Breach-Resistant Structured Encryption

Ghous Amjad[*] Brown University

Seny Kamara[†] Brown University

Tarik Moataz[‡] Brown University

Abstract

Motivated by the problem of data breaches, we formalize a notion of security for dynamic structured encryption (STE) schemes that guarantees security against a *snapshot* adversary; that is, an adversary that receives a copy of the encrypted structure at various times but does not see the transcripts related to any queries. In particular, we focus on the construction of dynamic encrypted multi-maps which are used to build efficient searchable symmetric encryption schemes, graph encryption schemes and encrypted relational databases. Interestingly, we show that a form of snapshot security we refer to as *breach resistance* implies previously-studied notions such as a (weaker version) of history independence and write-only obliviousness.

Moreover, we initiate the study of *dual-secure* dynamic STE constructions: schemes that are forward-private against a persistent adversary and breach-resistant against a snapshot adversary. The notion of forward privacy guarantees that updates to the encrypted structure do not reveal their association to any query made in the past. As a concrete instantiation, we propose a new dual-secure dynamic multi-map encryption scheme that outperforms all existing constructions; including schemes that are not dual-secure. Our construction has query complexity that grows with the selectivity of the query and the number of deletes since the client executed a linear-time rebuild protocol which can be de-amortized.

We implemented our scheme (with the de-amortized rebuild protocol) and evaluated its concrete efficiency empirically. Our experiments show that it is highly efficient with queries taking less than 1 microsecond per label/value pair.

[*]`ghous_amjad@brown.edu`.

[†]`seny@brown.edu`.

[‡]`tarik_moataz@brown.edu`.

Contents

1 Introduction

The constant occurrence of data breaches has generated a lot of interest from academia, industry and government in the subject of encrypted search and, in particular, in encrypted databases (EDB). Because EDBs can protect data at all times—even in use—they inherently provide stronger security and privacy guarantees than standard database systems. There are several ways in which a data breach can occur: (1) the server that runs the database system is compromised; (2) the database itself is somehow exfiltrated; and (3) the application is compromised and the database is retrieved using the standard database interface. The first threat can be modeled as a *persistent* adversary that controls the database server at all times. The second can be captured by a *snapshot* adversary that only gets a copy of the database at specific points in time. Note that the third cannot be addressed using cryptographic techniques because once the adversary compromises the application and gets its credentials, it is indistinguishable from the application.

Encrypted search solutions have traditionally been designed to address persistent adversaries with the understanding that security against a persistent adversary implies security against snapshot adversaries. While this is indeed the case in the *static* setting (for structures that do not modify themselves during query operations), it is not necessarily true in the dynamic case. In fact, solutions designed specifically against a persistent adversary could still leak query information to a snapshot adversary that one would expect to be hidden. The problem of designing snapshot-secure EDBs, therefore, is non-trivial and is even more challenging when security against both a persistent and a snapshot adversary is required.

Structured encryption. A promising approach to designing EDBs relies on *structured encryption* schemes which provide a balance of security, efficiency, expressiveness. A structured encryption scheme encrypts a data structure in such a way that it can be privately queried. Roughly speaking, an STE scheme is secure if it reveals nothing about the underlying structure and queries beyond some well-specified leakage. Special cases of STE include graph [10, 32], dictionary [10, 8] and, in particular, multi-map encryption schemes [11, 8, 4], which are going to be the focus of this work.

Multi-maps are data structures that store pairs of the form $(\ell, \mathbf{v})$, where ℓ is a label and $\mathbf{v} = (v_1, \ldots, v_n)$ is a tuple of n values. Multi-maps support a get operation that takes as input a label and returns its associated tuple. Encrypted multi-maps (EMM) naturally yield single-keyword SSE schemes by storing pairs of the form (w, id) where w is a keyword and id is a tuple of identifiers for the files that contain w. In addition, EMMs can be used as building blocks to design graph encryption schemes [10], boolean SSE schemes [6, 24] and encrypted relational databases [27].

In many practical scenarios, encrypted data structures need to be *dynamic*; that is, able to support additions and deletions. As such, efficient dynamic SSE/EMM constructions have received a lot of attention [25, 28, 40, 36, 8, 22, 37, 15, 4, 33, 24, 5, 14].

Forward-privacy. The most recent work on dynamic EMMs has focused on the notion of forward privacy, first proposed by Stefanov, Papamanthou and Shi [40]. Roughly speaking, forward privacy guarantees that updates to the structure do not reveal their association to any query made in the past. While forward privacy is a useful security property in of itself, it has also been shown to mitigate the adaptive file injection attacks of Zhang, Katz and Papamanthou [41] (though not the non-adaptive attacks).

Currently, the known forward private EMMs are the SPS construction by Stefanov et al. [40], the Sophos construction of Bost [4], and the EKPE construction by Etemad, Küpçü, Papamanthou and Evans [14]. Recently, Bost, Minaud and Ohrimenko [5] presented the first backward-private SSE schemes, a notion that was (informally) introduced in [40]. At a high level, backward privacy guarantees that queries do not reveal their association to deleted documents. While there are no known attacks that leverage the lack of backward privacy, improving the security guarantees of EMMs is well-motivated.

Snapshot security. While most STE constructions are known to be adaptively-secure against a *persistent* adversary, as far as we know, no previous work has considered STE in the context of *snapshot* adversaries. Informally, a persistent adversary has access to the encrypted structured and has access to the transcripts of the interactions between the client and the server. A snapshot adversary, on the other hand, only has access to the encrypted structure (but at various times). Given that snapshot adversaries are clearly weaker, it motivates the following natural question:

> *Can we design efficient and dynamic EMMs that are secure against a persistent adversary and are (almost) zero-leakage against a snapshot adversary?*

We answer this question positively and in doing so introduce a new kind of dynamic EMM which are efficient and secure against both snapshot and persistent adversaries. Specifically, these EMMs are *forward-private* against persistent adversaries and *breach-resistant* against snapshot adversaries, in the sense that they leak only the size of the structure at the time of the snapshot. Interestingly, while our constructions offer stronger security guarantees than previous work, it also achieves better asymptotic efficiency in terms of query and update complexity, token and encrypted structure size and client storage (under reasonable assumptions).

1.1 Our Contributions

In this work, we revisit dynamic EMMs in several ways.

Breach resistance. Snapshot adversaries are motivated by the threat of data breaches. In such a scenario, the adversary is not necessarily the server itself but some other party that gets access to a copy of the encrypted structure at some point(s) in time.[1]

We propose a formal definition of snapshot security which, intuitively, requires that a copy of the encrypted structure reveals nothing about the structure and the past operations beyond what is revealed by a well-specified leakage function we refer to as the *snapshot leakage*. We then say that an STE scheme is *breach-resistant* if its snapshot leakage reveals at most the current size of the structure. Our notion of breach resistance is similar to the notion of offline security of Lewi and Wu in the setting of property-preserving encryption (PPE) [30]. We also show that, breach-resistance implies some interesting and previously studied notions such as (a weaker version) of history independence [35] and write-only obliviousness [3].

We stress that our notion of breach resistance applies only to the encrypted structures we design and that, as pointed out by Grubbs, Ristenpart and Shmatikov [21], there are non-trivial implementation questions that have to be considered when they are integrated into real-world systems.

[1]Data breaches that occur as a consequence of the disclosure of user credentials (e.g., through phishing) cannot be addressed with cryptography so are out of scope for this work.

Forward privacy. Much like recent works, our proposal is also forward-private. Our scheme, DLS, does not make use of ORAM simulation or public-key operations. Furthermore, its query complexity grows only with the number of delete operations performed since the client executed (linear-time) rebuild protocol; as opposed to the Sophos [4] and Diana [5] constructions whose complexity grows with the total number of deletes ever performed. Our construction is also much more efficient as it only uses simple symmetric-key operations as opposed to other schemes which use public-key operations or constrained PRFs. Compared to the EKPE construction [14], which is not snapshot-secure and forward-private only for add operations, DLS offers better asymptotics if the scheme is used in scenarios in which the structure has more frequent updates and less frequent queries.

Surprisingly, forward privacy does not imply breach resistance. In fact, several known forward-private constructions are not necessarily breach-resistant [4, 14, 5]. [2] While this may seem counter-intuitive, it reinforces the need of rigorous security analysis of STE constructions against different types of adversaries.

Dual security. After formally defining snapshot security, we construct, as far as we know, the first provably-secure breach-resistant encrypted multi-map. In addition, our construction is also forward-private which means that it can be used to protect against both persistent and snapshot adversaries. We refer to such constructions as being *dual-secure*. Our scheme, DLS, achieves both notions of security, and we achieve our desirable asymptotic query complexity by an explicit rebuild protocol that must be executed by the client at certain times. We show, however, that the protocol can be de-amortized and we call this variant $\mathrm{DLS}^{\mathsf{d}}$.[3]

To sum up, DLS features the best query, token size, client memory, update and storage complexity among all dual-secure schemes in literature.

Experimental evaluation. We implemented $\mathrm{DLS}^{\mathsf{d}}$ (the variant with de-amortized rebuilding) in Java and evaluated its performance on the Wikipedia dataset. Our experiments show that this construction is highly practical. We ran experiments on up to 83 million label/value pairs on an Amazon EC2 instance with 32 vCPUs and 60GB of memory.

DLS is compact, producing EMMs of size 9.4GB for a 3.6GB folder composed of $554,059$ files. It has a search overhead less than 1 microsecond per pair for selectivities spanning from 100 to $10,000$ pairs. On the other hand, it has an update overhead of around 100 milliseconds per pair. This slowdown is mainly due to the de-amortized rebuilding protocol.

2 Related Work

Chase and Kamara introduced structured encryption as a generalization of SSE, along with several adaptively-secure constructions [10]. Subsequent works have focused on improving several problems of dynamic [18, 26, 25, 40, 4, 8], I/O-efficient [8, 33], local [7, 1, 13], forward-private [40, 17, 4, 5, 14], expressive [10, 6, 38, 15, 24], and multi-user [11, 23] SSE.

[2] The Sophos [4] and $\mathrm{Diana}_{\mathsf{del}}$ constructions are breach-resistant only if deletes are handled in the same structure as adds.

[3] While this can be of independent interest, DLS results in a very efficient write-only oblivious multi-map which can replace several existing constructions, e.g., the ones used in the hidden volume scheme HIVE [3] or in the oblivious file backup system ObliviSync [2].

Dynamic SSE. Many works have considered the problem of dynamic SSE schemes. Kamara, Papamanthou and Roeder gave the first construction with optimal search complexity [26]. Kamara and Papamanthou proposed a dynamic sub-linear scheme with parallel and I/O efficient search [25]. Kurosawa and Ohtaki considered the problem of dynamic verifiable SSE [28]. Naveed, Prabhakaran and Gunther proposed an efficient dynamic SSE scheme based on blind storage. Cash, Jaeger, Jarecki, Jutla, Krawczyk, Rosu and Steiner described I/O-efficient static and dynamic SSE schemes based on a dictionary structure [8]. Our construction follows the same dictionary-based approach. Pappas, Krell, Vo, Kolesnikov, Malkin, Choi, George, Keromytis and Bellovin gave an interactive construction with sub-linear search and complex queries [37, 15].

Forward privacy. Stefanov, Papamanthou and Shi first proposed the notion of forward privacy in the context of SSE and gave a construction based on ORAM techniques [40]. Their construction has worst-case search complexity that is poly-logarithmic in the total number of label/value pairs per value returned. Bost [4] proposed the Sophos construction which is the first practical forward-private dynamic SSE scheme. A recent followup by Bost, Minaud and Ohrimenko [5] proposed several practical constructions that are forward and backward-private. The Sophos [4] and Diana [5] constructions have a search complexity that grows linearly with the number of values and the total number of deletes ever performed. So they can only achieve optimal-time search if no delete operations are ever performed. Garg, Mohassel and Papamanthou [17] presented a forward-private dynamic SSE construction, TWORAM, that hides the search/query pattern by leveraging ORAM and garbled RAM techniques. Kamara and Moataz described various constructions, including a boolean dynamic SSE scheme which can be made forward-private [24]. Recently, Etemad, Küpcü, Papamanthou and Evans [14] proposed a forward-private (only for files additions) SSE scheme with search complexity that grows linearly in the number of additions. Authors also propose a dynamic SSE scheme that handles both files additions and deletions with better asympotitcs, but in this case the scheme is not forward-private.

Snapshot security. Snapshot security has been discussed informally in previous works [39, 21]. As far as we know, the only formal treatment was given by Lewi and Wu in [30] for the setting of PPE. Under our formulation of snapshot security for STE, there are a few existing constructions that are breach-resistant including the SPS construction of Stefanov et al. [40], the Sophos construction of Bost [4], the TWORAM-based scheme of Garg, Mohassel and Papamanthou [17] and the Diana construction of Bost, Minaud and Ohrimenko [5]. Note that Sophos and Diana constructions are breach-resistant if deletes are handled in the main table. While these schemes are breach-resistant, they either use ORAM and/or garbled RAM techniques [40, 17] or have query complexity linear in the number of updates [4, 5]. Our construction addresses all these limitations and results in the most efficient breach-resistant EMM all the while providing forward-privacy.

3 Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$, and $2^{[n]}$ is the corresponding power set. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \overset{\$}{\leftarrow} X$ to represent an element x being sampled uniformly at random from a set X. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{o}$ of n elements, we refer to its ith element as

$\mathbf{o}_i$ or $\mathbf{o}[i]$. If T is a set then $\#T$ refers to its cardinality. Given strings x and y, we refer to their concatenation as either $\langle x, y\rangle$ or $x\|y$.

Data types. An *abstract data type* is a collection of objects together with a set of operations defined on those objects. Examples include sets, dictionaries (also known as key-value stores or associative arrays) and graphs. The operations associated with an abstract data type fall into one of two categories: query operations, which return information about the objects; and update operations, which modify the objects. If the abstract data type supports only query operations it is *static*, otherwise it is *dynamic*. For simplicity we define data types as having a single operation and note that the definitions can be extended to capture multiple operations in the natural way.

Data structures. A *data structure* for a given data type is a representation in some computational model [4] of an object of the given type. Typically, the representation is optimized to support the type's query operation as efficiently as possible. For data types that support multiple queries, the representation is often optimized to efficiently support as many queries as possible. As a concrete example, the dictionary *type* can be represented using various data structures depending on which queries one wants to support efficiently. Hash tables support Get and Put in expected $O(1)$ time whereas balanced binary search trees support both operations in worst-case $\log(n)$ time. For ease of understanding and to match colloquial usage, we will sometimes blur the distinction between data types and structures. So, for example, when referring to a *dictionary structure* or a *multi-map structure* what we are referring to is an unspecified instantiation of the dictionary or multi-map data type. Given a sequence of operations $\mathbf{op} = (\mathsf{op}_1, \ldots, \mathsf{op}_n)$, each of which can be a query, an addition or a deletion, we denote by $\mathsf{add}(\mathbf{op})$ the subsequence of addition operations and by $\mathsf{del}(\mathbf{op})$ the subsequence of delete operations. We further define the subsequence of update operations as $\mathsf{up}(\mathbf{op}) = \mathsf{add}(\mathbf{op}) + \mathsf{del}(\mathbf{op})$.

Basic structures. We make use of several basic data types including dictionaries and multi-maps which we recall here. A dictionary DX of capacity n is a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i\leq n}$ and supports Get and Put operations. We write $v_i = \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] = v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i. We denote by $\mathbb{L}_{\mathsf{DX}}$ the set of labels stored in DX and by $\#\mathsf{DX}$ the volume of DX which is the number of label/value pairs it holds $n = \#\mathbb{L}_{\mathsf{DX}}$. A multi-map MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{v}_i)_i\}_{i\leq n}$ that supports Get and Put operations. Similarly to dictionaries, we write $\mathbf{v}_i = \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] = \mathbf{v}_i$ to denote operation of associating the tuple $\mathbf{v}_i$ to label ℓ_i. We denote by $\mathbb{L}_{\mathsf{MM}}$ the set of labels stored in MM and by $\#\mathsf{MM}$ the volume of MM which is $\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$. Multi-maps are the abstract data type instantiated by an inverted index. In the encrypted search literature multi-maps are sometimes referred to as indexes, databases or tuple-sets (T-sets) [6, 8].

3.1 Cryptographic Primitives

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm takes a key K

[4] In this work, the underlying model will always be the word RAM.

and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[5] In addition to encryption schemes, we also make use of pseudo-random functions (PRF), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary.

4 Definitions

Structured encryption schemes [10] encrypt data structures in such a way that they can support operations on encrypted data. With encrypted data structures, we can distinguish between different types of operations. This includes interactive and non-interactive operations where the former require only a single message and the latter require several rounds. We can also distinguish between response-revealing and response-hiding operations, where the former reveal the answer to the query and the latter do not.

STE schemes are used as follows. During a setup phase, the client constructs an encrypted structure EDS from a plaintext structure DS under a key K. If the scheme is stateful, this setup procedure also outputs a state st. The client then sends the encrypted structure EDS to an untrusted server and keeps the state st and key K private. The client can then query EDS using the supported operations. If the operation is non-interactive, the client sends to the server a token tk constructed with its key K, state st and query q. The server then uses the token tk to query the encrypted structure EDS. If the operation is interactive, the client and server execute a two-party protocol where the former inputs K, st and q and the latter inputs EDS.

Self-adjusting encrypted structures. A data structure is self-adjusting if it re-arranges itself after being queried or updated. This is usually done to maintain correctness, consistency or to improve efficiency. We provide below the syntax of a self-adjustable STE scheme. Note that, here, the update operation is interactive which is not a requirement.

Definition 4.1 (Self-adjusting STE)**.** *A response-hiding dynamic structured encryption scheme $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Update}, \mathsf{Rslv})$ with non-interactive queries and interactive updates consists of four polynomial-time algorithms and one two-party protocol between the client and server that work as follows:*

- $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*: is a probabilistic algorithm that takes as input a security parameter 1^k and a structure DS. It outputs a secret key K, a state st and an encrypted structure EDS.*
- $(st', \mathsf{tk}) \leftarrow \mathsf{Token}(K, st, q)$*: is a (possibly) probabilistic algorithm that takes as input a secret key K, a state st and a query q. It outputs a new state st' and a query token tk.*

[5]RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

- $(\mathrm{ct}, \mathsf{EDS}') \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a (possibly) probabilistic algorithm that takes as input an encrypted structure* EDS *and a token* tk*. It outputs a message* ct *and an (possibly) updated encrypted structure* EDS'*.*

- $(st', \mathsf{EDS}') \leftarrow \mathsf{Update}_{C,S}\Big((K, st, u), \mathsf{EDS}\Big)$*: is a two-party protocol between the client and the server. It takes as input from the client a key K, a state st and an update u and as input from the server an encrypted structure* EDS*. It outputs to the client an updated state st' and to the server a new encrypted structure* EDS'*.*

- $r \leftarrow \mathsf{Rslv}(K, \mathrm{ct})$*: is a deterministic algorithm that takes as input a secret key K and a message* ct*. It outputs a response r.*

The syntax of response-revealing Query operation can be recovered by having it output the response r directly and omitting the Rslv algorithm.

Self-adjusting STE correctness. We say that a self-adjusting dynamic STE scheme Σ is correct if for all $k \in \mathbb{N}$, for all $\mathsf{poly}(k)$-size structures DS, for all (K, st, EDS) output by $\mathsf{Setup}(1^k, \mathsf{DS})$, for all sequences of $m = \mathsf{poly}(k)$ operations $\mathsf{op}_1, \ldots, \mathsf{op}_m$ such that $\mathsf{op}_i \in \{q_i, u_i\}$, for all query tokens tk_i and all states st' output by $\mathsf{Token}(K, st, q_i)$, for all messages ct and structures EDS' output by $\mathsf{Query}(\mathsf{EDS}, \mathsf{tk}_i)$ or all structures EDS' and all states st' output by $\mathsf{Update}_{C,S}((K, st, u), \mathsf{EDS})$, $\mathsf{Rslv}(K, \mathrm{ct})$ returns the correct response with all but negligible probability.

Rebuildable encrypted structures. We say that a data structure is rebuildable if it supports a rebuild operation that reconstructs it. Rebuild operations are typically used to improve query efficiency or storage overhead after a sequence of operations have been performed. Whereas self-adjusting structures re-arrange themselves as part of a query or update operation, rebuildable structures support an explicit rebuild operation that is typically invoked after a sequence of operations. We provide below the syntax of a rebuildable STE scheme. Note that, here, the rebuild operation is interactive.

Definition 4.2 (Rebuildable STE). *A response-hiding dynamic structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{UToken}, \mathsf{Update}, \mathsf{Rebuild}, \mathsf{Rslv})$ *with non-interactive queries and interactive rebuilds consists of seven polynomial-time algorithms and one two-party protocol between the client and server that work as follows:*

- $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*: is a probabilistic algorithm that takes as input a security parameter 1^k and a structure* DS*. It outputs a secret key K, a state st and an encrypted structure* EDS*.*

- $(st', \mathsf{tk}) \leftarrow \mathsf{Token}(K, st, q)$*: is a (possibly) probabilistic algorithm that takes as input a secret key K, a state st and a query q. It outputs a new state st' and a query token* tk*.*

- $\mathrm{ct} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a (possibly) probabilistic algorithm that takes as input an encrypted structure* EDS *and a token* tk*. It outputs a message* ct*.*

- $(st', \mathsf{utk}) \leftarrow \mathsf{UToken}(k, st, u)$*: is a (possibly) probabilistic algorithm that takes as input a secret key K, a state st and an update u. It outputs a new state st' and an update token* utk*.*

- $\mathsf{EDS}' \leftarrow \mathsf{Update}(\mathsf{EDS}, \mathsf{utk})$*: is a (possibly) probabilistic algorithm that takes as input an encrypted structure* EDS *and an update token* utk*. It outputs an updated encrypted structure* EDS'*.*

- $(st', \mathsf{EDS}') \leftarrow \mathsf{Rebuild}_{\mathsf{C},\mathsf{S}}((K, st), \mathsf{EDS})$*: is a two-party protocol between the client and the server. It takes as input from the client a key K and a state st and as input from the server an encrypted structure* EDS*. It outputs to the client an updated state st' and to the server a new encrypted structure* EDS'*.*

- $r \leftarrow \mathsf{Rslv}(K, \mathrm{ct})$*: is a deterministic algorithm that takes as input a secret key K and a message* ct*. It outputs a response r.*

In the subsequent parts of this section, we will only focus on *non-interactive* dynamic STE. The proposed definitions can be naturally extended to STE schemes with interactive queries.

4.1 Security Against a Persistent Adversary

The standard notion of security for STE guarantees that: (1) an encrypted structure reveals no information about its underlying structure beyond the setup leakage $\mathcal{L}_\mathsf{S}$; (2) that the query algorithm reveals no information about the structure and the queries beyond the query leakage $\mathcal{L}_\mathsf{Q}$; and that (3) the update algorithm reveals no information about the structure and the update beyond the update leakage $\mathcal{L}_\mathsf{U}$. Naturally, if the scheme has a rebuild protocol then we require that it reveals no information about the underlying structure beyond the rebuild leakage $\mathcal{L}_\mathsf{R}$.

If this holds for non-adaptively chosen operations then the scheme is said to be non-adaptively secure. If, on the other hand, the operations can be chosen adaptively, the scheme is said to be adaptively-secure. This notion of security was first formalized by Curtmola et al. in the context of searchable encryption [11] and later generalized to structured encryption in [10].

Definition 4.3 (Adaptive security [11, 10])**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{UToken}, \mathsf{Update}, \mathsf{Rslv})$ *be non-interactive dynamic STE scheme and consider the following probabilistic experiments where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, $\mathcal{L}_\mathsf{S}$, $\mathcal{L}_\mathsf{Q}$ and $\mathcal{L}_\mathsf{U}$ are leakage profiles and $z \in \{0,1\}^*$:*

$\mathbf{Real}_{\Sigma,\mathcal{A}}(k)$*: given z the adversary $\mathcal{A}$ outputs a structure* DS *and receives* EDS *from the challenger, where* $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*. The adversary then* adaptively *chooses a polynomial number of operations* $\mathsf{op}_1, \ldots, \mathsf{op}_m$ *such that* op_i *is either a query q_i or an update u_i. For all $i \in [m]$, the adversary receives* $\mathsf{tk}_i \leftarrow \mathsf{Token}(K, st, q_i)$ *of* $\mathsf{op}_i = q_i$ *or* $\mathsf{utk}_i \leftarrow \mathsf{UToken}(K, st, u_i)$ *if* $\mathsf{op}_i = u_i$*. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.*

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: given z the adversary $\mathcal{A}$ generates a structure* DS *which it sends to the challenger. Given z and leakage* $\mathcal{L}_\mathsf{S}(\mathsf{DS})$ *from the challenger, the simulator $\mathcal{S}$ returns an encrypted structure* EDS *to $\mathcal{A}$. The adversary then* adaptively *chooses a polynomial number of operations* $\mathsf{op}_1, \ldots, \mathsf{op}_m$ *such that* op_i *is either a query q_i or an update u_i. For all $i \in [m]$, the simulator receives either query leakage* $\mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)$ *or update leakage* $\mathcal{L}_\mathsf{U}(\mathsf{DS}, u_i)$*. In the former case, it returns a query token* tk_i *to $\mathcal{A}$ and in the latter it returns an update token* utk_i *to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.*

We say that Σ is adaptively $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U})$*-secure if there exists a* PPT *simulator $\mathcal{S}$ such that for all* PPT *adversaries $\mathcal{A}$, for all $z \in \{0,1\}^*$,*

$$|\Pr[\mathbf{Real}_{\Sigma,\mathcal{A}}(k) = 1] - \Pr[\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k) = 1]| \leq \mathsf{negl}(k).$$

Definition 4.3 can be modified for rebuildable schemes as follows. In the $\mathbf{Real}_{\Sigma,\mathcal{A}}$ experiment, the adversary can also execute the $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}((K, st), \mathsf{EDS})$ protocol with the challenger playing the role of the client. In the $\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}$ experiment, the adversary executes the $\mathsf{Rebuild}$ protocol with the simulator playing the role of the client and receiving leakage $\mathcal{L}_{\mathsf{R}}(\mathsf{DS})$ as input.

Forward privacy. An important property for dynamic STE schemes is *forward privacy* which was introduced in [40] to address some of the limitations of dynamic SSE constructions at the time. The informal requirement described in [40] was that forward-private SSE schemes should not reveal if the file in a file update operation (i.e., a file add or delete) has keywords that were searched for in the past. This was formalized in [4] for $\mathsf{AddFile}$ operations as

$$\mathcal{L}_{\mathsf{AF}}(\mathsf{MM}, f) = \Big(\#f\Big),$$

where $f \subseteq \mathbb{W}$ is a file.

While forward privacy is a stronger notion than what was achieved in previous dynamic SSE schemes [26, 25], as observed by Kamara and Moataz in [24], there are some limitations to this notion. In particular, it does not prevent all correlations between updates and previous searches since the size of the update (i.e., $\#\mathbf{v}$) could itself be correlated with previous searches. As suggested in [24], a stronger notion would be to require leakage-free updates [6] which not only implies forward privacy but also guarantees that no correlations exist between updates and previous searches. Unfortunately, leakage-free updates seem to require expensive padding techniques so we focus here only on forward privacy.

4.2 Security Against a Snapshot Adversary

In the standard notions of security for STE, the adversary is assumed to be the server itself. As such, we seek security guarantees against an adversary that sees not only the encrypted structure but all the search and update operations as well. In many real-world scenarios, however, we are concerned with a weaker adversary that, only periodically, gets access to the encrypted structure and, in particular, does not get to see any query or update operations. This adversarial model captures, for example, data breaches, malicious employees and device theft. Such an adversary is called a *snapshot* adversary.

We propose a new security definition for STE against snapshot adversaries. In our definition 4.4, the adversary has access to multiple snapshots each of which is interspersed with a batch of operations. Intuitively, we require that the encrypted structure reveals no information about the underlying structure and the sequence of operations executed prior to the multiple snapshots, beyond some snapshot leakage $\mathcal{L}_{\mathsf{SN}}$.

Surprisingly, we demonstrate that for a particular class of snapshot leakage, (multiple) snapshot secure STE schemes imply insertion independence, a variant of history independent data structures [35], and can also provide a *write-only oblivious* structure [3].

Definition 4.4 (Snapshot security)**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{UToken}, \mathsf{Update}, \mathsf{Rslv})$ *be a non-interactive dynamic STE scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is*

[6]Note that when we say that a leakage profile is leakage-free we mean that the leakage only includes public information like the security parameter, the size of the query and response spaces, etc.

a stateful adversary, $\mathcal{S}$ is a stateful simulator, $\mathcal{L}_{\mathsf{SN}}$ is a stateful leakage function, $z \in \{0,1\}^*$, and $m \geq 1$:

$\mathbf{Real}^{\mathsf{ms}}_{\Sigma,\mathcal{A}}(k,m)$*:*

1. *given z the adversary $\mathcal{A}$ outputs a structure DS_0;*
2. *the challenger computes $(K, st, \mathsf{EDS}_0) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS}_0)$;*
3. *the adversary $\mathcal{A}(\mathsf{EDS}_0)$ outputs a sequence of operations $\mathbf{op}_1 = (\mathsf{op}_{1,1}, \ldots, \mathsf{op}_{1,\ell})$ where $\ell = \mathsf{poly}(k)$;*
4. *For all $i \in [m]$,*
   (a) the challenger applies all the operations in op_i to EDS_{i-1} by computing and applying the appropriate tokens. This results in EDS_i;
   (b) the adversary $\mathcal{A}(\mathsf{EDS}_i)$ outputs a sequence of operations $\mathbf{op}_{i+1} = (\mathsf{op}_{i+1,1}, \ldots, \mathsf{op}_{i+1,\ell})$ where $\ell = \mathsf{poly}(k)$;
5. *Finally, $\mathcal{A}$ outputs a bit b that is returned by the experiment.*

$\mathbf{Ideal}^{\mathsf{ms}}_{\Sigma,\mathcal{A},\mathcal{S}}(k,m)$*:*

1. *given z the adversary $\mathcal{A}$ outputs a structure DS_0;*
2. *the simulator $\mathcal{S}(z, \mathcal{L}_{\mathsf{SN}}(\mathsf{DS}_0, \perp))$ simulates EDS_0;*
3. *the adversary $\mathcal{A}(\mathsf{EDS}_0)$ outputs a sequence of operations $\mathbf{op}_1 = (\mathsf{op}_{1,1}, \ldots, \mathsf{op}_{1,\ell})$;*
4. *For all $i \in [m]$,*
   (a) the challenger applies all the operations in $\mathbf{op}_i$ to DS_{i-1}, resulting in DS_i;
   (b) the simulator $\mathcal{S}(\mathcal{L}_{\mathsf{SN}}(\mathsf{DS}_i, \mathbf{op}_i))$ simulates EDS_i;
   (c) the adversary $\mathcal{A}(\mathsf{EDS}_i)$ outputs a sequence of operations $\mathbf{op}_{i+1} = (\mathsf{op}_{i+1,1}, \ldots, \mathsf{op}_{i+1,\ell})$;
5. *Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.*

We say that Σ is $(m, \mathcal{L}_{\mathsf{SN}})$-snapshot secure if there exists a PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$ and for all $z \in \{0,1\}^*$,

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

Breach resistance. Ideally, the snapshot leakage of a scheme should be as small as possible. With this in mind, we deem that an encrypted structure should be regarded as *breach-resistant* if its snapshot leakage is at most the size of the plaintext structure at the time the snapshot is taken.

Definition 4.5 (Breach resistance)**.** *Let Σ be an $\mathcal{L}_{\mathsf{SN}}$-snapshot secure non-interactive dynamic STE scheme. We say that Σ is breach-resistant if*

$$\mathcal{L}_{\mathsf{SN}}(\mathsf{DS}, \mathbf{op}_1, \ldots, \mathbf{op}_i) = \#\mathsf{DS}_i,$$

where DS_i is the structure that results from applying $\mathbf{op}_1, \ldots, \mathbf{op}_i$ to DS and $\#\mathsf{DS}_i$ refers to its volume in the sense of the total number of "items" it stores. Note that the volume of a structure depends on its type.

For the remainder of this work, we focus on designing a multi-map encryption scheme that is secure against both persistent and snapshot adversaries. Specifically, we require that the scheme be forward-private against a persistent adversary and breach-resistant against a snapshot adversary. We refer to schemes that meet these two properties as dual-secure.

Definition 4.6 (Dual security)**.** *Let Σ be a dynamic rebuildable structured encryption scheme with leakage profiles $\Lambda_{\mathsf{Per}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{U}}, \mathcal{L}_{\mathsf{R}})$ and $\Lambda_{\mathsf{Sna}} = \mathcal{L}_{\mathsf{SN}}$. We say that Σ is dual-secure if it is forward-private and breach-resistant.*

4.3 Implications of Breach Resistance

In this Section, we show that breach resistance implies some interesting and previously studied notions of security including a weaker notion of history independence [35] and write-only obliviousness [3].

History and insertion independence. A data structure is history independent if its representation does not reveal any information beyond its state. In particular, it hides the order in which items were inserted/removed and the number of operations performed. While history independence is traditionally considered for plaintext data structures, it applies just as well to encrypted structures. Here, we consider a weaker form of history independence which only hides the order of the operations but reveals their number. We call this notion *insertion independence* and provide a formalization in Appendix A.1. In the following Theorem, whose proof is in Appendix B, we show that breach resistance under single-snapshot attacks (i.e., when $m = 1$) implies insertion independence.

Theorem 4.7. *If Σ is $(1, \#\mathsf{DS})$-snapshot secure, then it is insertion independent.*

Write-only obliviousnes. Oblivious RAM (ORAM) [20] is a cryptographic primitive that manages an array in such a way that the client's access pattern is hidden from a persistent adversary. ORAM supports read and a write operations. The most recent constructions achieve a poly-logarithmic multiplicative overhead per operation. Recently, the notion of write-only obliviousness was introduced by Blass, Mayberry, Noubir and Onarlioglu [3] for the purpose of designing deniable file systems, also known as hidden volumes (e.g., TrueCrypt [16]). Write-only obliviousness is a special case of standard obliviousness where the adversary only sees the access pattern produced by write operations (we recall the formal definition in Appendix A.2). Blass et al. show that, unlike standard ORAMs, write-only ORAM solutions can be achieved with only a constant overhead per operation. In the following Theorem, we show breach resistance against a multi-snapshot attack implies write-only obliviousness for encrypted structures that support fixed-size updates; that is, update operations always grow the volume of the encrypted structure by a fixed amount.

Theorem 4.8. *If Σ is $(m, \#\mathsf{DS})$-snapshot secure for $m \geq 1$ and if it has fixed-size updates, then it is write-only oblivious.*

The proof is similar to that of Theorem 4.7 so we defer it to the full version of this work.

5 DLS: A Dual-Secure Multi-Map Encryption Scheme

We now describe our construction, DLS, which is a *dual-secure* rebuildable multi-map encryption scheme. Unlike SPS [40] and Sophos [4], DLS does not make use of ORAM-like techniques or public-key operations. In addition, while Sophos and Diana [5] have query complexity that is linear in the number of delete operations ever executed, DLS's query complexity is linear only in the number of delete operations executed since the last rebuild operation. The rebuild protocol is linear in the size of the multi-map and can be de-amortized while maintaining snapshot security. In the following, we present a high level overview of the construction followed by a more detailed explanation and analysis.

5.1 Overview

DLS is a dictionary-based multi-map encryption scheme that extends the π_{dyn} construction of [8]. It relies on two main ideas: (1) unpredictable labels; and (2) incremental versioning.

The counter-based approach. In [8], Cash et al. introduced a dictionary-based construction called π_{bas} together with dynamic and I/O-efficient variants called π_{dyn} and $\mathsf{2Lev}$, respectively. Many follow-up works have used π_{bas} as a foundation to build new schemes that achieve various guarantees [4, 5, 29, 14]. At a high-level, π_{dyn} works as follows. Given a multi-map MM, the mapping between a label $\ell \in \mathbb{L}_{\mathsf{MM}}$ and its tuple $\mathbf{v}$ is broken down to $\#\mathbf{v}$ pairs where the ith pair has the form $(\ell \| i, \mathbf{v}_i)$.

The encrypted multi-map is a dictionary DX that stores all the label/value pairs encrypted as follows. The key to encrypt a pair is generated as a function of the label itself. The "encrypted" label is the result of applying a pseudo-random function on a counter that is incremented whenever the label is updated (both for deletions and additions). To search for a label, the client sends the key derived from the label and the server generates all encrypted labels by applying a PRF on a counter. Note that all extensions of this approach [4, 5, 29, 14] mostly differ on the way the encrypted labels are generated (e.g., using a trapdoor permutation or a constrained PRF).

Unpredictable labels. This counter-based approach provides the server with the ability (in the form of the keyword-derived key) to compute all future encrypted labels and, therefore, to relate future updates to previous searches. We can address this by preventing the server from computing the PRF evaluations itself and instead having the client directly send the encrypted labels; as done, for example, in the SSE-2 construction of Curtmola et al. [11]. This has two main advantages: (1) it makes the scheme forward-private; and (2) it can be proven secure in the standard model. The drawback, however, is that the query, communication and storage complexities will be in function of the total number of updates ever made. Below, we introduce a technique that addresses this limitation.

Incremental versioning. As mentioned above, the main drawback of counter-based approach is that the query, communication and storage overhead increases linearly in the number of updates (including both additions and deletions). This is the case for most of the recent forward-private constructions as they are based on a *lazy deletion* approach where deleting a pair translates to adding a new pair to the dictionary with a deletion flag. At search time, the server recovers all the pairs for a given label and substracts the deleted pairs. While this may seem unintuitive, lazy deletions makes the design of dual-secure schemes easier because deletions do not induce any modifications to the data structure that are in function of previous search operations. Our goal will consist not only of designing dual-secure schemes but of designing efficient ones. To achieve this we extend the counter-based approach with a version.

Now instead of using a single dictionary, our encrypted multi-map will consist of two dictionaries: an *old* DX_o and a *new* DX_n. After a specific number of update operations which constitute an *epoch*, the old dictionary is merged into the new one. During this merge operation, all the deleted pairs are removed and after the operation the new dictionary becomes then the old, and a new dictionary is initialized. This rebuild process is periodic and is repeated at the end of every epoch. The intuition is that through this rebuild operation, we *clean up* the structure in such a way that the new dictionary mostly consists of the right response for most labels (this depends of course on the number of deletions performed before the rebuilding). To preserve the unpredictability of the labels, we augment the label with a version number

$$(\ell \| i \| \mathsf{version}, \mathbf{v}_i),$$

where $\mathsf{version}$ represents the epoch during which the pair has been added to the dictionary.

It is clear that the rebuild protocol is a key part of our design and that it should offer several properties in order to meet both our security and efficiency requirements. We provide a high level description below.

Almost zero-leakage rebuilds. We require that our rebuild protocol: (1) be almost zero-leakage in the sense that it should not leak "too much" information beyond what has already been leaked before the rebuild operation; and (2) produce a compact structure by removing stale pairs due to lazy deletion. [7] We provide below a gradual explanation of our design starting from a naive rebuild protocol.

One could design a rebuild protocol by simply making the client fetch a pair uniformly at random from the old dictionary and inserting it in the new dictionary with a fresh encryption (under a new version). Unfortunately, while this clearly satisfies our security goals, it falls short of achieving the main purpose of rebuilding which is to generate a compact structure. This is because it does not remove any deleted pairs from the old structure so the new structure remains as large as the old one.

To have a compact structure, the rebuild protocol has to remove the deleted pairs from the old structure. For this, it seems natural to first query every label in order to retrieve all of its pairs, remove the deleted pairs, and finally insert the remaining label/value pairs into the new structure encrypted with a new counter and fresh randomness, respectively. This approach differs from the previous naive construction by fetching all pairs corresponding to a specific label (sampled uniformly at random) instead of fetching one pair at a time. Unfortunately, while this achieves our compactness requirement, it fails at achieving our security requirement. A persistent adversary can learn the response length of unsearched pairs which is more than what it can learn from pre-rebuild operations; clearly violating our almost zero-leakage requirement.

The previous issue can be solved by differentiating between two types of labels: (1) labels that were searched for in the past; (2) and labels that are still unsearched. For the former, we can

[7] We emphasize here that reducing storage also improves the query complexity and token size.

rebuild as above. That is, given a label, retrieve all the corresponding pairs in the old structure, remove the deleted pairs and insert newly encrypted versions into the new dictionary. Clearly, this reduces the storage as the deleted pairs are removed. Also, because the pairs have been searched for in the past, a persistent adversary does not gain any additional knowledge besides the number of deleted values associated with the searched labels. For unsearched labels, the protocol samples a pair uniformly at random and insert it into the new structure. This reveals nothing to a persistent adversary beyond the total number of unsearched pairs.

As long as the intermediate steps of the rebuild protocol are hidden from a snapshot adversary, the scheme will achieve dual-security. Unfortunately, this rebuild protocol has worst-case linear complexity in the size of the old dictionary. We show in Section 6, however, how to de-amortize it which avoids the above worst-case cost while maintaining dual-security. The de-amortized variant also removes the need to keep intermediate steps hidden from a snapshot adversary.

5.2 Detailed Description

DLS makes use of a pseudo-random function F and of a private-key encryption scheme SKE. The details of the scheme are provided in Figs. 1, 2, 3 and 4. At a high-level, it works as follows.

Setup. The Setup algorithm takes as input a security parameter k and a multi-map MM. It makes use of two dictionary data structures, an old dictionary, $\mathsf{DX_o}$, and a new dictionary, $\mathsf{DX_n}$. The old dictionary will be rebuilt and merged into the new one. The rebuild occurs periodically such that after every rebuild epoch, the old dictionary will be totally merged into the new one. The new dictionary becomes the old and a new empty dictionary will be instantiated. We associate to every rebuild epoch a global version, called $\mathsf{version}_g$, which is the number of rebuilds ever performed. Along with this global version, the setup algorithm also instantiates other components of the state: a searched label set $\mathbb{S}_e$, and two state dictionaries, an old $\mathsf{DX_o^{st}}$ and a new $\mathsf{DX_n^{st}}$. The set $\mathbb{S}_e$ is a temporary one that keeps track of all searched labels within a single rebuild epoch. Dictionaries $\mathsf{DX_o^{st}}$ and $\mathsf{DX_n^{st}}$ map a label to both its counter and version. The counter of a label ℓ in $\mathsf{DX_o^{st}}$ and $\mathsf{DX_n^{st}}$ is the number of all values that have been added to $\mathsf{DX_o}$ and $\mathsf{DX_n}$ in the previous and current epoch, respectively. In other words, $\mathsf{DX_o^{st}}$ contains only labels that existed within the previous version, i.e., the previous rebuild epoch.

Setup outputs a dictionary as its encrypted multi-map. At a higher level, in order to store $(\ell \| \mathsf{version}_g, \mathsf{MM}[\ell])$ in the old dictionary $\mathsf{DX_o}$, it stores the pairs $(\ell \| i \| \mathsf{version}_g, v_i)$, for all $v_i \in \mathsf{MM}[\ell]$ and $i \in [\#\mathsf{MM}[\ell]]$, where $\mathsf{version}_g$ is the current rebuild epoch (the rebuild epoch at setup time is initialized to 1). To store the pair in an encrypted way, a PRF evaluation is performed on the concatenation of the label ℓ, its counter count and rebuild epoch $\mathsf{version}_g$. The corresponding value in $\mathsf{MM}[\ell]$ is first concatenated with the string edit^+ and then simply encrypted. The new dictionary $\mathsf{DX_n}$ is only instantiated and remains empty. The output of the setup algorithm includes the encrypted structures $(\mathsf{DX_o}, \mathsf{DX_n})$, the keys as well as the state.

Search token. The Token algorithm takes as input a key, a state and a label. First, it fetches from both the old and new state dictionaries the corresponding counters and rebuilding versions. Recall the old and new counters, count_o and count_n, count the number of times the label has been added, deleted from the old and new dictionaries, respectively. Based on the counters, whether old or new, it creates two sub-token vectors such that $\mathsf{otk} = (\mathsf{otk}_1, \cdots, \mathsf{otk}_{\mathsf{count}_o})$ and $\mathsf{ntk} = (\mathsf{ntk}_1, \cdots, \mathsf{ntk}_{\mathsf{count}_n})$. The old subtokens otk_i, for $i \in [\mathsf{count}_o]$, will allow the server to query the old dictionary $\mathsf{DX_o}$, while the new tokens ntk_i, for $i \in [\mathsf{count}_n]$, will allow the server to query the new dictionary $\mathsf{DX_n}$. Finally, it updates the state by adding ℓ to $\mathbb{S}_e$ as it is now searched. The output includes the state and the token $\mathsf{tk} = (\mathsf{otk}, \mathsf{ntk})$.

Query. The Query algorithm takes as input the token and the encrypted data structure. The token is divided into two sub-token vectors, otk and ntk. Each sub-token in otk corresponds to a label in the old dictionary $\mathsf{DX_o}$ from which the server fetches the value and inserts in the result set, Result. The server performs the same operations for every sub-token in ntk, but on the new dictionary $\mathsf{DX_n}$. The server finally outputs the result set Result. Note that the set Result does not exactly equal $\mathsf{MM}[\ell]$ as the client might have deleted several pairs both in the old or new dictionaries. The client, based on the meta-information included with the decrypted values, i.e., edit^+ or edit^-, can easily compute the correct $\mathsf{MM}[\ell]$.

Update token. The UToken algorithm takes as input a key, a state, and an update consisting of the type of operation op, a label ℓ, and its value $\mathbf{v}$. It first computes tk_1 which is a PRF evaluation on the concatenation of the label, its counter and the current rebuilding version. The counter represents the number of times the label ℓ has been added to the new dictionary $\mathsf{DX_n}$ throughout the current rebuilding version, and is fetched from the new state dictionary $\mathsf{DX_n^{st}}$, given the label ℓ. The counter is then updated accordingly. The value v is concatenated to the operation op before being encrypted, and this represents the second part of the token, tk_2. The output of the algorithm includes $\mathsf{utk} = (\mathsf{tk}_1, \mathsf{tk}_2)$ and an updated state.

Update. The Update algorithm takes as input the update token and the encrypted data structure. The server will simply update the new dictionary $\mathsf{DX_n}$ by adding the update token, $\mathsf{utk} = (\mathsf{tk}_1, \mathsf{tk}_2)$, to it. The output of the Update algorithm consists of the updated encrypted structure.

Rebuilding. The Rebuild algorithm is a two-party protocol between the client and the server. The client's input is a key and a state, while the server's input is the encrypted data structure. The goal of the rebuild operation is to merge the old dictionary into the new one. The new one will then become the old at the end of the rebuild. The rebuild differentiates between two types of labels: (1) labels that have been searched for in $\mathsf{DX_o}$, and (2) labels that have not. For each of the labels in $\mathbb{S}_e$, the client fetches all the values corresponding to $\ell \in \mathbb{S}_e$, removes all values that have to be deleted and then insert the remaining ones into the new dictionary $\mathsf{DX_n}$. Inserting these values into $\mathsf{DX_n}$ follows a similar process to the one in UToken and Update algorithms. For the remaining labels, i.e., labels that have never been searched for in $\mathsf{DX_o}$, the client picks a random label, fetches a value with the largest counter and inserts it into the new dictionary $\mathsf{DX_n}$. The client updates the state by decreasing the counter of the selected label and removes it whenever it equals 1. Once all labels have been reinserted, both the old state dictionary $\mathsf{DX_o^{st}}$ and old dictionary $\mathsf{DX_o}$ are deleted, the set $\mathbb{S}_e$ is reinitialized, the new state dictionary $\mathsf{DX_n^{st}}$ and new dictionary $\mathsf{DX_n}$ becomes the old state dictionary $\mathsf{DX_o^{st}}$ and old dictionary $\mathsf{DX_o}$, and the epoch is incremented. The output of the Rebuild is an updated state for the client and an updated encrypted structure for the server.

In order to achieve snapshot security, the entire transcript of the Rebuild protocol, including its internal state, has to be kept secret. That is, a snapshot adversary must not get a snapshot of the encrypted structures while the Rebuild protocol is executing. This is mainly due to the fact that while rebuilding searched for labels, a snapshot adversary will get to know the response size of the searched for labels; which is clearly at odds with our security goals that consist of only disclosing the size of the data structure. We show how to lift this constraint in section 6.

Efficiency. The query complexity of DLS is

$$O\Big(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e)\Big),$$

where $\mathsf{del}(\ell, e)$ is the number of deletes for ℓ since epoch e when ℓ was most recently searched for (i.e., since the last rebuild during which ℓ was a searched for label). As a point of comparison, the Sophos construction of Bost [4] and the Diana construction of Bost et al. [5] which are dual-secure have query complexity

$$O\Big(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, 0)\Big),$$

where $\mathsf{del}(\ell, 0)$ denotes the number of deletes for ℓ since the structure was setup. The storage complexity of DLS is

$$O\bigg(\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \Big(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e)\Big)\bigg),$$

while recent constructions [4, 5] have storage complexity $O(\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} (\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, 0)))$. DLS has search and update tokens of size $O(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))$ and $O(\#\mathbf{v})$, respectively, and its rebuild complexity is

$$O\bigg(\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \Big(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e) - \mathsf{up}(\ell, c)\Big)\bigg),$$

where $\#\mathsf{up}(\ell, c)$ is the number of updates for ℓ since the last rebuild operation. Finally, the client locally stores the state composed of both the old and new state dictionaries, and the set of searched for labels. That is, client storage is

$$O\bigg(\#\mathbb{L}_{\mathsf{MM}} \cdot \log\Big(\max_{\ell \in \mathbb{L}_{\mathsf{MM}}} \left(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e)\right)\Big)\bigg).$$

Variants. While DLS improves on the query, update and storage complexity of all previous dual-secure dynamic EMM schemes, its token size is larger. This can be improved using one of the following three approaches. The first reduces the token size to be $O(\#\mathsf{up}(\ell, c))$. It consists of using the counter-based approach for the *old* dictionary; that is, granting the server the ability to compute all encrypted labels for the old dictionary (as π_{dyn}). Given that all new updates are going to be added to the new dictionary, the server, with a key derived from the label, can generate all the encrypted labels. That is, the client will only send a key along with the new token ntk which has size equal to the number of updates for ℓ in the current epoch.

The second approach leverages constrained pseudo-random functions, introduced in Diana [5]. The client can simply send two constrained keys for the required ranges for both the old and new counters. This approach reduces the token size to be

$$O\Big(\log\left(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e)\right)\Big),$$

when using GGM [19] as the constrained PRF.

The third approach is the combination of the first two approaches. The client sends a token composed of: (1) a key for a standard PRF with which the server computes all encrypted labels in the old dictionary; and (2) a constrained key for the appropriate range in the new dictionary. The size of the search token in this case is

$$O\bigg(\log\left(\#\mathsf{up}(\ell, c)\right) \bigg),$$

when using GGM as the constrained PRF.

This last variant of DLS outperforms all previous constructions in query complexity, update complexity, token size, query round complexity and client and server storage—all while being secure in the standard model. DLS can also be easily modified to have constant client memory by storing the state on the server in a zero-leakage encrypted multi-map, e.g., using ORAM at the cost of an additive poly-logarithmic overhead per query.

5.3 Security

In the following, we detail the leakage of DLS against standard and snapshot adversaries, respectively.

Against a persistent adversary. The setup leakage of DLS consists of the size of the multi-map MM. The query leakage of DLS for a label ℓ consists of the search, response length and operation patterns. The search pattern captures if and when the label has been searched for in the past. As DLS is response-hiding, it does not reveal the access pattern but only the response length which is the cardinality of the result. The operation pattern reveals if an operation for a label ℓ was an update (i.e., an add or delete) or not. The update leakage of DLS is the size of the tuple to be updated. The rebuild leakage is the size of the updated multi-map and, for each label that was searched for in the current epoch, the number of deletes in $\mathsf{DX_o}$. We now give a precise description of DLS's leakage profile and show that it is adaptively-secure with respect to it. Its setup leakage is

$$\mathcal{L}_{\mathsf{S}}(\mathsf{MM}) = \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell].$$

The query leakage is

$$\mathcal{L}_{\mathsf{Q}}(\mathsf{MM}, \ell) = \bigg(\mathrm{QP}, \mathrm{RL}, \mathrm{OP} \bigg).$$

Here, QP is the query pattern which reveals if and when a query is repeated. More formally, it is defined as $\mathrm{QP} = B$, where B is a binary square matrix of size t and t is the total number of operations that have been made. B is such that $B_{i,j} = 1$ if the ith and jth queries are the same, and 0 otherwise. The response length pattern is

$$\mathrm{RL}(\mathsf{MM}, \ell) = \#\mathsf{MM}[\ell].$$

The operation pattern is

$$\mathrm{OP}(\mathsf{MM}, \ell) = \mathbf{m},$$

where $\mathbf{m}$ is a binary vector of size t, where t is the total number of operations. For all $i \in [t]$, $m_i = 1$ if the ith operation is an update and $m_i = 0$ otherwise. Its update leakage is

$$\mathcal{L}_{\mathsf{U}}(\mathsf{MM}, (\mathsf{op}, \ell, \mathbf{v})) = \#\mathbf{v},$$

Let F be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme. Consider the dynamic encrypted multi-map DLS $= (\mathsf{Setup}, \mathsf{Token}, \mathsf{UToken}, \mathsf{Get}, \mathsf{Put}, \mathsf{Rebuild})$ defined as follows:

- $\mathsf{Setup}\big(1^k, \mathsf{MM}\big)$:
  1. sample $K_1, K_2 \xleftarrow{\$} \{0,1\}^k$;
  2. initialize an empty set $\mathbb{S}_e$ and four empty dictionaries $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}$, $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}$, and DX_{o} and DX_{n} with capacities $\#\mathbb{L}_{\mathsf{MM}}$ and $\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$ respectively;
  3. for all $i \in [\#\mathsf{MM}]$,
     (a) sample a pair $(\ell, \mathsf{MM}[\ell])$ from MM without replacement;
     (b) set $\mathsf{count} = 1$ and $\mathsf{version} = 1$;
     (c) for all $v \in \mathsf{MM}[\ell]$,
        i. compute
        $$\mathsf{label} := F_{K_1}(\ell \| \mathsf{version} \| \mathsf{count}) \quad \text{and} \quad \mathsf{value} := \mathsf{Enc}(K_2, v \| \mathsf{edit}^{+});$$
        ii. set $\mathsf{DX}_{\mathsf{o}}[\mathsf{label}] := \mathsf{value}$;
        iii. set $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell] := (\mathsf{version}, \mathsf{count})$;
        iv. increment count;
  4. increment $\mathsf{version}_g$;
  5. output (K, st, EMM) where $K = (K_1, K_2)$, $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}})$ and $\mathsf{EMM} = (\mathsf{DX}_{\mathsf{o}}, \mathsf{DX}_{\mathsf{n}})$.
- $\mathsf{Token}\big(K, st, \ell\big)$:
  1. parse $K = (K_1, K_2)$ and $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}})$;
  2. if $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] \neq \perp$, set $(\mathsf{version}_n, \mathsf{count}_n) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell]$, and if $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell] \neq \perp$ set $(\mathsf{version}_o, \mathsf{count}_o) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell]$ and add ℓ to $\mathbb{S}_e$;
  3. set
     $$\mathsf{otk} = \big(\mathsf{otk}_1, \cdots, \mathsf{otk}_{\mathsf{count}_o}\big) \quad \text{and} \quad \mathsf{ntk} = \big(\mathsf{ntk}_1, \cdots, \mathsf{ntk}_{\mathsf{count}_n}\big),$$
     where
     $$\mathsf{otk}_i := F_{K_1}(\ell \| \mathsf{version}_o \| i) \quad \text{and} \quad \mathsf{ntk}_i := F_{K_1}(\ell \| \mathsf{version}_n \| j),$$
     for all $i \in [\mathsf{count}_o]$ and $j \in [\mathsf{count}_n]$;
  4. output the update state $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}})$ and the token $\mathsf{tk} = (\mathsf{otk}, \mathsf{ntk})$.
- $\mathsf{Get}\big(\mathsf{tk}, \mathsf{EMM}\big)$:
  1. parse $\mathsf{tk} = (\mathsf{otk}, \mathsf{ntk})$ where
     $$\mathsf{otk} = \big(\mathsf{otk}_1, \cdots, \mathsf{otk}_{\mathsf{count}_o}\big) \quad \text{and} \quad \mathsf{ntk} = \big(\mathsf{ntk}_1, \cdots, \mathsf{ntk}_{\mathsf{count}_n}\big),$$
     and $\mathsf{EMM} = (\mathsf{DX}_{\mathsf{o}}, \mathsf{DX}_{\mathsf{n}})$;
  2. instantiate an empty set Result;
  3. add $\mathsf{DX}_{\mathsf{o}}[\mathsf{otk}_i]$ and $\mathsf{DX}_{\mathsf{n}}[\mathsf{ntk}_j]$ for all $i \in [\mathsf{count}_o]$ and $j \in [\mathsf{count}_n]$ to Result;
  4. output Result.

Figure 1: DLS (Part 1).

```
• $\mathsf{UToken}\big(K, st, (\mathsf{op}, \ell, \mathbf{v})\big)$:
    1. parse $K = (K_1, K_2)$ and $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}_\mathsf{o}^\mathsf{st}, \mathsf{DX}_\mathsf{n}^\mathsf{st})$;
    2. if $\ell \notin \mathbb{L}_\mathsf{MM}$,
        (a) set $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] := (\mathsf{version}_g, \mathsf{count})$ where $\mathsf{count} = 1$;
    3. otherwise,
        (a) if $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] \neq \perp$,
            i. set $(\mathsf{version}, \mathsf{count}) := \mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell]$;
            ii. increment $\mathsf{count}$;
            iii. set $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] := (\mathsf{version}_g, \mathsf{count})$;
        (b) otherwise
            i. set $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] := (\mathsf{version}_g, \mathsf{count})$ where $\mathsf{count} = 1$;
    4. for all $v \in \mathbf{v}$,
        (a) compute
                $$\mathsf{tk}_{v,1} := F_{K_1}\big(\ell \| \mathsf{version}_g \| \mathsf{count}\big) \quad \text{and} \quad \mathsf{tk}_{v,2} := \mathsf{Enc}(K_2, v \| \mathsf{op});$$
        (b) increment $\mathsf{count}$ and set $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] := (\mathsf{version}_g, \mathsf{count})$;
    5. output the updated state $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}_\mathsf{o}^\mathsf{st}, \mathsf{DX}_\mathsf{n}^\mathsf{st})$ and $\mathsf{utk} = (\mathsf{tk}_{v,1}, \mathsf{tk}_{v,2})_{v \in \mathbf{v}}$.
• $\mathsf{Put}\big(\mathsf{utk}, \mathsf{EMM}\big)$:
    1. parse $\mathsf{EMM} = (\mathsf{DX}_\mathsf{o}, \mathsf{DX}_\mathsf{n})$ and $\mathsf{utk} = (\mathsf{tk}_{v,1}, \mathsf{tk}_{v,2})_{v \in \mathbf{v}}$;
    2. for all $v \in \mathbf{v}$, set $\mathsf{DX}_\mathsf{n}[\mathsf{tk}_{v,1}] := \mathsf{tk}_{v,2}$;
    3. output $\mathsf{EMM}$.
```

Figure 2: DLS (Part 2).

```
• $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\Big(\big(K, st\big), \mathsf{EMM}\Big)$:
    1. $\mathbf{C}$ parses $K = (K_1, K_2)$, $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}_\mathsf{o}^\mathsf{st}, \mathsf{DX}_\mathsf{n}^\mathsf{st})$ and $\mathsf{EMM} = (\mathsf{DX}_\mathsf{o}, \mathsf{DX}_\mathsf{n})$;
    2. for all $\ell \in \mathbb{S}_e$ such that $\mathsf{DX}_\mathsf{o}^\mathsf{st}[\ell] \neq \perp$, ,
        (a) $\mathbf{C}$ sets $(\mathsf{version}_o, \mathsf{count}_o) := \mathsf{DX}_\mathsf{o}^\mathsf{st}[\ell]$ and removes $\ell$ from $\mathsf{DX}_\mathsf{o}^\mathsf{st}$;
        (b) $\mathbf{C}$ computes and sends to $\mathbf{S}$,
                $\mathsf{tk} = (\mathsf{otk}_1, \cdots, \mathsf{otk}_{\mathsf{count}_o})$,
            where $\mathsf{otk}_i = F_{K_1}(\ell \| \mathsf{version}_o \| i)$;
        (c) $\mathbf{S}$ computes and sends to $\mathbf{C}$,
                $\mathsf{Result} = (\mathrm{ct}_1, \cdots, \mathrm{ct}_{\mathsf{count}_o})$;
            where $\mathrm{ct}_i = \mathsf{DX}_\mathsf{o}[\mathsf{otk}_i]$ for all $i \in [\mathsf{count}_o]$;
        (d) $\mathbf{C}$ computes $V = \mathsf{Result}^+ \setminus \mathsf{Result}^-$, where
                $\mathsf{Result}^+ = \{v : \forall \mathrm{ct} \in \mathsf{Result},\ v \| \mathsf{edit}^+ = \mathsf{Dec}\big(K_2, \mathrm{ct}\big)\}$
                $\mathsf{Result}^- = \{v : \forall \mathrm{ct} \in \mathsf{Result},\ v \| \mathsf{edit}^- = \mathsf{Dec}\big(K_2, \mathrm{ct}\big)\}$
        (e) if $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] \neq \perp$, $\mathbf{C}$ sets $(\mathsf{version}_n, \mathsf{count}_n) := \mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell]$, otherwise sets $\mathsf{count}_n = 1$;
        (f) for all $v \in V$,
            i. $\mathbf{C}$ computes and sends,
                $\mathsf{tk}_1 := F_{K_1}\big(\ell \| \mathsf{version}_g \| \mathsf{count}_n\big)$    $\mathsf{tk}_2 := \mathsf{Enc}(K_2, v \| \mathsf{edit}^+)$;
            ii. $\mathbf{S}$ computes $\mathsf{DX}_\mathsf{n}[\mathsf{tk}_1] := \mathsf{tk}_2$;
            iii. $\mathbf{C}$ increments $\mathsf{count}_n$;
```

Figure 3: DLS (Part 3).

```
• $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\Big((K, st), \mathsf{EMM}\Big)$:

    3. while $\#\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}} > 0$,
        (a) $\mathbf{C}$ picks $\ell$ at random
        (b) $\mathbf{C}$ sets $(\mathsf{version}_o, \mathsf{count}_o) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell]$ and $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell] := (\mathsf{version}_o, \mathsf{count}_o - 1)$;
        (c) if $\mathsf{count}_o - 1 < 1$, $\mathbf{C}$ then removes $\ell$ from $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}$;
        (d) $\mathbf{C}$ computes and sends to S $\mathsf{otk} = F(K_1, \ell \| \mathsf{version}_{\mathsf{o}} \| \mathsf{count}_{\mathsf{o}})$;
        (e) $\mathbf{S}$ computes and sends to $\mathbf{C}$ $\mathrm{ct} = \mathsf{DX}_{\mathsf{o}}[\mathsf{otk}]$;
        (f) if $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] \neq \perp$,
            i. $\mathbf{C}$ sets $(\mathsf{version}_n, \mathsf{count}_n) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell]$;
            ii. $\mathbf{C}$ computes and sends to $\mathbf{S}$,

                $\mathsf{tk}_1 := F_{K_1}\big(\ell \| \mathsf{version}_n \| \mathsf{count}_n\big) \qquad \mathsf{tk}_2 := \mathsf{Enc}\big(K_2, \mathsf{Dec}(K_2, \mathrm{ct})\big)$;

            iii. $\mathbf{S}$ computes $\mathsf{DX}_{\mathsf{n}}[\mathsf{tk}_1] := \mathsf{tk}_2$;
            iv. $\mathbf{C}$ sets $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] := (\mathsf{version}_n, \mathsf{count}_n + 1)$;
        (g) otherwise if $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] = \perp$, then
            i. $\mathbf{C}$ sets $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] := (\mathsf{version}_g, 1)$;
            ii. $\mathbf{C}$ computes and sends to $\mathbf{S}$,

                $\mathsf{tk}_1 := F_{K_1}\big(\ell \| \mathsf{version}_g \| 1\big) \qquad \mathsf{tk}_2 := \mathsf{Enc}\big(K_2, \mathsf{Dec}(K_2, \mathrm{ct})\big)$

            iii. $\mathbf{S}$ computes $\mathsf{DX}_{\mathsf{n}}[\mathsf{tk}_1] := \mathsf{tk}_2$;
    4. $\mathbf{C}$ sets $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}} := \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}$ and initializes an empty dictionary $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}$ with capacity $2 \cdot \#\mathbb{L}_{\mathsf{MM}}$;
    5. $\mathbf{S}$ sets $\mathsf{DX}_{\mathsf{o}} := \mathsf{DX}_{\mathsf{n}}$ and initializes an empty dictionary $\mathsf{DX}_{\mathsf{n}}$ with capacity $2 \cdot \#\mathbb{L}_{\mathsf{MM}}$;
    6. $\mathbf{C}$ empties $\mathbb{S}_e$ and increments $\mathsf{version}_g$;
    7. $\mathbf{C}$ outputs the updated states $st = (\mathsf{version}_g, \mathbb{S}_e, \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}})$ and $\mathbf{S}$ the updated encrypted
       multi-map $\mathsf{EMM} = (\mathsf{DX}_{\mathsf{o}}, \mathsf{DX}_{\mathsf{n}})$.
```

Figure 4: DLS (Part 4).

for all $\mathsf{op} \in \{\mathsf{edit}^+, \mathsf{edit}^-\}$. Its rebuild leakage is

$$\mathcal{L}_{\mathsf{R}}(\mathsf{MM}) = \left(\#\mathsf{del}^o_\ell\right)_{\ell \in \mathbb{S}_e},$$

where $\#\mathsf{del}^o_\ell$ is number of pairs with label ℓ removed from the old dictionary $\mathsf{DX_o}$ and $\mathbb{S}_e \subseteq \mathbb{L}$ is the set of labels that were searched for in the current epoch.

Theorem 5.1. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* DLS *is* $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U}, \mathcal{L}_\mathsf{R})$ *secure.*

The proof of Theorem 5.1 is deferred to Appendix C.

Leakage against a snapshot adversary. The snapshot leakage $\mathcal{L}_{\mathsf{SN}}$ of DLS is

$$\mathcal{L}_{\mathsf{SN}}(\mathsf{MM}, \mathsf{op}_1, \ldots, \mathsf{op}_i) = \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}_i[\ell]$$

where MM_i is the current version of the multi-map. Note that, given $\mathcal{L}_{\mathsf{SN}}$, DLS is breach-resistant based on Definition 4.5.

Theorem 5.2. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* DLS *is* $(m, \mathcal{L}_{\mathsf{SN}})$*-snapshot secure, for* $m \in \mathsf{poly}(k)$*.*

The proof of Theorem 5.2 is in Appendix E. Note that this Theorem also implies that DLS is insertion-independent and write-only oblivious (when the client updates the structure with fixed-sized tuples).

Corollary 5.3. *Given Theorems 5.1 and 5.2,* DLS *is dual-secure.*

6 DLS$^\mathsf{d}$: DLS with De-amortized Rebuilding

In Section 5, we introduced an instantiation of the rebuilding protocol with a worst-case complexity linear in the size of the old dictionary, i.e., in

$$O\Big(\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \Big(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e) - \mathsf{up}(\ell, c)\Big)\Big),$$

where $\#\mathsf{up}(\ell, c)$ is the number of updates for ℓ since the last rebuild operation. Moreover, DLS only achieves dual security under the assumption that all intermediate steps of the rebuild protocol are hidden from the snapshot adversary. In this section, we show how to overcome these two challenges, resulting in a variant of our construction called DLS$^\mathsf{d}$. DLS$^\mathsf{d}$ uses a de-amortized rebuild process, that maintains dual security with minimal leakage while providing optimal worst-case efficiency under *no assumptions* on when snapshots occur.

Overview. $\mathrm{DLS}^{\mathsf{d}}$ is a variant of DLS with de-amortized rebuilding, i.e., it has the same setup, search token and get algorithms, but differs on how the rebuilding process is performed. We now provide a gradual description of this process starting from our previous rebuild protocol.

We can de-amortize the $\mathsf{Rebuild}$ protocol of DLS by simply operating on one label at a time in the case of a searched label, or on a pair in the other case. This solution provides a de-amortized rebuilding and clearly does not introduce any additional leakage against a persistent adversary. Unfortunately, this solution still induces additional leakage against a snapshot adversary because it can differentiate between the two types of labels based on the number of pairs that have been inserted into the new dictionary. Remember that our goal is to have snapshot leakage composed only of the size of the current structure. So far, this is not the case as we have an additional leakage.

We solve the issue above by allowing some client storage. That is, instead of inserting an arbitrary number of pairs in the case of searched for labels, only λ pairs are inserted and the remaining pairs are kept in the client side. For the unsearched for labels, the same number of pairs, λ, is similarly inserted in the structure. This will make differentiating between the two types of labels impossible for a snapshot adversary.

De-amortization. An important design decision we have to make is to figure out when to rebuild. There are three possible approaches, including:

- **at update time**: this approach runs the de-amortized rebuild steps as a sub-routine of the update protocol. That is, whenever the client updates the encrypted structure, the client simultaneously performs a partial rebuilding that operates on λ pairs. Note that this choice is natural as a snapshot adversary already knows that a data structure has been modified due to the update operation. Therefore performing a partial-rebuilding that is triggered by an update will not leak more than the number of rebuilt pairs, which is λ.

- **at query time**: this approach consists of running the de-amortized rebuild step as a sub-routine of the query protocol. In this case, it is easy to see that a snapshot adversary would learn that a search occurred by just looking to the encrypted structure, which violates breach-resistance and, therefore, dual security.

- **continuously**: this approach runs the de-amortized rebuild steps continuously in the background. This is quite similar to the previous rebuild protocol except that the client can always query or update the structure independently of the rebuilding process. In other words, the rebuild does not affect the correctness of the query operation. This approach provides the client with the flexibility to rebuild the data structure whenever it is required (which is not the case of the previous approaches).

Our preferred approach is to execute rebuild steps at update time. In the following, we provide the algorithmic details. Note that the syntax of the STE scheme will change slightly because we are now dealing with a self-adjusting EMM. We refer the reader to Definition 4.1 for more details.

Details. $\mathrm{DLS}^{\mathsf{d}}$ is a self-adjusting dynamic encrypted multi-map composed of four non-interactive algorithms and one interactive protocol such that $\mathrm{DLS}^{\mathsf{d}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get}, \mathsf{Update}, \mathsf{Rslv})$. Below, we only detail the Update protocol as Setup, Token, Get and Rslv remain the same as in DLS. For more details of the scheme we refer the readers to Figs. 5 and 6.

Update is a two-party protocol between the client and the server. The client's input consists of a key K, a state st and an update u. The server's input consists of the encrypted multi-map EMM. The state is the same as the state of DLS except that it is augmented with the de-amortization rate λ and two lists $\mathsf{L_{sr}}$ and $\mathsf{L_{un}}$. $\mathsf{L_{sr}}$ is a stash that stores the pairs corresponding to the searched for labels, while $\mathsf{L_{un}}$ is a stash that stores the pairs for the unsearched for labels.

The client starts by creating the update token and sending it to the server which then inserts the updates in the new dictionary using DLS UToken and Put algorithms, respectively. The client then samples a bit b. If $b = 0$, the client rebuilds a searched label in $\mathsf{L_{sr}}$, otherwise it rebuilds an unsearched label in $\mathsf{L_{un}}$. If $\#\mathsf{L_{sr}} < \lambda$ or $\#\mathsf{L_{un}} < \lambda$, it means the client does not hold enough pairs in the stashes to rebuild and needs to prepare additional pairs as follows.

If it is rebuilding searched labels and $\#\mathsf{L_{sr}} < \lambda$, then the client picks a label $\ell \in \mathbb{S}_e$, where $\mathbb{S}_e$ contains all the searched for labels. The client fetches the corresponding pairs from the old dictionary $\mathsf{DX_o}$, decrypts them, removes all the deleted pairs, and appends the remaining pairs to $\mathsf{L_{sr}}$. The client then removes the label ℓ from $\mathbb{S}_e$. When it is time to send pairs to the server, it generates λ freshly encrypted pairs from $\mathsf{L_{sr}}$ to send. Similarly to the UToken algorithm, a fresh pair is composed of two sub-tokens. The first, tk_1, is the evaluation of the pseudo-random function on the label ℓ, the global version $\mathsf{version}_g$, and a counter count_n. The second, tk_2, is generated by encrypting the value v corresponding to the label ℓ concatenated to the string edit^+.

Otherwise, if it is rebuilding unsearched labels and $\#\mathsf{L_{un}} < \lambda$, the client retrieves an unsearched label uniformly at random from $\mathbb{L}_{\mathsf{MM}} \setminus \mathbb{S}_e$ and only retrieves one pair using the label's counter. The client then decrements the counter and updates the state accordingly. If the counter is less than one, it removes the label from the state, $\mathsf{DX_o^{st}}$. The client refreshes the pair similarly to the operations described earlier, but then appends it to $\mathsf{L_{un}}$. The client repeats this process as long as $\#\mathsf{L_{un}} < \lambda$. Finally if both the stashes are empty and each pair in $\mathsf{DX_o}$ is refreshed, the server deletes the old dictionary and the new dictionary becomes the old dictionary. The client similarly deletes the state of the old dictionary and replaces it with the state of the new dictionary. A new empty dictionary and state are initialized for future updates. Finally the client increments the global version.

Efficiency. $\mathrm{DLS}^{\mathsf{d}}$ has the same query complexity, storage complexity and token size as DLS. Below, we only detail the client memory and update complexity. $\mathrm{DLS}^{\mathsf{d}}$ introduces two new data structures that are stored at the client. The first, $\mathsf{L_{sr}}$, stores the pairs for the searched labels and has size

$$O\bigg(\max\bigg\{\lambda, \max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))\bigg\}\bigg).$$

The second, $\mathsf{L_{un}}$, stores the pairs for the unsearched labels and has size $O(\lambda)$. In addition, similarly to DLS, the client also stores the state which is composed of both the old and new state dictionaries, and the set of searched for labels. This results in $O(\#\mathbb{L}_{\mathsf{MM}} \cdot \log\big(\max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))\big))$ storage at the client. The overall client storage is then

$$O\bigg(\#\mathbb{L}_{\mathsf{MM}} \cdot \log\bigg(\max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))\bigg) + \max\bigg\{\lambda, \max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))\bigg\}\bigg).$$

The update complexity is $O(\lambda + \#\mathbf{v})$ with two rounds of interactions.[8] The first round fetches the pairs and sends the update token to the server and the second inserts the freshly encrypted pairs

[8]One might think that the update complexity should be equal to $O(\#\mathbf{v} + \max\{\lambda, \max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell] + \mathsf{del}(\ell, e))\})$. However, the quantity $\max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell])$ can be de-amortized over $\lambda^{-1} \cdot \max_{\ell\in\mathbb{L}_{\mathsf{MM}}}(\#\mathsf{MM}[\ell])$ updates, as the client

from the states along with the updates.

6.1 Security

In the following, we describe the leakage of $\mathrm{DLS}^{\mathsf{d}}$ against persistent and snapshot adversaries, respectively.

Leakage against a persistent adversary. The setup and query leakage of $\mathrm{DLS}^{\mathsf{d}}$ is the same as DLS. The Update leakage consists of the update leakage of DLS and the de-amortization rate which is public. Note that, contrary to DLS, there is no explicit rebuild leakage since the rebuild process is de-amortized and executed as a sub-routine of the update operation.

The update with rebuild leakage is then

$$\mathcal{L}_{\mathsf{U_r}}(\mathsf{MM}, u) = \Big(\mathcal{L}_{\mathsf{U}}(\mathsf{MM}, u), \mathcal{L}_{\mathsf{R_d}}(\mathsf{MM})\Big),$$

where $u = (\mathsf{op}, \ell, \mathbf{v})$, $\mathcal{L}_{\mathsf{U}}(\mathsf{MM}, u) = \#\mathbf{v}$, $\mathcal{L}_{\mathsf{R_d}}(\mathsf{MM}) = \big(\lambda, (\#\mathsf{del}^{o}_{\ell})_{\ell \in \mathbb{S}_e}\big)$, λ is the rebuild rate of the Update protocol, and $\mathbb{S}_e \subseteq \mathbb{L}$ is the set of labels that were searched for in the current epoch.

Theorem 6.1. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* $\mathrm{DLS}^{\mathsf{d}}$ *is* $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{U_r}})$ *secure.*

The proof of Theorem 6.1 appears in Appendix D.

Leakage against a snapshot adversary. The snapshot leakage $\mathcal{L}_{\mathsf{SN}}$ of $\mathrm{DLS}^{\mathsf{d}}$ is

$$\mathcal{L}_{\mathsf{SN}}(\mathsf{MM}, \mathbf{op}) = \Big(\lambda, \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}(\ell)\Big).$$

Note that λ is a public parameter so $\mathrm{DLS}^{\mathsf{d}}$ is breach-resistant.

Theorem 6.2. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* $\mathrm{DLS}^{\mathsf{d}}$ *is* $(m, \mathcal{L}_{\mathsf{SN}})$*-snapshot secure, for* $m \in \mathsf{poly}(k)$.

The proof of Theorem 6.2 is deferred to Appendix F. Note that this Theorem implies that $\mathrm{DLS}^{\mathsf{d}}$ is insertion independent and write-only oblivious (when the client updates the structure with fixed-size tuples).

Corollary 6.3. $\mathrm{DLS}^{\mathsf{d}}$ *is dual-secure.*

7 Empirical Evaluation

We now evaluate how our construction performs in practice. We implemented $\mathrm{DLS}^{\mathsf{d}}$, the de-amortized variant of DLS, in Java using the Clusion encrypted search library [34]. It consists of 1114 lines of codes excluding 180 lines for testing purposes calculated using CLOC [12]. We set λ to 3 for all experiments except for one where we vary λ to study its effect on update time.

will not fetch any pair as long as the state $\mathsf{L_{sr}}$ contains more than λ pairs.

Let F be a pseudo-random function, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme. Let $\mathrm{DLS} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{UToken}, \mathsf{Get}, \mathsf{Put}, \mathsf{Rebuild})$ be the dynamic encrypted multi-map described in Section 5. Consider the dynamic encrypted multi-map $\mathrm{DLS}^{\mathsf{d}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get}, \mathsf{Update})$ where Setup, Token, Get and Update are defined as follows:

- $\mathsf{Setup}\big(1^k, \mathsf{MM}\big)$: $(K, st, \mathsf{EMM}) \leftarrow \mathrm{DLS}.\mathsf{Setup}\big(1^k, \mathsf{MM}\big)$.
- $\mathsf{Token}\big(K, st, \ell\big)$: $(st, \mathsf{tk}) \leftarrow \mathrm{DLS}.\mathsf{Token}\big(K, st, \ell\big)$.
- $\mathsf{Get}\big(\mathsf{tk}, \mathsf{EMM}\big)$: $\mathsf{Result} \leftarrow \mathrm{DLS}.\mathsf{Get}\big(\mathsf{tk}, \mathsf{EMM}\big)$
- $\mathsf{Update}_{\mathbf{C},\mathbf{S}}\Big(\Big(K, st, \big(\mathsf{op}, \ell, \mathbf{v}\big)\Big), \mathsf{EMM}\Big)$:

```
1. $\mathbf{C}$ sends $\mathsf{utk} \leftarrow \mathrm{DLS}.\mathsf{UToken}\big(K, st, (\mathsf{op}, \ell, \mathbf{v})\big)$ to $\mathbf{S}$;
2. $\mathbf{S}$ executes $\mathrm{DLS}.\mathsf{Put}\big(\mathsf{utk}, \mathsf{EMM}\big)$;
3. $\mathbf{C}$ parses $K = (K_1, K_2)$, $st = (\mathsf{version}_g, \mathbb{S}_e, \lambda, \mathsf{L}_{\mathsf{sr}}, \mathsf{L}_{\mathsf{un}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}, \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}})$, and $\mathsf{EMM} = (\mathsf{DX}_{\mathsf{o}}, \mathsf{DX}_{\mathsf{n}})$;
4. $\mathbf{C}$ samples a bit $b$
5. if $b = 0$
   (a) if $\#\mathsf{L}_{\mathsf{sr}} < \lambda$ then,
       i. $\mathbf{C}$ picks at random without replacement $\ell \in \mathbb{S}_e$ such that $\mathsf{ODX}[\ell] \neq \bot$,
       ii. $\mathbf{C}$ sets $(\mathsf{version}_o, \mathsf{count}_o) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}[\ell]$ and removes $\ell$ from $\mathsf{DX}^{\mathsf{st}}_{\mathsf{o}}$;
       iii. $\mathbf{C}$ computes and sends to $\mathbf{S}$,
                $\mathsf{tk} = (\mathsf{otk}_1, \cdots, \mathsf{otk}_{\mathsf{count}_o})$,
            where $\mathsf{otk}_i = F(K_1, \ell \| \mathsf{version}_o \| i)$ for all $i \in [\mathsf{count}_o]$;
       iv. $\mathbf{S}$ computes and sends to $\mathbf{C}$,
                $\mathsf{Result} = (\mathrm{ct}_1, \cdots, \mathrm{ct}_{\mathsf{count}_o})$,
            where $\mathrm{ct}_i = \mathsf{DX}_{\mathsf{o}}[\mathsf{otk}_i]$ for all $i \in [\mathsf{count}_o]$;
       v. $\mathbf{C}$ computes $V = \mathsf{Result}^+ \setminus \mathsf{Result}^-$, where
                $\mathsf{Result}^+ = \{v : \forall \mathrm{ct} \in \mathsf{Result},\ v \| \mathsf{edit}^+ = \mathsf{Dec}\big(K_2, \mathrm{ct}\big)\}$
                $\mathsf{Result}^- = \{v : \forall \mathrm{ct} \in \mathsf{Result},\ v \| \mathsf{edit}^- = \mathsf{Dec}\big(K_2, \mathrm{ct}\big)\}$
       vi. for all $v \in V$, $\mathbf{C}$ adds $(\ell, v)$ to $\mathsf{L}_{\mathsf{sr}}$;
       vii. If still $\#\mathsf{L}_{\mathsf{sr}} < \lambda$, $\mathbf{C}$ picks at random another $\ell \in \mathbb{S}_e$ and repeats these steps
   (b) else, for $i \in [\lambda]$,
       i. $\mathbf{C}$ picks and removes a pair $(\ell, v)$ from $\mathsf{L}_{\mathsf{sr}}$
       ii. if $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] \neq \bot$, $\mathbf{C}$ sets $(\mathsf{version}_n, \mathsf{count}_n) := \mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell]$, otherwise sets $\mathsf{count}_n = 1$;
       iii. $\mathbf{C}$ computes
                $\mathsf{tk}_1 := F\big(K_1, \ell \| \mathsf{version}_g \| \mathsf{count}_n\big)$        $\mathsf{tk}_2 := \mathsf{Enc}(K_2, v \| \mathsf{edit}^+)$;
       iv. $\mathbf{C}$ increments $\mathsf{count}_n$;
       v. $\mathbf{S}$ sets $\mathsf{DX}_{\mathsf{n}}[\mathsf{tk}_1] := \mathsf{tk}_2$
       vi. $\mathbf{C}$ sets $\mathsf{DX}^{\mathsf{st}}_{\mathsf{n}}[\ell] := (\mathsf{version}_n, \mathsf{count}_n)$
```

Figure 5: $\mathrm{DLS}^{\mathsf{d}}$ (Part 1).

```
• $\mathsf{Update}_{\mathbf{C},\mathbf{S}}\Big(\big(K, st, (\mathsf{op}, \ell, \mathbf{v})\big), \mathsf{EMM}\Big)$:

    6. if $b = 1$, then
        (a) $\mathbf{C}$ sets $\mathsf{count} = \lambda$;
        (b) $\mathbf{C}$ picks at random without replacement $\ell \in \mathbb{L}_{\mathsf{MM}} \setminus \mathbb{S}_e$ such that $\mathsf{DX}_\mathsf{o}^\mathsf{st}[\ell] \neq \perp$
        (c) while $\mathsf{count} > 0$,
            i. $\mathbf{C}$ sets $(\mathsf{version}_o, \mathsf{count}_o) := \mathsf{DX}_\mathsf{o}^\mathsf{st}[\ell]$ and $\mathsf{DX}_\mathsf{o}^\mathsf{st}[\ell] := (\mathsf{version}_o, \mathsf{count}_o - 1)$;
            ii. if $\mathsf{count}_o - 1 < 1$, then $\mathbf{C}$ removes $\ell$ from $\mathsf{DX}_\mathsf{o}^\mathsf{st}$;
            iii. $\mathbf{C}$ computes and sends to $\mathbf{S}$ $\mathsf{otk} = F(K_1, \ell \| \mathsf{version}_o \| \mathsf{count}_o)$;
            iv. $\mathbf{S}$ computes and sends to $\mathbf{C}$ $\mathrm{ct} = \mathsf{DX}_\mathsf{o}[\mathsf{otk}]$;
            v. if $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] \neq \perp$, $\mathbf{C}$ sets $(\mathsf{version}_n, \mathsf{count}_n) := \mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell]$, otherwise sets $\mathsf{count}_n = 1$;
            vi. $\mathbf{C}$ computes,
                    $\mathsf{tk}_1 := F\big(K_1, \ell \| \mathsf{version}_n \| \mathsf{count}_n\big) \qquad \mathsf{tk}_2 := \mathsf{Enc}\big(K_2, \mathsf{Dec}(K_2, \mathrm{ct})\big)$;
            vii. $\mathbf{C}$ adds $(\mathsf{tk}_1, \mathsf{tk}_2)$ to $\mathsf{L}_\mathsf{un}$;
            viii. $\mathbf{C}$ sets $\mathsf{DX}_\mathsf{n}^\mathsf{st}[\ell] := (\mathsf{version}_n, \mathsf{count}_n + 1)$;
            ix. $\mathbf{C}$ decrements $\mathsf{count}$;
        (d) for $i \in [\lambda]$,
            i. $\mathbf{S}$ sets $\mathsf{DX}_\mathsf{n}[\mathsf{tk}_{i,1}] := \mathsf{tk}_{i,2}$, where $(\mathsf{tk}_{i,1}, \mathsf{tk}_{i,2}) \in \mathsf{L}_\mathsf{un}$;
            ii. $\mathbf{C}$ removes $(\mathsf{tk}_{i,1}, \mathsf{tk}_{i,2})$ from $\mathsf{L}_\mathsf{un}$;
    7. if $\#\mathsf{DX}_\mathsf{o} = 0$, then
        (a) $\mathbf{C}$ sets $\mathsf{DX}_\mathsf{o}^\mathsf{st} := \mathsf{DX}_\mathsf{n}^\mathsf{st}$ and initializes an empty dictionary $\mathsf{DX}_\mathsf{n}^\mathsf{st}$ with capacity $2 \cdot \#\mathbb{L}_{\mathsf{MM}}$;
        (b) $\mathbf{S}$ sets $\mathsf{DX}_\mathsf{o} := \mathsf{DX}_\mathsf{n}$ and initializes an empty dictionary $\mathsf{DX}_\mathsf{n}$ with capacity $2 \cdot \#\mathbb{L}_{\mathsf{MM}}$;
        (c) $\mathbf{C}$ initializes an empty set $\mathbb{S}_e$, increments $\mathsf{version}_g$.
    8. $\mathbf{C}$ outputs the updated state $st = (\mathsf{version}_g, \mathbb{S}_e, \lambda, \mathsf{L}_\mathsf{sr}, \mathsf{L}_\mathsf{un}, \mathsf{DX}_\mathsf{o}^\mathsf{st}, \mathsf{DX}_\mathsf{n}^\mathsf{st})$, and $\mathbf{S}$ outputs the
       updated encrypted multi-map $\mathsf{EMM} = (\mathsf{DX}_\mathsf{o}, \mathsf{DX}_\mathsf{n})$.
```

Figure 6: DLS^d (Part 2).

Parsing and indexing. We used the parsing and indexing functionality of the Clusion library to process data (which is itself based on the Lucene parser [31]). Through Clusion, our implementation handles `pdf` files, Microsoft Office files (`doc`, `docx`, `pptx`, `xlsx`), basic text files. For media files, it only indexes the file names.

Cryptographic primitives. We use the cryptographic primitives provided by Clusion (themselves based on Bouncy Castle [9]). For symmetric encryption, we use AES in CTR mode with a 256 bit key. We use of HMAC-SHA256/512 for PRFs and random oracles.

Experimental setup. We ran our experiments on an Amazon EC2 instance running Ubuntu Linux (c3.8xlarge) with an Intel Xeon E5-2680 v2 (Ivy Bridge) Processor with 32 vCPU and 60 GB of RAM. For all our experiments, we used the Wikipedia data dumps. The total uncompressed size of our dataset was 26.5GB. There are a total number of $2,681,795$ files in this dataset. We partitioned these files into different folders. The first folder had $17,600$ files with a total size of about 250 MB and the last folder had $554,059$ files with a total size of 3.6GB. In our empirical evaluation, we want to quantify the following characteristics of $\mathrm{DLS}^{\mathsf{d}}$:

1. The time to set up the EMM as a function of the number of pairs;
2. The size of each EMM and of the client state as a function of the number of pairs;
3. The time taken to respond to a query for labels with different selectivity as a function of the number of pairs. The selectivity of a label is the number of values associated to it;
4. The time taken for an update operation as a function of the number of pairs in the EMM. We also measure how different rebuild parameters λ affect the time taken;
5. The effect of de-amortized rebuilding on the time taken by a query operation in $\mathrm{DLS}^{\mathsf{d}}$ specifically when there are deletes involved.

Setup time and storage overhead. Fig. 7 describes the time taken to set up an EMM as a function of the number of label/value pairs stored. We created EMMs with number of pairs ranging from $2,758,254$ to $83,239,341$. The setup phase takes under 13 minutes with a multi-threaded implementation. Fig. 8 shows the size on disk of both the client state and the EMM for different number of pairs. We observe that even for about 83 million pairs, the client state is only 210MB for an 11GB EMM.

Search and update operations. Fig. 9 describes the time taken to search for labels of different selectivities ($\mathsf{MM}(w)$). We also measure time taken by an update operation. In our experiments, the client and server are running on the same machine. We measure the effect of increasing the EMM size on the query time, which seems negligible. We do not send any update tokens between the query operations during the experiment. For each EMM we first search for labels of different selectivities (100, 1000 and 10,000). The search time for all selectivities is less than 1 microsecond per pair. We ran every search data point corresponding to every number of pairs in the x-axis 500 times. We re-ran the whole experiment 10 times, then we took the median. From Fig. 10 we can see that the update operation is more costly as it takes around 100 milliseconds when λ is set to 3. This can be attributed to the fact that the rebuild is performed with the update algorithm.

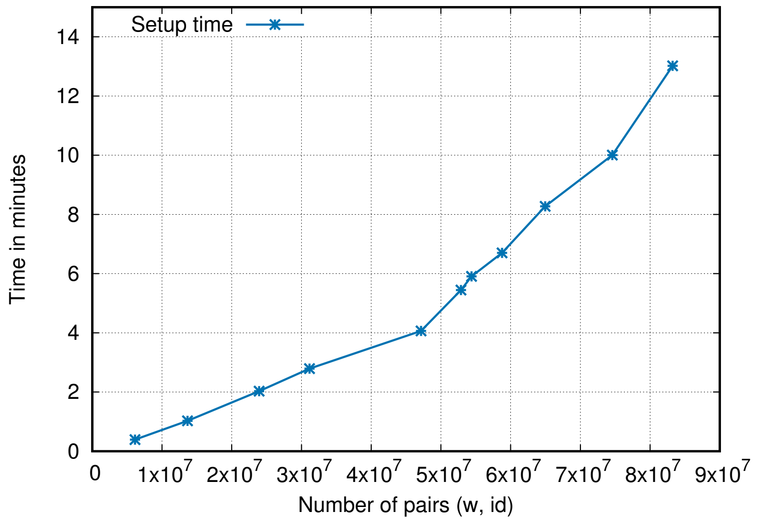

Figure 7: Setup time.

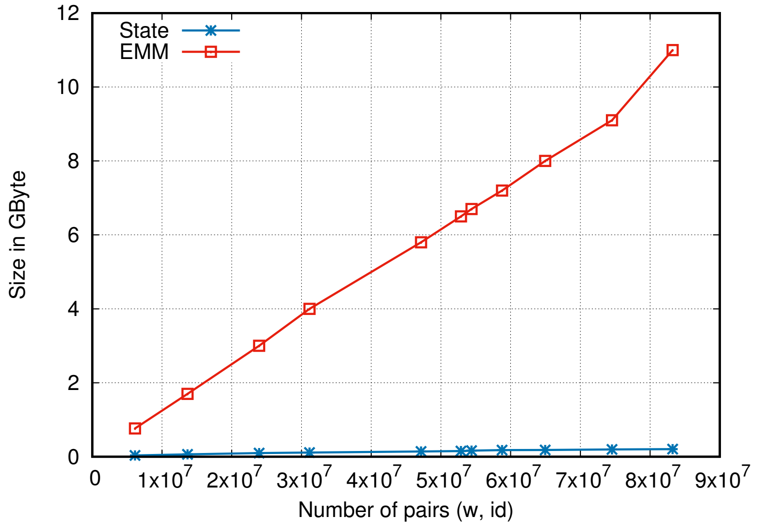

Figure 8: Encrypted multi-map and state sizes.

Whenever we do an update, we also execute λ de-amortized rebuild steps. One can see that as we increase λ, the update time increases proportionally. Note also that both the search and update operations are independent of the number of pairs in the EMM.

Effect of rebuilding. Figure 11 describes the query time as a function of the number of delete operations. The keyword being queried initially has selectivity $100,000$ which was achieved by updating the EMM with $100,000$ update tokens. The consequence of this is that these values are initially in DX_n. The first point of epoch 0 (the green line) represents the first query. After that, we send 8000 delete tokens, execute a search, send 8000 more deletes, execute a search and so on.

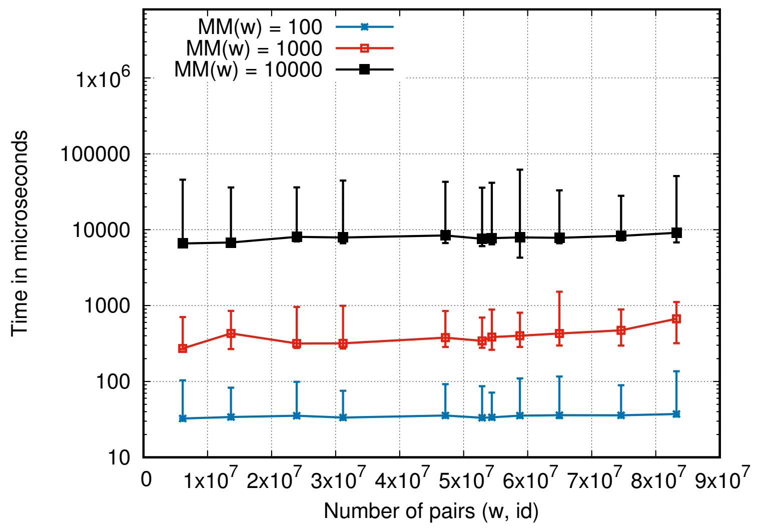

Figure 9: Search time for 100, 1000 and 10000 search selectivities.

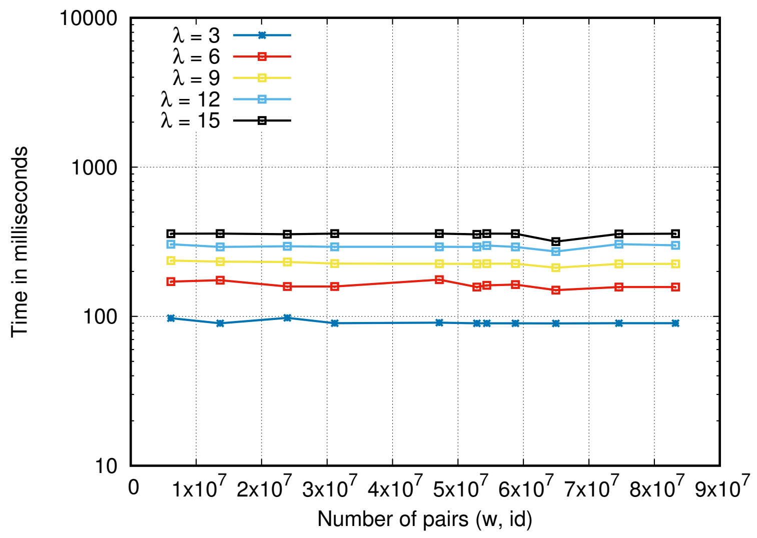

Figure 10: Update time for different rebuiling parameters.

This continues until we send a total of $40,000$ delete tokens (the end of the green line). At this stage, the selectivity of the keyword is $60,000$ but the total number of pairs in $\mathsf{DX_n}$ associated with the keyword is $140,000$. During epoch 0 we can clearly see that query time increases as the number of deletes increases.

At some point, the rebuild operation ends and epoch 1 begins and $\mathsf{DX_n}$ becomes $\mathsf{DX_o}$. At the beginning of epoch 1 two queries occur before the keyword has been selected to be rebuilt. The gap in query time between epoch 0 and the first two queries of epoch 1 is due to the fact that the pairs are stored in different sized dictionaries in the two epochs. In epoch 0 they are stored in a relatively empty $\mathsf{DX_n}$ whereas in epoch 1 they are stored in $\mathsf{DX_o}$ along with 50 million other pairs

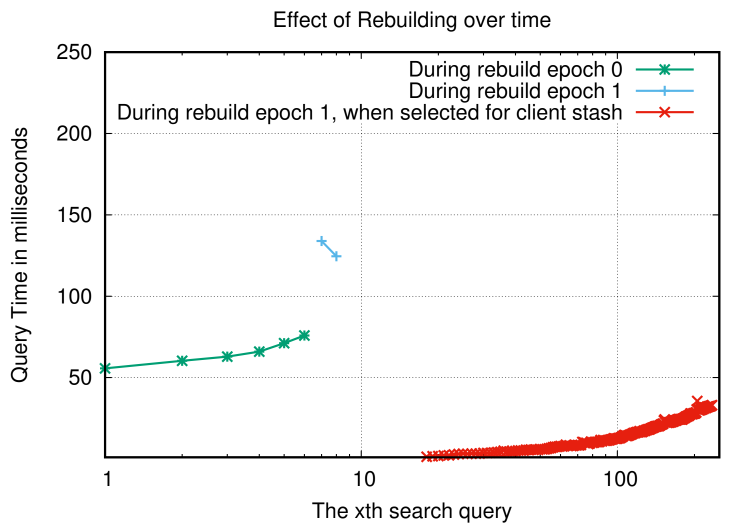

Figure 11: Query times for the same keyword at different times. During epoch 0, the keyword starts with a selectivity of 100, 000 and 8000 delete operations are performed between each query operation. No further deletes happen in the next epoch.

(remember that we need to keep updating the EMM to make the rebuild process progress). Once the keyword is selected to be rebuilt (red line), it is moved to the client's local stash; specifically, to L_{sr} since it was searched. At this stage, query time is negligible but increases as the pairs are inserted into DX_{n}. When all pairs have been inserted, DX_{n} only holds 60, 000 pairs associated to this keyword (i.e., all the delete pairs have been removed).

One can clearly see the difference in query time before and after the keyword has been rebuilt. Before it has been rebuilt (i.e., epoch 0 and beginning of epoch 1 which are the green and blue lines, respectively), the query times range from 55.6 ms to 75.8 ms. After being rebuilt (i.e., the end of epoch 1 which is the red line), query times range from 0.1 ms to 33 ms.

Also, note that all query times stay well below 1 microsecond per pair.

Comparison with previous constructions. Given the asymptotic overhead of the SPS construction, we do not compare it to $\mathrm{DLS}^{\mathsf{d}}$ and focus mainly on the Sophos and Diana schemes of [4] and [5]. The empirical evaluations of these constructions, however, are based on C/C++ implementations whereas our implementation of $\mathrm{DLS}^{\mathsf{d}}$ is in Java. This naturally makes a quantitative comparison very difficult. In addition, the Sophos implementation is multi-threaded (whereas our $\mathrm{DLS}^{\mathsf{d}}$ implementation is only multi-threaded for setup) and is evaluated on disk. Under these conditions, [4] reports search times ranging from 24 microseconds to 7 microseconds per pair depending on the selectivity of the keyword. On similar datasets, the Diana implementation in [5] is 10 times faster in part because it uses hardware-accelerated AES instructions. Similarly to our $\mathrm{DLS}^{\mathsf{d}}$ implementation, the hardware accelerated C/C++ implementation of Diana takes less than 1 microsecond per pair.

Another point of comparison we can make is with respect to delete operations. While the query time of Sophos and Diana will increase with the total number of deletes ever performed, the query time of $\mathrm{DLS}^{\mathsf{d}}$ will only increase with the number of deletes since the last rebuild (or the current epoch). As shown above, this makes a significant difference in the query time of $\mathsf{DLS^d}$.

References

[1] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *ACM Symposium on Theory of Computing (STOC '16)*, STOC '16, pages 1101–1114, New York, NY, USA, 2016. ACM.

[2] Adam J. Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S. Roche. Oblivisync: Practical oblivious file backup and synchronization. In *Network and Distributed System Security Symposium (NDSS '16)*, 2016.

[3] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward robust hidden volumes using write-only oblivious RAM. In *ACM Conference on Computer and Communications Security (CCS '14)*, pages 203–214, 2014.

[4] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[5] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM Conference on Computer and Communications Security (CCS '17)*, 2017.

[6] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[7] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[8] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[9] Bouncy Castle. Crypto API. In *http://www.bouncycastle.org*.

[10] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[12] Al Danial. Cloc. In *http://www.cloc.sourceforge.net*.

[13] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *ACM International Conference on Management of Data (SIGMOD '17)*, SIGMOD '17, pages 1053–1067, New York, NY, USA, 2017. ACM.

[14] Mohammad Etemad, Alptekin KÃijpÃğÃij, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *CoRR*, abs/1710.00208, 2017.

[15] B. A Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[16] TrueCrypt Foundation. Truecrypt. In *http://truecrypt.sourceforge.net/*.

[17] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016*, pages 563–592, 2016.

[18] E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.

[19] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. In *IEEE Symposium on the Foundations of Computer Science (FOCS '84)*, pages 464–479. IEEE Computer Society, 1984.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[21] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 162–168, New York, NY, USA, 2017. ACM.

[22] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *ACM Conference on Computer and Communications Security (CCS '14)*, CCS '14, pages 310–320, New York, NY, USA, 2014. ACM.

[23] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS '13)*, pages 875–888, 2013.

[24] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.

[25] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[26] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[27] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. *IACR Cryptology ePrint Archive*, 2016:453, 2016.

[28] K. Kurosawa and Y. Ohtaki. How to update documents verifiably in searchable symmetric encryption. In *International Conference on Cryptology and Network Security (CANS '13)*, pages 309–328, 2013.

[29] Russell W. F. Lai and Sherman S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings*, pages 478–497, 2017.

[30] K. Lewi and D. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[31] Lucene. Parser. In *http://lucene.apache.org*.

[32] X. Meng, S. Kamara, K. Nissim, and G. Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *ACM Conference on Computer and Communications Security (CCS 15)*, 2015.

[33] I. Miers and P. Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. Cryptology ePrint Archive, Report 2016/830, 2016. http://eprint.iacr.org/2016/830.

[34] T. Moataz. Clusion. https://github.com/encryptedsystems/Clusion.

[35] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501, New York, NY, USA, 2001. ACM.

[36] M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[37] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[38] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[39] R. Poddar, T. Boelter, and R. Ada Popa. Arx: A Strongly Encrypted Database System. Technical Report 2016/591.

[40] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[41] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, 2016.

A Insertion Independence and Write-Only Obliviousness

A.1 Insertion Independence

Definition A.1 (Insertion independence)**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{UToken}, \mathsf{Update})$ *be a dynamic STE scheme with self-adjusting queries and consider the following probabilistic experiment between a stateful adversary* $\mathcal{A}$ *and a challenger:*

$\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k)$*:*

1. $\mathcal{A}$ *outputs a data structure* DS_0 *and two sequences of* m *operations*

$$\mathbf{op}_0 = (\mathsf{op}_{0,1}, \cdots, \mathsf{op}_{0,m}) \quad \textit{and} \quad \mathbf{op}_1 = (\mathsf{op}_{1,1}, \cdots, \mathsf{op}_{1,m}),$$

   where $\mathsf{op}_{0,i}, \mathsf{op}_{1,i} \in \{q_i, u_i\}$ *and* $\#\mathsf{DS}_{0,m} = \#\mathsf{DS}_{1,m}$ *where* $\#\mathsf{DS}_{j,m}$ *is the result of applying all operations of the* j*th sequence on* DS_0*;*

2. *the challenger computes* $(K, st, \mathsf{EDS}_0) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS}_0)$ *and samples a bit* $b \overset{\$}{\leftarrow} \{0,1\}$*;*
3. *for all* $i \in [m]$*,*
   (a) *if* $\mathsf{op}_{b,i} = q_{b,i}$*, the challenger*
      i. *computes* $\mathsf{qtk}_{b,i} \leftarrow \mathsf{Token}(K, st, q_{b,i})$*;*
      ii. *computes* $(r, \mathsf{EDS}_{b,i}) \leftarrow \mathsf{Query}(\mathsf{EDS}_{b,i-1}, \mathsf{qtk}_{b,i})$*;*
   (b) *if* $\mathsf{op}_{b,i} = u_{b,i}$*,*
      i. *computes* $\mathsf{utk}_{b,i} \leftarrow \mathsf{UToken}(K, st, u_{b,i})$*;*
      ii. *computes* $\mathsf{EDS}_{b,i} \leftarrow \mathsf{Update}(\mathsf{EDS}_{b,i-1}, \mathsf{utk}_{b,i})$*;*
4. *the adversary* $\mathcal{A}(\mathsf{EDS}_{b,m})$ *output* b'*;*
5. *if* $b' = b$*, the experiment outputs* 1 *and* 0 *otherwise.*

We say that Σ *is insertion independent if for all* PPT *adversaries* $\mathcal{A}$*,*

$$\Pr\left[\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k).$$

A.2 Write-Only Obliviousness

In the following, we introduce a game based security definition for *write-only oblivious* structured encryption scheme.

Definition A.2 (Write-only obliviousness)**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{UToken}, \mathsf{Query}, \mathsf{Update})$ *be a STE and consider the following probabilistic experiments where* $\mathcal{A}$ *is a stateful adversary:*

$\mathbf{Exp}^{\mathsf{wo}}_{\Sigma,\mathcal{A}}(k)$*:*

1. $\mathcal{A}$ *outputs a data structure* DS_0 *and two sequences of* m *update operations*

$$\mathbf{op}_0 = (u_{0,1}, \cdots, u_{0,m}) \quad \textit{and} \quad \mathbf{op}_1 = (u_{1,1}, \cdots, u_{1,m});$$

2. *output* EDS_0 *where* $(K, st, \mathsf{EDS}_0) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS}_0)$*;*

3. *sample a bit* $b \overset{\$}{\leftarrow} \{0,1\}$*;*
4. *for all* $i \in [m]$*,*
   - *output* $\mathsf{utk}_{b,i} \leftarrow \mathsf{UToken}(K, st, u_{b,i})$*;*
   - *compute* $(\perp, \mathsf{EDS}_{b,i}) \leftarrow \mathsf{Update}(\mathsf{utk}_{b,i}, \mathsf{EDS}_{b,i-1})$*;*
5. $\mathcal{A}$ *output* b'*;*
6. *if* $b' = b$*, the experiment outputs* 1 *and* 0 *otherwise.*

We say that Σ *is* write-only oblivious *if for all* PPT *adversaries* $\mathcal{A}$*,*

$$\Pr\left[\mathbf{Exp}^{\mathsf{wo}}_{\Sigma,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k).$$

B Proof of Theorem 4.7

Theorem 4.7. *If* Σ *is* $(1, \#\mathsf{DS})$*-snapshot secure, then it is insertion independent.*

Proof. Assume that there exists a PPT adversary $\mathcal{A}$ for which Definition A.1 does not hold. We write,

$$\varepsilon(k) = \Pr\left[\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k) = 1\right] - \frac{1}{2}$$

to denote a non-negligible probability in k. We use $\mathcal{A}$ to build a PPT adversary $\mathcal{B}$ for all PPT simulator $\mathcal{S}$, such that $\mathcal{B}$ is able to distinguish between $\mathbf{Real}^{\mathsf{ss}}_{\Sigma,\mathcal{B}}(k)$ and $\mathbf{Ideal}^{\mathsf{ss}}_{\Sigma,\mathcal{B},\mathcal{S}}(k)$ experiments with probability $\varepsilon(k)$.

$\mathcal{B}$ starts by computing $(\mathsf{DS}_0, \mathbf{op}_0, \mathbf{op}_1) \leftarrow \mathcal{A}(1^k)$ where $\mathbf{op}_0$ and $\mathbf{op}_1$ contain the same number of update operations and have the same length m. $\mathcal{B}$ outputs DS_0. It then samples a random bit b uniformly at random and outputs $\mathbf{op}_b$. Upon receiving $\mathsf{EDS}_{b,m}$, which is either the output of the $\mathbf{Real}^{\mathsf{ss}}_{\Sigma,\mathcal{B}}(k)$ or the $\mathbf{Ideal}^{\mathsf{ss}}_{\Sigma,\mathcal{B},\mathcal{S}}(k)$ experiment, $\mathcal{B}$ computes $b' \leftarrow \mathcal{A}(\mathsf{EDS}_{b,m})$ and outputs 1 if $b' = b$ and 0 otherwise. Note that $\mathcal{B}$ is PPT since $\mathcal{A}$ is and that $\mathcal{B}$ will output 1 if and only if $\mathcal{A}$ succeeds in guessing the bit b.

First, consider the case where $\mathsf{EDS}_{b,m}$ is the output of the $\mathbf{Real}^{\mathsf{ss}}_{\Sigma,\mathcal{B}}(k)$ experiment. In this case, the view of $\mathcal{A}$ (while being simulated by $\mathcal{B}$) is exactly the same as it would be in the $\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k)$ experiment. That is,

$$\begin{aligned}\Pr\left[\mathbf{Real}^{\mathsf{ss}}_{\Sigma,\mathcal{B}}(k) = 1\right] &= \Pr\left[\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k) = 1\right] \\ &= \varepsilon(k) + \frac{1}{2}.\end{aligned}$$

Second, consider the case when $\mathsf{EDS}_{b,m}$ is the output of the $\mathbf{Ideal}^{\mathsf{ss}}_{\Sigma,\mathcal{B},\mathcal{S}}(k)$ experiment. The simulator $\mathcal{S}$ takes as input z and $\mathcal{L}_{\mathsf{SN}}(\mathsf{DS}_{b,m}, \mathbf{op}_b) = \#\mathsf{DS}$ which is independent of b. Note, in particular, that $\#\mathsf{DS}_{b,m}$ is independent of b because $\mathbf{op}_0$ and $\mathbf{op}_1$ lead to two data structures that have the same volume. It follows that the simulated structure EDS produced by the simulator is also independent of b which in turn implies that the best $\mathcal{A}(\mathsf{EDS})$ can do in guessing b is to choose uniformly at random. We therefore have,

$$\begin{aligned}\Pr\left[\mathbf{Ideal}^{\mathsf{ss}}_{\Sigma,\mathcal{B},\mathcal{S}}(k) = 1\right] &= \Pr\left[\mathbf{Exp}^{\mathsf{ii}}_{\Sigma,\mathcal{A}}(k) = 1\right] \\ &= \frac{1}{2}.\end{aligned}$$

Combining both equalities, we obtain,

$$\Pr\left[\mathbf{Real}^{\mathsf{ss}}_{\Sigma,\mathcal{B}}(k)=1\right]-\Pr\left[\mathbf{Ideal}^{\mathsf{ss}}_{\Sigma,\mathcal{B},\mathcal{S}}(k)=1\right]=\varepsilon(k).$$

This concludes our proof.

■

C Proof of Theorem 5.1

Theorem 5.1. *If* SKE *is an RCPA-secure encryption scheme and F is a pseudo-random function, then* DLS *is* $(\mathcal{L}_{\mathsf{S}},\mathcal{L}_{\mathsf{Q}},\mathcal{L}_{\mathsf{U}},\mathcal{L}_{\mathsf{R}})$ *secure.*

Proof. Consider the simulator that works as follows.

1. **Setup simulation.** Given the setup leakage $\mathcal{L}_{\mathsf{S}}(\mathsf{MM})=\left(\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}}\#\mathsf{MM}[\ell]\right)$, the simulator initializes two dictionaries DX_0 and DX_1 of size $\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}}\#\mathsf{MM}[\ell]$. It then initializes a *state multi-map* MM_0 and three vectors vec_l, vec_o vec_n. For $1\leq i\leq\sum_{\ell\in\mathbb{L}_{\mathsf{MM}}}\#\mathsf{MM}[\ell]$:

   (a) compute $str_i, ctr_i \overset{\$}{\leftarrow}\{0,1\}^k$ where str_i, ctr_i are of appropriate lengths;

   (b) set $\mathsf{DX}_0[str_i]=ctr_i$;

   (c) add str_i to vec_l.

   It then outputs $\mathsf{EMM}=(\mathsf{DX}_0,\mathsf{DX}_1)$.

2. UToken **simulation.** Given the update leakage

$$\mathcal{L}_{\mathsf{U}}(\mathsf{MM},\ell,\mathbf{v},\mathsf{op})=\#\mathbf{v},$$

   for all $\mathsf{op}\in\{\mathsf{edit}^+,\mathsf{edit}^-\}$. For $1\leq i\leq\#\mathbf{v}$,

   (a) compute $str_i, value_i \overset{\$}{\leftarrow}\{0,1\}^k$ where $str_i, value_i$ are of appropriate lengths;

   (b) add str_i to vec_n.

   It then outputs $\mathsf{UToken}=(str_i, value_i)_{i\in\bar{\mathbf{v}}}$.

3. Token **simulation.** Given the query leakage

$$\mathcal{L}_{\mathsf{Q}}(\mathsf{MM},\ell)=\Big(\mathrm{QP},\mathrm{RL},\mathrm{OP}\Big),$$

   where QP is the search pattern, $\mathrm{RL}=(\#\mathsf{DX}_{\mathsf{o}}[\ell],\#\mathsf{DX}_{\mathsf{n}}[\ell])$ is the access pattern and OP is the operation pattern. It adds ℓ to a set of searched labels $\mathbb{S}$ and executes the following steps.

   (a) if $\mathsf{MM}_0[\ell]=\perp$ (which occurs if ℓ is being searched for the first time),

      i. set $count=\#\mathsf{DX}_{\mathsf{o}}[\ell]$ from RL and initialize an empty set S_l;

      ii. using OP, extract and remove all labels that were updates of ℓ from vec_o and add them to S_l. This step is performed in a case if where a rebuild has already occurred and some updates are in the old dictionary.

iii. from $i = \#S_l$ to *count*, pick and remove a label from vec_l and add it to S_l.

iv. assign S_l to $\mathsf{MM}_0[\ell]$

(b) else if $\mathsf{MM}_0[\ell] < \#\mathsf{DX_o}[\ell]$ (which happens when there is a rebuild and some new updates are now in the old dictionary),

- using OP, extract and remove all labels that were updates of ℓ from vec_o and prepend them to $\mathsf{MM}_0[\ell]$.

(c) set $count = \#\mathsf{DX_n}[\ell]$ from RL and initialize an empty set S_l;

(d) Using OP, add all labels that were updates of ℓ from vec_n (all updates in the new dictionary) and add them to S_l.

It then outputs $\mathsf{otk} = \mathsf{MM}_0[\ell]$ and $\mathsf{ntk} = S_l$.

4. $\mathsf{Rebuild}$ **simulation.** Given the rebuild leakage

$$\mathcal{L}_\mathsf{R}(\mathsf{MM}) = (\#\mathsf{del}^o_\ell)_{\ell\in\mathbb{S}},$$

where $\#\mathsf{del}^o_\ell$ is the number of pairs with label ℓ that have been removed from the old dictionary since the last rebuild. It executes the following steps,

(a) to remove the appropriate number of tuples it does the following. For $\ell \in \mathbb{S}$,

i. if $\mathsf{MM}_0[\ell] \neq \perp$,

A. initialize v_{lab};

B. for $count = 0$ to $\#\mathsf{MM}_0[\ell] - \#del^o_\ell$;

- compute $str, ctr \xleftarrow{\$} \{0,1\}^k$ where str, ctr are of appropriate lengths;
- set $v_{lab}[count] = str$ and $\mathsf{DX}_1[str] = ctr$;
- send the pair (str, ctr) to adversary.

C. set $\mathsf{MM}_0[\ell] := v_{lab}$.

(b) To freshly encrypt the remaining pairs n the old dictionary, it sets *counter* to $\#vec_l + \#vec_o + \sum_{\ell\in\mathbb{K}(\mathsf{MM}_0)\setminus\mathbb{S}} \#\mathsf{MM}_0[\ell]$ where vec_l has all the still unsearched for labels from setup, vec_o has the unsearched for updates which happened before the last rebuild and $\mathbb{K}(\mathsf{MM}_0) \setminus \mathbb{S}$ is a set of all ℓ that were searched for before the last rebuild but not since (where $\mathbb{K}(\mathsf{MM}_0)$ is a set that contains all the labels of the state multi-map MM_0). It then does the following:

(c) initialize new_l of size *counter*;

(d) for $i = 1$ to *counter*,

i. $str, ctr \xleftarrow{\$} \{0,1\}^k$ where str, ctr are of appropriate lengths;

ii. append (str, ctr) to new_l;

iii. send (str, ctr) to adversary.

Now that the simulator has all fresh encryptions, it just needs to replace them in its internal state.

(e) for all $\ell \in (\mathbb{K}(\mathsf{MM}_0) \setminus \mathbb{S})$,

i. initialize v_{lab} and for $count = 0$ to $\#\mathsf{MM}_0[\ell]$;

```
            A. pick and remove a pair $(str, ctr)$ from $new_l$ at random;
            B. append $str$ to $v_{lab}$;
            C. set $\mathsf{DX}_1[str] = ctr$.
         ii. set $\mathsf{MM}_0[\ell] := v_{lab}$.
   (f) initialize $v_{lab}$ and for $\ell \in vec_l$,
         i. pick and remove a pair $(str, ctr)$ from $new_l$ at random;
         ii. append $str$ to $v_{lab}$;
         iii. set $\mathsf{DX}_1[str] = ctr$.
   (g) $vec_l := v_{lab}$;
   (h) initialize $v_{lab}$ and for $\ell \in vec_o$,
         i. pick and remove a pair $(str, ctr)$ from $new_l$ at random;
         ii. append $str$ to $v_{lab}$;
         iii. set $\mathsf{DX}_1[str] = ctr$;
   (i) set $vec_o := v_{lab}$;
   (j) prepend $vec_n$ to $vec_0$ and delete everything from $vec_n$;
   (k) set $\mathsf{DX}_0 := \mathsf{DX}_1$ and delete everything from $\mathsf{DX}_1$;
   (l) delete all labels from $\mathbb{S}$.
```

Now, we have to show that for PPT adversaries $\mathcal{A}$, the output of the real experiment and ideal experiment are indistinguishable. This can be shown by standard sequence of games argument that shows that EMM, utk and tk are indistinguishable from the real ones due to the RCPA security of SKE and the pseudo-randomness of F.

Game_0: It is the same as a real experiment.

Game_1: It is the same as Game_0 except that we replace the function F by a call to a random function G. This is indistinguishable because of the pseudo-randomness of F.

Game_2: It is the same as Game_1 except that we do not generate any keys and replace encryption steps to simply producing a random string. During rebuild step for searched for labels, we simple remove encrypted values at random from the result set till the result set is of the leaked size. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.

Game_3: It is the same as Game_2 except that we replace the random function G and use random strings for labels. In setup, keep track of all labels (random strings) generated. When generating search tokens, we pick these labels at random for a fresh query as the number of labels originally in the old dictionary for the query is leaked. We further pick appropriate labels from both dictionaries as the total number of updates in each dictionary and when they were received is leaked. We assign these labels to this particular query for future repetitions. During rebuild step for unsearched for labels, we pick them one by one at random and then replace them with a new random string. This is the same as Game_2 as the output of G and a random string are indistinguishable.

Game_3 is the same as an ideal experiment.

This concludes our proof.

■

D Proof of Theorem 6.1

Theorem D.1. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* DLS^d *is* $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U})$ *secure.*

Consider the simulator that works as follows.

1. **Setup simulation.** The simulator takes as input the setup leakage $\mathcal{L}_\mathsf{S}(\mathsf{MM}) = \sum_{\ell \in \mathbb{L}_\mathsf{MM}} \#\mathsf{MM}[\ell]$ and initializes dictionaries DX_0 and DX_1 of size $\sum_{\ell \in \mathbb{L}_\mathsf{MM}} \#\mathsf{MM}[\ell]$. It also initializes a state multi-map MM_0 and three vectors vec_l, vec_o vec_n. Then For $1 \leq i \leq \sum_{\ell \in \mathbb{L}_\mathsf{MM}} \#\mathsf{MM}[\ell]$:

   (a) compute $str_i, ctr_i \xleftarrow{\$} \{0,1\}^k$ where str_i, ctr_i are of appropriate lengths;

   (b) set $\mathsf{DX}_0[str_i] = ctr_i$;

   (c) add str_i to vec_l.

   It then outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$

2. **Update Simulation.** The simulator first simulates an update token utk and then outputs the rebuilt label, value pairs for the adversary. To simulate the utk, it takes as input the update leakage which is equal to $\mathcal{L}_\mathsf{U}(\mathsf{MM}, \ell, \mathbf{v}, \mathsf{op}) = \#\mathbf{v}$ for all $\mathsf{op} \in \{\mathsf{edit}^+, \mathsf{edit}^-\}$. To output the rebuilt pairs, it takes as input the public parameter λ and also when a rebuild gets completed. Then, for $1 \leq v \leq \#\mathbf{v}$, It does the following:

   (a) $str_v, value_v \xleftarrow{\$} \{0,1\}^k$ where $str_v, value_v$ are of appropriate lengths;

   (b) add str_v to vec_n.

   It outputs $\mathsf{utk} = (str_v, value_v)_{v \in \bar{\mathbf{v}}}$, and executes the following steps if the rebuild was completed:

   (a) delete vec_l and MM_0;

   (b) set $vec_o := vec_n$ and reset vec_n where vec_n had all the updates and re-inserts during current rebuild epoch.

   If the rebuild was not complete, then for $v \in [\lambda]$,

   (a) $str_v, value_v \xleftarrow{\$} \{0,1\}^k$ where $str_v, value_v$ are of appropriate lengths

   (b) add str_v to vec_n and output $(str_v, value_v)$.

3. **Token Simulation.** The simulator takes as input the query leakage which is equal to $\mathcal{L}_\mathsf{Q}(\mathsf{MM}, \ell) = (\mathrm{QP}, \mathrm{RL}, \mathrm{OP})$ where QP is the search pattern, $\mathrm{RL} = \#\mathsf{MM}[\ell]$ is the response length pattern and OP is the operation pattern which captures the update tokens and rebuild insertions of ℓ. It first extracts from RL and OP the old and new response lengths, $\#\mathsf{DX}_o[\ell]$ and $\#\mathsf{DX}_n[\ell]$ respectively, such that $\#\mathsf{MM}[\ell] = \#\mathsf{DX}_o[\ell] + \#\mathsf{DX}_n[\ell]$. It initializes empty vectors S_1 and S_2 and if the first rebuild hasn't been completed yet it executes the following steps:

(a) if $\mathsf{MM}_0[\ell] = \perp$:

    i. set $count = \#\mathsf{DX}_o[\ell]$;

    ii. from i = 1 to $count$, pick and remove a label from vec_l and add it to S_1;

    iii. assign S_1 to $\mathsf{MM}_0[\ell]$.

(b) Using OP, extract all labels that were updates of ℓ from vec_n, and add them to S_2.

But if the first rebuild was completed, it simply does the following:

(a) using OP, extract all labels that were updates or re-inserts of ℓ from vec_0 and add to S_1;

(b) using OP, extract all labels that were updates or re-inserts of ℓ from vec_n and add to S_2.

It then outputs $\mathsf{otk} = S_1$ and $\mathsf{ntk} = S_2$.

Now, we have to show that for all PPT adversaries $\mathcal{A}$, the output of the real experiment and ideal experiment are indistinguishable. This can be shown by standard sequence of games argument that shows that EMM, utk and tk are indistinguishable from the real ones due to the RCPA security of SKE and the pseudo-randomness of F.

Game_0: It is the same as a real experiment.

Game_1: It is the same as Game_0 except that we replace the function F by a call to random function G. This is indistinguishable because of the pseudo-randomness of F.

Game_2: It is the same as Game_1 except that we do not generate any keys and replace encryption steps to simply producing a random string. During rebuild step for a searched for label, we simply remove encrypted values at random from the result set till the result set is of the leaked size. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.

Game_3: We now get rid of G and use random strings for labels. In setup, keep track of all labels (random strings) generated. When generating search tokens, we pick these labels at random for a fresh query as the number of labels originally in the old dictionary for the query is leaked if the first rebuild hasn't been completed. We assign these labels to this particular query for future repetitions till the first rebuild is completed. If it has, these labels have already been re-inserted so using OP, we can retrieve them. We further pick appropriate labels from both dictionaries as the total number of updates and re-inserts in each dictionary and when they were received is also leaked. During rebuild step for unsearched labels, we pick λ labels one by one at random and then replace them with a new random string. This is the same as Game_2 because the output of G and a random string are indistinguishable.

Game_3: It is the same as an ideal experiment. The rebuild step is now essentially equivalent to sending λ label, value pairs (which are both random strings now) whenever Update protocol is executed.

E Proof of Theorem 5.2

Theorem 5.2. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* DLS *is* $(\mathcal{L}_{\mathsf{SN}}, m, \ell)$*-multi-snapshot secure, for* $m, \ell \in \mathsf{poly}(k)$*.*

Proof. Consider the simulator that works as follows.

1. The snapshot leakage is composed of the total number of label, value pairs in the old dictionary. So the simulator initializes DX_0 and DX_1 of the size $\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$. For $1 \leq i \leq \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$, it performs the following steps:

   (a) compute $str_i, ctr_i \xleftarrow{\$} \{0,1\}^k$ where str_i, ctr_i are of appropriate lengths;

   (b) set $\mathsf{DX}_0[str_i] = ctr_i$.

   Then output $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$ and set a variable $size_1 = 0$ which captures the size of DX_1.

2. The snapshot leakage is composed of the size of the dictionaries $(\#\mathsf{DX}_o, \#\mathsf{DX}_n)$. The simulator then sets *count* to be $\#\mathsf{DX}_n - size_1$ where $size_1$ captures the size of DX_n before the update. Then for $1 \leq v \leq count$, it does the following:

   (a) compute $str, value \xleftarrow{\$} \{0,1\}^k$ where $str, value$ are of appropriate lengths;

   (b) set $\mathsf{DX}_1[str] = ctr$.

   It outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$ and sets $size_1$ to $\#\mathsf{DX}_n$.

3. The rebuild eakage for a snapshot adversary is, similarly, composed of the size of the dictionaries $(\#\mathsf{DX}_o, \#\mathsf{DX}_n)$ after the rebuild. The simulator sets *count* to be $\#\mathsf{DX}_o - size_1$ as the DX_n is now DX_o. Then for i from 1 to *count*, it does the following:

   (a) compute $str, ctr \xleftarrow{\$} \{0,1\}^k$ where str, ctr are of appropriate lengths;

   (b) set $\mathsf{DX}_1[str] = ctr$.

   It then sets $size_1 := 0$, $\mathsf{DX}_0 := \mathsf{DX}_1$, deletes everything from DX_1 and outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$.

Now, we have to show that for PPT adversaries $\mathcal{A}$, the output of the real experiment and ideal experiment are indistinguishable. This can be shown by standard sequence of games argument that shows that snapshots of EMM after different protocols are indistinguishable from the real ones due to the RCPA security of SKE and the pseudo-randomness of F.

Game_0: It is the same as a real experiment.

Game_1: It is the same as Game_0 except that we replace the function F by a call to random function G. This is indistinguishable because of the pseudo-randomness of F.

Game_2: It is the same as Game_1 except that we do not generate any keys and replace encryption steps to simply producing a random string. During rebuild step for searched for labels, we now use the snap leakage capturing the decrease in size ($\#del$) of the old dictionary, and just create label, value pairs that are exactly $\#del$ less than the original size in number.

It does not matter if we remove from each individual result sets of the searched for labels accurately as long as the overall size decrease is satisfied. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.

Game_3: It is the same as Game_2 except that we now remove the random function G and replace it with random strings for labels and during rebuild. We simply generate random label, value pairs that are equal in number to the new size of the old dictionary and add them to the new dictionary. This is the same as Game_2 because output of G and a random string are indistinguishable.

Game_3 is the same as an ideal experiment.

■

F Proof of Theorem 6.2

Theorem F.1. *If* SKE *is an RCPA-secure encryption scheme and* F *is a pseudo-random function, then* DLS^d *is* $(m, \mathcal{L}_{\mathsf{SN}})$*-snapshot secure, for* $m, \ell \in \mathsf{poly}(k)$.

Proof. Consider the simulator that works as follows.

1. The snapshot leakage consists of the total number of label, value pairs in the old dictionary. The simulator takes this as input and initializes a dictionary DX_0 of size $\sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$ and an empty dictionary DX_1. Then for $1 \leq i \leq \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$, it does the following:

   (a) $str_i, ctr_i \xleftarrow{\$} \{0,1\}^k$ where str_i, ctr_i are of appropriate lengths;

   (b) set $\mathsf{DX}_0[str_i] = ctr_i$.

   It then outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$ and sets $size_1 = 0$ which captures the size of DX_1.

2. We can derive from the snapshot leakage the size of the old and new dictionaries, $\#\mathsf{DX}_o$ and $\#\mathsf{DX}_n$ respectively. The simulator takes this derived leakage as input and simulates Update as follows:

   (a) if $\#\mathsf{DX}_n$ is 0 (this is when a rebuild just got completed), it sets $count$ to be $\#\mathsf{DX}_o - size_1$ and for $1 \leq v \leq count$ it does the following,

      i. $str, value \xleftarrow{\$} \{0,1\}^k$ where $str, value$ is of appropriate lengths;

      ii. set $\mathsf{DX}_1[str] = ctr$.

   If $size_1 > 0$, it then sets $size_1 := 0$, $\mathsf{DX}_0 := \mathsf{DX}_1$, deletes everything from DX_1. It then outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$.

   (b) Otherwise, it sets $count$ to be $\#\mathsf{DX}_n - size_1$ and for $1 \leq v \leq count$ it does the following,

      i. $str, value \xleftarrow{\$} \{0,1\}^k$ where $str, value$ is of appropriate lengths;

      ii. set $\mathsf{DX}_1[str] = ctr$.

   It then outputs $\mathsf{EMM} = (\mathsf{DX}_0, \mathsf{DX}_1)$ and sets $size_1$ to $\#\mathsf{DX}_n$.

Now, we have to show that for all PPT adversaries $\mathcal{A}$, the output of the real experiment and ideal experiment are indistinguishable. This can be shown by standard sequence of games argument that shows that snapshots of EMM after different protocols are indistinguishable from the real ones due to the RCPA security of SKE and the pseudo-randomness of F.

Game_0: It is the same as a real experiment.

Game_1: It is the same as Game_0 except that we replace the function F by a call to random function G. This is indistinguishable because of the pseudo-randomness of F.

Game_2: It is the same as Game_1 except that we do not generate any keys and replace encryption steps to simply producing a random string. During rebuild step for searched for labels, as λ is leaked we simply output λ pairs without caring about deletes. When the rebuild is complete, we would automatically send the the right number of pairs. RCPA security of SKE guarantees indistinguishability between a ciphertext and a randomly generated string.

Game_3: We now get rid of G and use random strings for labels and during rebuild, we simply generate random label, value pairs that are equal in number to the new size of the old dictionary and add them to the new dictionary. This is the same as Game_2 because output of G and a random string are indistinguishable.

Game_3 is the same as an ideal experiment.

This concludes our proof.

$\blacksquare$