated Annealing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT 2012*, volume 7668 of *LNCS*, pages 489–508, Kolkata, India, December 9–12, 2012. Springer, Heidelberg, Germany.

[4] Raphael Bost. $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1143–1154, Vienna, Austria, October 24–28, 2016. ACM Press.

[5] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. Cryptology ePrint Archive, Report 2016/062, 2016. `http://eprint.iacr.org/2016/062`.

[6] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1465–1482, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 668–679, Denver, CO, USA, October 12–16, 2015. ACM Press.

[8] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, San Diego, CA, USA, February 23–26, 2014. The Internet Society.

[9] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[10] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1038–1055, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[11] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 442–455, New York, NY, USA, June 7–10, 2005. Springer, Heidelberg, Germany.

[12] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 79–88, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press.

[13] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2433–2450. USENIX Association, 2020.

[14] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[15] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 563–592, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM STOC'09*, pages 169–178, 2009.

[17] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216`.

[18] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[19] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 315–331, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[20] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy*, pages 1067–1083, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[21] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 361–378. ACM Press, November 11–15, 2019.

[22] Johanna Hardin, Ghassan Sarkis, and P. C. Urc. Network Analysis with the Enron Email Corpus. *arXiv e-prints*, page arXiv:1410.2759, October 2014.

[23] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society.

[24] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 94–124, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[25] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. Cryptology ePrint Archive, Report 2018/978, 2018. `https://eprint.iacr.org/2018/978`.

[26] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 183–213, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[27] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 965–976, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[28] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1329–1340, Vienna, Austria, October 24–28, 2016. ACM Press.

[29] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1449–1463, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[30] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Proceedings of the 15th European Conference on Machine Learning*, ECML'04, page 217–226, Berlin, Heidelberg, 2004. Springer-Verlag.

[31] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[32] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy*, pages 297–314, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[33] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. Result pattern hiding searchable encryption for conjunctive queries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 745–762, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

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

[35] David A. McGrew and John Viega. The security and performance of the galois/counter mode of operation (full version). Cryptology ePrint Archive, Report 2004/193, 2004. `http://eprint.iacr.org/2004/193`.

[36] National Institute of Standards and Technology Gaithersburg MD. Specification for the advanced encryption standard (aes). Federal Information Processin Standards Publication 197, 2001.

[37] Oracle. Java cryptography architecture (jca) reference guide, 4 2020.

[38] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Symmetric searchable encryption with sharing and unsharing. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *ESORICS 2018, Part II*, volume 11099 of *LNCS*, pages 207–227, Barcelona, Spain, September 3–7, 2018. Springer, Heidelberg, Germany.

[39] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1341–1352, Vienna, Austria, October 24–28, 2016. ACM Press.

[40] NLTK Project. *Natural Language Toolkit*. `https://www.nltk.org/`.

[41] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, USA, May 2000. IEEE Computer Society Press.

[42] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized I/O efficiency. Cryptology ePrint Archive, Report 2018/497, 2018. `https://eprint.iacr.org/2018/497`.

[43] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*, San Diego, CA, USA, February 23–26, 2014. The Internet Society.

[44] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 763–780, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

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

[46] CMU William W. Cohen, MLD. Enron email dataset.

[47] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 707–720, Austin, TX, USA, August 10–12, 2016. USENIX Association.

A Co-occurrence Leakage Attack: Details

The Simulated-Annealing Process. We present here the details of the simulated annealing algorithm [2] used to search for the most likely assignment of keywords to the rows

Algorithm 18 Co-occurrence Leakage Attack based on Simulated Annealing

```
Random_Permutation(): generate a random permutation on the keywords of M̄.
temperature(T, i): computes new temperature based on the current temperature and the iteration.
neighbour(P): randomly generate a permutation that is close to P.

procedure ATTACK(M̄, M, T_0, t_max)
    P ← Random_Permutation(M̄)
    T ← T_0
    for i ← 0, ..., t_max do
        T ← temperature(T, i)
        P_new ← neighbour(P)
        if Pr[P_new | M̄, M] − Pr[P | M̄, M] > −T then
            P ← P_new
    return P
```

of the observed co-occurrence matrix $\bar{M}$. The algorithm takes in four inputs, namely the observed co-occurrence matrix M, the observed co-occurrence matrix $\bar{M}$, a real number T_0 which is known as the initial temperature, and an integer k which specifies the number of iterations of the algorithm.

On a high level, the algorithm works as follows. The algorithm begins by generating a random assignment of keywords s and set the temperature T to T_0. For the subsequent iterations, the algorithm decreases T and randomly permutes two of the keywords in s to get a new assignment s'. The likelihoods scores of the new assignment is computed and compared to that given by s. If the likelihood score has improved, then the algorithm replaces s with s'. On the other hand, if the likelihood score decreased, then the new assignment is used to replace the old one if the difference of likelihood is less than T, and the old assignment is kept otherwise. The pseudocode of the algorithm is shown in Algorithm 18.

Next, we derive the likelihood function used in the simulated annealing process.

NOTATION. Let $M \in \mathbb{R}^{n \times n}$ be the real co-occurrence matrix where n is the number of keywords. Let $\mathcal{K}$ be a vector of n keywords associated to M, i.e. $M_{i,j}$ corresponds to $|\mathtt{DB}(\mathcal{K}_i) \cap \mathtt{DB}(\mathcal{K}_j)|$. Let $\bar{M} \in \mathbb{R}^{n \times n}$ be the observed co-occurrence matrix where n is the number of keywords. Let $\mathcal{K}'$ be a vector of n keywords associated to $\bar{M}$. Let $P : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a permutation such that $\mathcal{K}_{P(i)} = \mathcal{K}'_i$ for all $i \in \{1, \ldots, n\}$. Then the goal of our attack is to find P given M and $\bar{M}$.

OVERVIEW OF THE DERIVATION. The ultimate goal of the derivation is to find $\mathsf{Pr}\left[P \mid M, \bar{M}\right]$, the probability that the permutation applied on the real co-occurrence matrix is P before deriving the observed co-occurrence matrix $\bar{M}$. As an abuse of notation, we apply P as a permutation matrix on $\bar{M}$ directly and write $\mathsf{Pr}\left[P\bar{M}P^t \mid M\right]$ to mean $\mathsf{Pr}\left[P \mid M, \bar{M}\right]$. Let

M' be a co-occurrence matrix after padding, then we can write the probability as

$$\begin{aligned}
&\mathsf{Pr}\left[P\bar{M}P^t \mid M\right] \\
=&\sum_{M' \in \mathcal{R}^{n\times n}} \mathsf{Pr}\left[P\bar{M}P^t, M' \mid M\right] \\
=&\sum_{M' \in \mathcal{R}^{n\times n}} \mathsf{Pr}\left[P\bar{M}P^t \mid M'\right] \cdot \mathsf{Pr}\left[M' \mid M\right].
\end{aligned}$$

The first term in the product is the probability of observing the co-occurrence counts in $\bar{M}$ given that P is the permutation and M' is the co-occurrence matrix after padding is applied. The second term is the probability that M' is the padded co-occurrence matrix given that M is the co-occurrence matrix before padding. It remains to derive the expressions for the two probabilities.

Derivation of the First Probability. We assume that the entries of the observed co-occurrence counts are independently generated. As half of the documents are written back in each iteration, the count observed in position (i, j) of $P\bar{M}P^t$ is distributed according to $\mathsf{Binom}([M']_{i,j}, 0.5)$ for all $i \neq j$. Hence, we can write the probability as:

$$\begin{aligned}
&\mathsf{Pr}\left[P\bar{M}P^t \mid M'\right] \\
=&\prod_{i<j} \mathsf{Pr}\left[[P\bar{M}P^t]_{i,j} \mid [M']_{i,j}\right] \\
=&\prod_{i<j} \mathsf{Pr}\left[\mathsf{Binom}([M']_{i,j}, 0.5) = [P\bar{M}P^t]_{i,j}\right].
\end{aligned}$$

Derivation of the Second Probability. The second probability represents the likelihood that each of the entry in M is padded to M'. As before, we assume that the process is independent for each pair of keywords. Without loss of generality, consider the i-th and j-th keywords of the database. Suppose the i-th keyword is padded to c_i from $M_{i,i}$ and the j-th keyword is padded to c_j from $M_{j,j}$. Our goal is to find the probability that the co-occurrence count with the two keywords increases from $M_{i,j}$ to $M'_{i,j}$.

There are three ways a fake keyword pair can appear. Firstly, it can be the case that there is a document with the i-th keyword and the j-th keyword is added to it. Secondly, it can be the case that both the i-th and j-th keywords are added to the same document that did not have them before. Finally, it can be the case that there is a document with the j-th keyword and the i-th keyword is added to it. Let d be the total number of documents after padding, then the number of co-occurrence counts generated by each way can be expressed as a hypergeometric distribution with parameters,

$$\begin{aligned}
&(d - M_{i,j}, M_{i,i} - M_{i,j}, \bar{M}_{j,j} - M_{j,j}), \\
&(d - M_{i,j}, \bar{M}_{i,i} - M_{i,i}, \bar{M}_{j,j} - M_{j,j}), \\
&(d - M_{i,j}, M_{j,j} - M_{i,j}, \bar{M}_{i,i} - M_{i,i})
\end{aligned}$$

respectively. Call the three variables $X_{i,j,1}, X_{i,j,2}$ and $X_{i,j,3}$ respectively, we can write the

likelihood as a product of convolutions as:

$$\begin{aligned}&\mathsf{Pr}\left[M' \mid M\right]\\ =&\prod_{i<j}\mathsf{Pr}\left[\sum_{k=1}^{3} X_{i,j,k} = M'_{i,j} - M_{i,j}\right].\end{aligned}$$

Computational Approximations and Optimisations. The likelihood function derived above has two computational problems. First of all, there are infinitely many M' in the summation and there is no closed form for the infinite sum. To tackle the problem, we propose to sum over a small range where $\mathsf{Pr}\left[P\bar{M}P^t \mid M'\right]$ is reasonably large. For each $[P\bar{M}P^t]_{i,j}$, we compute the 95% confidence interval for n of $\mathsf{Binom}(n, 0.5)$ assuming that $[P\bar{M}P^t]_{i,j}$ is drawn from it, and use it as the range for $M'_{i,j}$.

Secondly, there are three nested summations in the likelihood expression. The first one comes from the sum over M' and the other two comes from the convolution in $\mathsf{Pr}\left[\sum_{k=1}^{3} X_{i,j,k} = M'_{i,j} - M_{i,j}\right]$. This means it is time consuming to compute a likelihood score and hence, within a given time, less iterations can be performed, which leads to a less optimal solution. To solve the problem, we decided to approximate the fake co-occurrence counts with a single hypergeometric distribution with parameters $(d - M_{i,j}, \bar{M}_{i,i} - M_{i,j}, \bar{M}_{j,j} - M_{i,j})$.

Finally, we note that if we only change two rolls in the permutation P, then only the corresponding rows and columns of the likelihood within the summation are affected. This means that we can store the score as an n by n matrix, and recompute the affected rows and columns only.

B Description of Experimental Database

In this appendix we describe the Enron email corpus used in our experiments, and the steps we took to process the emails and extract the keywords. We also provide general statistics of the database.

The Enron email corpus. The Enron email corpus is a collection of over 600K emails generated by 158 employees of the Enron Corporation and acquired by the Federal Energy Regulatory Commission (FERC) during its investigation of the Enron scandal. At the conclusion of the investigation, and upon the issuance of the FERC staff report, the email corpus is released to the public for historical research and academic purposes. The version of the email corpus used in our paper can be freely accessed from [46]. The database has slightly less emails, about 500K of them, as some messages have been deleted "as part of a redaction effort due to requests from affected employees". The email corpus is widely used in research fields such as studies on social networking [22] and computer analysis of language [30], and more recently, it has been a popular choice of experimental database used in the searchable encryption literature [47, 38].

Email processing and keyword extraction. We implemented our email processing and keyword extraction script in python using the Natural Language Toolkit [40] module as

# documents	480228
# keywords	33372
# keyword-document pairs	17428755
Max. keyword rank	23998
Min. keyword rank	1
Mean keyword rank	522.3
Max. # keywords per document	3483
Min. # keywords per document	1
Mean # keywords per document	36.3

Figure 8: General statistics of the Enron email corpus after pre-processing.

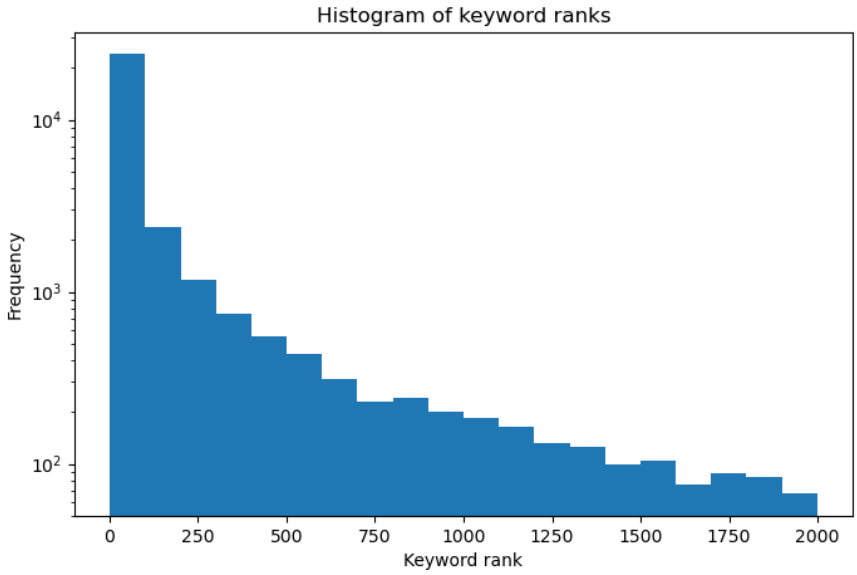

Figure 9: Histogram of the keyword frequency.

the tokeniser. The headers of the emails are not used in the keyword extraction process. We decided to convert all the keywords to the lower case but this is not necessary in general. The English stop words and the keywords with frequency higher than 5% are removed from the set of keywords for each email. The emails with no keyword after this step are removed from the database. The content of the emails are compressed with Java's built-in GZIP compression algorithm. The compressed emails that are larger than 1KB are then partitioned into chunks of 1KB files.

General Statistics. We provide some general statistics of the database in Figure 8 and the histogram of the keyword ranks in Figure 9.