Structured Encryption and Dynamic Leakage Suppression

Marilyn George
Brown University

Seny Kamara
Brown University

Tarik Moataz
Aroki Systems

Abstract. Structured encryption (STE) schemes encrypt data structures in such a way that they can be privately queried. Special cases of STE include searchable symmetric encryption (SSE) and graph encryption. Like all sub-linear encrypted search solutions, STE leaks information about queries against persistent adversaries. To address this, a line of work on *leakage suppression* was recently initiated that focuses on techniques to mitigate the leakage of STE schemes.
A notable example is the query equality suppression framework (Kamara et al. *CRYPTO'18*) which transforms dynamic STE schemes that leak the query equality into new schemes that do not. Unfortunately, this framework can only produce static schemes and it was left as an open problem to design a solution that could yield dynamic constructions.
In this work, we propose a dynamic query equality suppression framework that transforms volume-hiding semi-dynamic or mutable STE schemes that leak the query equality into new *fully-dynamic* constructions that do not. We then use our framework to design three new fully-dynamic STE schemes that are "almost" and fully zero-leakage which, under natural assumptions on the data and query distributions, are asymptotically more efficient than using black-box ORAM simulation. These are the first constructions of their kind.

1 Introduction

The problem of encrypted search has received a lot of attention over the years from both the research community and industry. The ability to efficiently search and query encrypted data has the potential to change how we store and process data and help increase the wide-scale deployment of end-to-end encryption. A key requirement for any practical encrypted search solution is handling search queries in sub-linear time. Sub-linear encrypted search can be achieved based on several cryptographic primitives, including property-preserving encryption (PPE), structured encryption (STE) and oblivious RAM (ORAM). Each of these primitives have been heavily investigated and are known to achieve different tradeoffs between efficiency, expressiveness and security/leakage.

Leakage. All sub-linear encrypted search primitives leak information which has motivated the study of leakage attacks to investigate the real-world security of these primitives. In 2015, Naveed, Kamara and Wright [34] described data-recovery attacks in the snapshot setting against schemes that leak data equality and order. In 2012, Islam, Kuzu and Kantarcioglu [23] described a query-recovery attack against schemes that leak query co-occurrences (i.e., whether two keywords appear in the same document). The IKK attack was subsequently shown not to work in the standard adversarial model [10] but followup work described attacks in stronger adversarial models where the adversary is assumed to either know or choose a fraction of the client's data [10,6]. The known-data attacks of [10] exploit co-occurrence leakage and require a large fraction of known data whereas the attacks of [6] require a smaller fraction of known-data and exploit response length leakage; making them applicable to ORAM-based solutions as well. The chosen-data attacks of [46] exploit the response identity (i.e., identifiers of the files that contain the keyword) whereas the recent attacks of [6] only exploit response lengths; again, making them applicable to ORAM-based solutions. Several works have also described leakage attacks on the profiles of known oblivious and encrypted range schemes [30,32,21,22]. In [2], it is shown that highly-efficient STE schemes with zero-leakage queries can be achieved in the snapshot model.

Leakage suppression. Recently, Kamara, Moataz and Ohrimenko initiated the study of leakage suppression [27], which are methods to diminish and eradicate the leakage of STE schemes. There are two kinds of leakage suppression techniques: compilers and data transformations. Compilers take an STE scheme and transform it into a new scheme with similar efficiency but with an improved leakage profile. An example is the cache-based compiler (CBC) of [27] which is a generalization of the seminal Square Root ORAM construction of Goldreich and Ostrovsky [19]. The CBC takes any rebuildable STE scheme that leaks the query equality and possibly some other pattern patt, and transforms it into a new scheme that leaks only the non-repeating sub-pattern of patt. The non-repeating sub-pattern of a leakage pattern is the leakage it produces when queried only on non-repeating query sequences.

Data transformations change plaintext data structures in such a way that leakage is less harmful. The simplest example of a data transformation is padding, which mitigates response length leakage, but more sophisticated approaches include the clustering-based techniques of Bost and Fouque [8] and the transformation that underlies the PBS construction [27], both of which mitigate volume leakage. Recently, Kamara and Moataz also introduced computationally-secure transformations (as opposed to the previously mentioned approaches which are information-theoretic) to mitigate volume leakage [26].

Dynamic leakage suppression. The main advantage of suppression compilers over transformations is that they can be applied to large classes of schemes. For example, the CBC can be applied to any rebuildable STE scheme and, furthermore, [27] shows that any semi-dynamic STE scheme can be made rebuildable. An STE scheme is semi-dynamic if it supports additions but not deletions, and it is fully-dynamic if it supports both. The main limitation of the techniques from [27] is that they only produce *static* schemes even if the base construction is dynamic. While static STE schemes have several applications, dynamic schemes allow the encrypted data structure to adapt to changing data, which is more useful from a practical standpoint.

1.1 Our Contributions

In this work, we address the main problem left open by [27] which is to design a dynamic leakage suppression framework for the query equality. As we will see, solving this open problem results in three new low- and zero-leakage dynamic constructions that, under natural conditions on the data and queries, are asymptotically more efficient than black-box ORAM simulation.

Dynamic compilers. The suppression framework of [27], which includes the CBC and the rebuild compiler (RBC), can be used to compile any semi-dynamic STE scheme that leaks the query equality into a new scheme that does not. But, as discussed, this framework can only produce static schemes; i.e., it does not preserve the (semi-)dynamism of the base scheme. In this work, we propose dynamic variants of the CBC and RBC that suppress the query equality while preserving the dynamism of the base scheme.

Designing such compilers is challenging for several reasons. For example, consider that if the base scheme leaks the response length as well as the operation identity pattern (i.e., whether an operation is a query or an update), the adversary can learn the query equality as follows. Suppose that the largest response length observed is n and that it occurs at some time t. Furthermore, suppose that at time $t+1$ an update operation occurs and that at some time $t' > t+1$ another query occurs with response larger than n. For some datasets and query distributions, it would be reasonable for the adversary to infer that the two queries are for the same value which, effectively, is the query equality. Unfortunately, all currently-known fully-dynamic STE schemes leak both the response length and the operation identity patterns.

Our approach, therefore, is to start with schemes that do not leak the response length like PBS [27] and AVLH [26]. The challenge in using these schemes, however, is that they are not dynamic but only semi-dynamic or mutable (i.e., they only support edit operations). To address this, our compiler is designed to work with these limited forms of dynamism but this requires overcoming a set of additional technical challenges like "upgrading" the base scheme's dynamism from semi-dynamic or mutable to fully-dynamic without leaking any additional information.

New constructions. We apply our compilers to three base multi-map encryption schemes to construct dynamic zero- and almost zero-leakage multi-map encryption schemes. Our first construction results from applying our compilers to the PBS construction of [27]. This results in a dynamic variant of the AZL scheme [27] which, given a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_t)$, reveals nothing on operations $(\mathsf{op}_1, \ldots, \mathsf{op}_{t-1})$ and then reveals the sum of the operations' response lengths on operation op_t. Similarly, our second construction results from applying our compilers to a variant of PBS and is a dynamic variant of the FZL scheme of [27]. This scheme has zero-leakage queries but only achieves probabilistic correctness. Our third construction, which results from applying our compilers to the AVLH construction of [26], also has ZL queries but achieves perfect correctness. We show that all three schemes are asymptotically more efficient than state-of-the-art black-box ORAM simulation under natural assumptions.

2 Related Work

Structured encryption Structured encryption was introduced by Chase and Kamara in [14] as a generalization of searchable symmetric encryption (SSE) [41,15]. Several aspects of STE and SSE have been studied including dynamism [29,28,11,35], expressiveness [12,37,17,24,25], locality and I/O-efficiency [13,3,11,16,4], security [42,7,18,9,2] and cryptanalysis [23,10,30,46,32,6].

Leakage suppression. Leakage suppression was first proposed by Kamara, Moataz and Ohrimenko [27] who generalized and adapted the techniques from Goldreich and Ostrovsky's seminal Square-Root ORAM to STE. Recently, Kamara and Moataz showed, for the first time, how to design volume-hiding STE schemes [26] without making use of padding. In follow up work, Patel, Persiano, Yeo and Yung [39] proposed new volume-hiding constructions that achieve better query and storage efficiency.

Oblivious RAM. Oblivious RAM was first proposed by Goldreich and Ostrovsky [19]. Several aspects of ORAM have been studied and improved in the last twenty years including its communication complexity, the number of rounds and client and server storage [36,45,20,31,40,43,18,38]. Another line of work initiated by Wang et al. [44] considers the design of oblivious data structures, without making use of general-purpose ORAM techniques. These constructions are typically more efficient than using general-purpose ORAM but are usually static or require setting an upper bound the structure at setup time.

3 Preliminaries and Notation

Notation. We denote the security parameter as k, and all algorithms run in time polynomial in k. 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 $\mathbf{v}$ of n elements, we refer to its i^{th} element as v_i or $\mathbf{v}[i]$. If S is a set then $\#S$ refers to its cardinality. If s is a string then $|s|_2$ refers to its bit length.

Sorting networks. A sorting network is a circuit of comparison-and-swap gates. A sorting network for n elements takes as input a collection of n elements $(a_1,\ldots,a_n)$ and outputs them in increasing order. Each gate g in an n-element network SN_n specifies two input locations $i, j \in [n]$ and, given a_i and a_j, returns the pair (a_i, a_j) if $i < j$ and (a_j, a_i) otherwise. Sorting networks can be instantiated with the asymptotically-optimal Ajtai-Komlos-Szemeredi network

[1] which has size $O(n \log n)$ or Batcher's more practical network [5] with size $O(n \log^2 n)$ but with small constants.

The word RAM. Our model of computation is the word RAM. In this model, we assume memory holds an infinite number of w-bit words and that arithmetic, logic, read and write operations can all be done in $O(1)$ time. We denote by $|x|_w$ the word-length of an item x; that is, $|x|_w = |x|_2/w$. Here, we assume that $w = \Omega(\log k)$.

Abstract data types. An *abstract data type* specifies the functionality of a data structure. It is a collection of data 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*. We model a dynamic data type $\mathbf{T}$ as a collection of four spaces: the object space $\mathbb{D} = \{\mathbb{D}_k\}_{k\in\mathbb{N}}$, the query space $\mathbb{Q} = \{\mathbb{Q}_k\}_{k\in\mathbb{N}}$, the response space $\mathbb{R} = \{\mathbb{R}_k\}_{k\in\mathbb{N}}$ and the update space $\mathbb{U} = \{\mathbb{U}_k\}_{k\in\mathbb{N}}$. We also define the query map $\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}$ and the update map $\mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D}$ to represent operations associated with the dynamic data type. We refer to the query and update spaces of a data type as the operation space $\mathbb{O} = \mathbb{Q} \cup \mathbb{U}$. When specifying a data type $\mathbf{T}$ we will often just describe its maps $(\mathsf{qu}, \mathsf{up})$ from which the object, query, response and update spaces can be deduced. The spaces are ensembles of finite sets of finite strings indexed by the security parameter. We assume that $\mathbb{R}$ includes a special element $\perp$ and that $\mathbb{D}$ includes an empty object d_0 such that for all $q \in \mathbb{Q}$, $\mathsf{qu}(d_0, q) = \perp$.

Data structures. A type-$\mathbf{T}$ data *structure* is a representation of data objects in $\mathbb{D}$ in some computational model (as mentioned, here it is the word RAM). Typically, the representation is optimized to support qu as efficiently as possible; that is, such that there exists an efficient algorithm Query that computes the function qu. 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 $O(\log n)$ time.

Definition 1 (Structuring scheme). *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is composed of three polynomial-time algorithms that work as follows:*

- $\mathsf{DS} \leftarrow \mathsf{Setup}(d)$*: is a possibly probabilistic algorithm that takes as input a data object* $d \in \mathbb{D}$ *and outputs a data structure* DS*. Note that* d *can be represented in any arbitrary manner as long as its bit length is polynomial in* k*. Unlike* DS*, its representation does not need to be optimized for any particular query.*
- $r \leftarrow \mathsf{Query}(\mathsf{DS}, q)$*: is an algorithm that takes as input a data structure* DS *and a query* $q \in \mathbb{Q}$ *and outputs a response* $r \in \mathbb{R}$*.*

– $\mathsf{DS} \leftarrow \mathsf{Update}(\mathsf{DS}, u)$*: is a possibly probabilistic algorithm that takes as input a data structure* DS *and an update* $u \in \mathbb{U}$ *and outputs a new data structure* DS.

Here, we allow Setup and Update to be probabilistic but not Query. This captures most data structures but the definition can be extended to include structuring schemes with probabilistic query algorithms. We say that a data structure DS *instantiates* a data object $d \in \mathbb{D}$ if for all $q \in \mathbb{Q}$, $\mathsf{Query}(\mathsf{DS}, q) = \mathsf{qu}(d, q)$. We denote this by $\mathsf{DS} \equiv d$. We denote the set of queries supported by a structure DS as $\mathbb{Q}_{\mathsf{DS}}$; that is,

$$\mathbb{Q}_{\mathsf{DS}} \overset{def}{=} \Big\{ q \in \mathbb{Q} : \mathsf{Query}(\mathsf{DS}, q) \neq \perp \Big\}.$$

Similarly, the set of responses supported by a structure DS is denoted $\mathbb{R}_{\mathsf{DS}}$.

Definition 2 (Correctness). *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \rightarrow \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \rightarrow \mathbb{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is perfectly correct if it satisfies the following properties:*

1. *(static correctness) for all* $d \in \mathbb{D}$,

$$\Pr\left[\, \mathsf{DS} \equiv d : \mathsf{DS} \leftarrow \mathsf{Setup}(d) \,\right] = 1,$$

where the probability is over the coins of Setup.

2. *(dynamic correctness) for all* $d \in \mathbb{D}$ *and* $u \in \mathbb{U}$*, for all* $\mathsf{DS} \equiv d$,

$$\Pr\left[\, \mathsf{Update}(\mathsf{DS}, u) \equiv \mathsf{up}(d, u) \,\right] = 1,$$

where the probability is over the coins of Update.

Note that the second condition guarantees the correctness of an updated structure whether the original structure was generated by a setup operation or a previous update operation. Weaker notions of correctness (e.g., for data structures like Bloom filters) can be derived from Definition 2.

Basic data structures. We use structures for several basic data types including arrays, dictionaries and multi-maps which we recall here. An array RAM of capacity n stores n items at locations 1 through n and supports read and write operations. We write $v := \mathsf{RAM}[i]$ to denote reading the item at location i and $\mathsf{RAM}[i] := v$ the operation of storing an item at location i. A dictionary structure DX of capacity n holds 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. A multi-map structure 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. 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).

Data structure logs. Given a structure DS that instantiates an object d, we will be interested in the sequence of update operations needed to create a new structure DS' that also instantiates d. We refer to this as the *query log* of DS and assume the existence of an efficient algorithm Log that takes as input DS and outputs a tuple $(u_1, \ldots, u_n)$ such that adding $u_1, \ldots, u_n$ to an empty structure results in some $\mathsf{DS}' \equiv d$.

Extensions. An important property we will need from a data structure is that it be *extendable* [27] in the sense that, given a structure DS one can create another structure $\overline{\mathsf{DS}} \neq \mathsf{DS}$ that is functionally equivalent to DS but that also supports a number of *dummy* queries. We say that a structure is efficiently extendable if there exist a query set $\overline{\mathbb{Q}} \supset \mathbb{Q}$ and a PPT algorithm $\mathsf{Ext}_{\mathbf{T}}$ that takes as input a structure DS of type $\mathbf{T}$ and a *capacity* $\lambda \geq 1$ and returns a new structure $\overline{\mathsf{DS}}$ also of type $\mathbf{T}$ [1] such that: (1) $\overline{\mathsf{DS}} \equiv d$; and (2) for all $q \in \overline{\mathbb{Q}} \backslash \mathbb{Q}$, $\mathsf{Query}(\overline{\mathsf{DS}}, q) = \perp$.We say that $\overline{\mathsf{DS}}$ is an extension of DS and that DS is a sub-structure of $\overline{\mathsf{DS}}$.

Cryptographic protocols. We denote by $(\mathsf{out}_A, \mathsf{out}_B) \leftarrow \Pi_{A,B}(X, Y)$ the execution of a two-party protocol Π between parties A and B, where X and Y are the inputs provided by A and B, respectively; and out_A and out_B are the outputs returned to A and B, respectively.

3.1 Structured Encryption

We recall the syntax definition of STE.

Definition 3 (Structured encryption [14]). *An interactive structured encryption scheme $\Sigma = (\mathsf{Setup}, \mathsf{Operate})$ consists of an algorithm and a two-party protocol that work as follows:*

- $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \lambda, \mathsf{DS})$*: is a probabilistic polynomial-time algorithm that takes as input a security parameter 1^k, a query capacity $\lambda \geq 1$ and a type-$\mathbf{T}$ structure DS. It outputs a secret key K, a state st and an encrypted structure EDS. If $\mathsf{DS} \equiv d_0$, it outputs an empty EDS.*
- $\big((st', r), \mathsf{EDS}'\big) \leftarrow \mathsf{Operate}_{\mathbf{C},\mathbf{S}}\big((K, st, \mathsf{op}), \mathsf{EDS}\big)$*: is a two-party protocol executed between a client and a server where the client inputs a secret key K, a state st and an operation op and the server inputs an encrypted structure EDS. The client receives as output a (possibly) updated state st' and a response $r \in \mathbb{R} \cup \perp$ while the server receives a (possibly updated) encrypted structure EDS'.*

If Σ also has a $\mathsf{Rebuild}$ protocol as defined below, we say that it is rebuildable,

- $\big((st', K'), \mathsf{EDS}'\big) \leftarrow \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\left((K, st), \mathsf{EDS}\right)$*: is a two-party protocol executed between the client and server where the client inputs a secret key K and a state st. The server inputs an encrypted data structure EDS. The client receives an updated state st' and a new key K' as output while the server receives a new structure EDS'.*

[1] We consider that the inclusion of dummy queries in a query space does not impact the type of a structure.

Operations. Note that an STE schemes usually supports more than a single operation and the syntax above can be used (or extended) to capture this in one of two ways. The first is to notice that the $\mathsf{Operate}$ protocol can take as input an operation op that describes one of a set of operations and its operands. For example, if $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Operate})$ supports both query and add operations, then op can have the form $\mathsf{op} = (\mathsf{qry}, q)$ to denote a query operation for q or $\mathsf{op} = (\mathsf{add}, a)$ to denote an add operation for a. The $\mathsf{Operate}$ protocol can then operate on EDS accordingly and output $((st, r), \mathsf{EDS}')$, where $r \neq \perp$ and $\mathsf{EDS}' = \mathsf{EDS}$ in the case of a query, and where $r = \perp$ and $\mathsf{EDS}' \neq \mathsf{EDS}$ in the case of an add. For notational convenience we will usually omit the flags qry or add and just write $\mathsf{op} = q$ or $\mathsf{op} = a$ to denote that it is a query or and add. This formulation is particularly convenient when working with schemes that hide which operation is being executed, as will be the case with our main constructions. Another approach is to include the different operations explicitly in Σ_{DS}'s syntax. For example, if it supports queries and adds, then we would write $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$, where Query is a special case of $\mathsf{Operate}$ that (usually) outputs a response $r \neq \perp$ and an $\mathsf{EDS}' = \mathsf{EDS}$ and Add is a special case that (usually) outputs $r = \perp$ and $\mathsf{EDS}' \neq \mathsf{EDS}$. This formulation is particularly convenient when working with schemes that reveal which operation is being executed, as will be the case with the constructions we use as building blocks.

Dynamism. We consider several kinds of dynamic STE schemes. The first are *fully-dynamic* schemes which support add and delete operations. We usually refer to such schemes simply as *dynamic*. Add operations insert a query/response pair (q, r) into the data structure whereas delete operations remove query/response pairs (q, r) associated with a given query q. If a scheme only handles add operations we say it is *semi-dynamic*. Finally, we consider *mutable* schemes which are schemes that support an edit operation which takes as input a query/response pair (q, r') and changes a pre-existing pair (q, r) to (q, r'). If a scheme is either semi-dynamic or mutable we say that it is *weakly dynamic*.

Security. We recall the notion of adaptive semantic security for STE.

Definition 4 (Security [15,14]). *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Operate}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *be a structured encryption scheme and consider the following probabilistic experiments where* $\mathcal{C}$ *is a stateful challenger,* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator,* $\Lambda = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{O}}, \mathsf{patt}_{\mathsf{R}})$ *is a leakage profile,* $\lambda \geq 1$ *and* $z \in \{0,1\}^*$*:*

$\mathbf{Real}_{\Sigma,\mathcal{C},\mathcal{A}}(k)$*: given* z *and* λ *the adversary* $\mathcal{A}$ *outputs a structure* DS *and receives* EDS *from the challenger, where* $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \lambda, \mathsf{DS})$*.* $\mathcal{A}$ *then adaptively chooses a polynomial-size sequence of operations* $(\mathsf{op}_1, \ldots \mathsf{op}_m)$*. For all* $1 \leq i \leq m$ *the challenger and adversary do the following:*

1. *if* op_i *is a query or an update, they execute* $\mathsf{Operate}_{\mathcal{C},\mathcal{A}}\big((K, st, \mathsf{op}_i), \mathsf{EDS}\big)$*;*
2. *if* op_i *is a rebuild, they execute* $\mathsf{Rebuild}_{\mathcal{C},\mathcal{A}}\big((K, st), \mathsf{EDS}\big)$*.*

Finally, $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: given* z *and* λ *the adversary* $\mathcal{A}$ *outputs a structure* DS *of type* $\mathbf{T}$*. Given* $\mathsf{patt}_{\mathsf{S}}(\mathsf{DS})$*, the simulator returns an encrypted structure* EDS *to* $\mathcal{A}$*.* $\mathcal{A}$

then adaptively chooses a polynomial-size sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_m)$. *For all* $1 \leq i \leq m$*, the challenger, simulator and adversary do the following:*

1. *if* op_i *is either a query or an update,* $\mathcal{S}$ *is given* $\mathsf{patt}_{\mathsf{O}}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_i)$ *and it executes* $\mathsf{Operate}_{\mathcal{S},\mathcal{A}}$ *with* $\mathcal{A}$*;*
2. *if* op_i *is a rebuild,* $\mathcal{S}$ *is given* $\mathsf{patt}_{\mathsf{R}}(\mathsf{DS})$ *and it executes* $\mathsf{Rebuild}_{\mathcal{S},\mathcal{A}}$ *with* $\mathcal{A}$*;*

Finally, $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

We say that Σ *is* Λ*-secure if there exists a* PPT *simulator* $\mathcal{S}$ *such that for all* PPT *adversaries* $\mathcal{A}$*, for all* $\lambda \geq 1$ *and all* $z \in \{0,1\}^*$*,*

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

Note that security of non-rebuildable schemes can be recovered by not allowing rebuild operations.

Leakage. We extend the leakage patterns defined in [27] to the dynamic setting. In particular [27] defined leakage patterns as functions of queries on a static data type. We will have to extend the definitions to account for general operations (queries or updates) on a dynamic data type. Let $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ be a dynamic data type. We assume that updates can be written as query/response pairs, i.e., $\mathbb{U} = \mathbb{Q} \times \mathbb{R}$. Given a data structure d and a sequence of t operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$, we denote by d_t the structure that results from applying the given sequence of operations to d. Consider the following leakage patterns,

- the *operation identity pattern* is the function family $\mathsf{oid} = \{\mathsf{oid}_{k,t}\}_{k,t \in \mathbb{N}}$ with $\mathsf{oid}_{k,t} : \mathbb{D}_k \times \mathbb{O}_k^t \to \{0,1\}^t$ such that $\mathsf{oid}_{k,t}(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \mathbf{m}$, where $\mathbf{m}$ is a binary t-dimensional vector such that $\mathbf{m}[i] = 0$ if $\mathsf{op}_i \in \mathbb{Q}$ and $\mathbf{m}[i] = 1$ if $\mathsf{op}_i \in \mathbb{U}$;
- the *update query equality pattern* is the function family $\mathsf{uqeq} = \{\mathsf{uqeq}_{k,t}\}_{k,t \in \mathbb{N}}$ with $\mathsf{uqeq}_{k,t} : \mathbb{D}_k \times \mathbb{U}_k^t \to \{0,1\}^{t \times t}$ such that $\mathsf{uqeq}_{k,t}(d, u_1, \ldots, u_t) = M$, where M is a binary $t \times t$ matrix such that for updates $u_i = (q_i, r_i)$ and $u_j = (q_j, r_j)$, $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ otherwise;
- the *operation total response length pattern* is the function family $\mathsf{otrlen} = \{\mathsf{otrlen}_k\}_{k \in \mathbb{N}}$ with $\mathsf{otrlen}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{otrlen}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \sum_{q \in \mathbb{Q}_k} |\mathsf{qu}(d_t, q)|_w$ and d_t is d after t operations.;
- the *operation data size pattern* is the function family $\mathsf{odsize} = \{\mathsf{odsize}_k\}_{k \in \mathbb{N}}$ with $\mathsf{odsize}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{odsize}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = |d_t|_w$;
- the *operation log size pattern* is the function family $\mathsf{olsize} = \{\mathsf{olsize}_k\}_{k \in \mathbb{N}}$ with $\mathsf{olsize}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{olsize}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \#\mathsf{Log}(\mathsf{DS})$ where DS is an instantiation of d_t such that $\mathsf{DS} \equiv d_t$;
- the *operation max log length pattern* is the function family $\mathsf{omllen} = \{\mathsf{omllen}_k\}_{k \in \mathbb{N}}$ with $\mathsf{omllen}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{omllen}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \max_{\mathsf{op} \in \mathsf{Log}(d_t)} |\mathsf{op}|_w$.

Note that in the static setting, i.e., when $\mathbb{O} = \mathbb{Q}$, the leakage patterns $\mathsf{otrlen}, \mathsf{odsize}, \mathsf{olsize}, \mathsf{omllen}$ are equivalent to the patterns $\mathsf{trlen}, \mathsf{dsize}, \mathsf{lsize}, \mathsf{mllen}$ originally defined in [27].

Leakage sub-patterns. We recall the notion of leakage sub-patterns introduced in [27]. Given a leakage pattern patt, it can be decomposed into sub-patterns capturing its behavior on restricted classes of query sequences. In particular, we can decompose a leakage pattern into repeating and non-repeating sub-patterns. The non-repeating sub-pattern is pattern that results from evaluating patt on non-repeating query sequences (i.e., where all queries are unique).

Definition 5 (Non-repeating sub-patterns). *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic data type and* $\mathsf{patt} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$ *be a query leakage pattern. The non-repeating sub-pattern of* patt *is the function* uniq *such that*

$$\mathsf{patt}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \mathsf{uniq}(\mathsf{DS}, q_1, \ldots, q_t) & \text{if } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{other}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise.} \end{cases}$$

Safe extensions. We recall and extend the notion of safe extension from [27] to support updates.

Definition 6 (Safe extensions). *Let* $\Lambda = (\mathsf{patt}_\mathsf{S}, \mathsf{patt}_\mathsf{Q}, \mathsf{patt}_\mathsf{U}, \mathsf{patt}_\mathsf{R})$ *be a leakage profile. We say that an extension* Ext *is* Λ*-safe if for all* $k \in \mathbb{N}$*, for all* $d \in \mathbb{D}_k$*, for all* $\mathsf{DS} \equiv d$*, for all* $\lambda \geq 1$*, for all* $\overline{\mathsf{DS}}$ *output by* $\mathsf{Ext}(\mathsf{DS}, \lambda)$*, for all* $t \in \mathbb{N}$*, for all* $\mathbf{op} = (\mathsf{op}_1, \ldots, \mathsf{op}_t) \in \mathbb{O}_k^t$*,*

- $\mathsf{patt}_\mathsf{S}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_\mathsf{S}(\mathsf{DS})$*;*
- $\mathsf{patt}_\mathsf{Q}(\overline{\mathsf{DS}}, q_1, \ldots, q_p) \leq \mathsf{patt}_\mathsf{Q}(\mathsf{DS}, q_1, \ldots, q_p)$*, where* $(q_1, \ldots, q_p)$ *is the subsequence of queries in* $\mathbf{op}$*;*
- $\mathsf{patt}_\mathsf{U}(\overline{\mathsf{DS}}, u_1, \ldots, u_w) \leq \mathsf{patt}_\mathsf{U}(\mathsf{DS}, u_1, \ldots, u_w)$*, where* $(u_1, \ldots, u_w)$ *is the subsequence of updates in* $\mathbf{op}$*;*
- $\mathsf{patt}_\mathsf{R}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_\mathsf{R}(\mathsf{DS})$*,*

where $\mathsf{patt}_1 \leq \mathsf{patt}_2$ *means that* patt_1 *can be simulated from* patt_2*.*

4 Our Dynamic Suppression Framework

In this section, we present a dynamic variant of the query equality suppression framework proposed by [27]. Our framework transforms non-rebuildable weakly-dynamic STE schemes that leak the query equality into fully-dynamic STE schemes that do not. Recall that the static framework relies on two compilers: (1) a rebuild compiler (RBC) which transforms a semi-dynamic and non-rebuildable scheme into a static and rebuildable one; and (2) the cache-based compiler (CBC) which transforms a static and rebuildable scheme that leaks the query equality into a static scheme that does not.

Challenges. One of the challenges in designing a dynamic variant of the CBC is handling subtle correlations between various leakage patterns. For example, suppose the base STE scheme leaks the response length and the operation identity patterns and consider a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_4)$ such that

$\mathsf{op}_1 = q_1$, $\mathsf{op}_2 = q_2$, $\mathsf{op}_3 = u_3$ and $\mathsf{op}_4 = q_4$. Now, given the operation identities and the response lengths, suppose the adversary observes that: q_1 has the largest response length ℓ_1; that q_3 is an update operation; and that q_4 has response length $\ell_1 + 1$. From this, it can reasonably infer that q_1 might be equal to q_4 which is a "probabilistic" variant of the query equality. It is therefore not enough to suppress the exact query equality but also the patterns that can reveal partial information about it.

To address this, our compiler will have to suppress the response length and the operation identity in addition to the query equality. One can trivially suppress the former by padding responses to the maximum length but this induces a large storage cost; especially when the response lengths are skewed. A better approach would be to start with base schemes that are volume-hiding in the sense that they hide the response lengths (without naive padding). Unfortunately, all volume-hiding constructions we are aware of [26,39] are only weakly dynamic. Our goal, therefore, will be to design a compiler that suppresses the query equality, the operation identity and the response length while upgrading the base scheme from being weakly-dynamic to fully-dynamic.

Another important challenge we must overcome is making the base scheme rebuildable. [27] already showed how to make semi-dynamic schemes rebuildable but, in our setting, we also need to handle mutable constructions which do not support add operations but only edits. To summarize, our compiler has to handle the following challenges:

– *(weak dynamism)* it must transform a weakly-dynamic (i.e., either semi-dynamic or mutable) scheme to a fully-dynamic one;
– *(operation identity)* it must suppress the operation identity; that is, queries and updates should look identical.
– *(rebuild)* it must make the base scheme rebuildable even if it is only weakly dynamic.

Overview of the dynamic CBC. The dynamic CBC is similar to the static CBC of [27] with the exception of a few steps to handle adds and edits. Let $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$ be a semi-dynamic STE scheme and let $\Sigma_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Put})$ be a semi-dynamic and zero-leakage dictionary encryption scheme. The compiler produces a new scheme $\Sigma_{\mathsf{DDS}} = (\mathsf{Setup}, \mathsf{Operate})$ that works as follows. Given a structure DS and a capacity $\lambda \geq 1$, its setup algorithm outputs a structure $\mathsf{EDDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$, where $\overline{\mathsf{EDS}}$ is the encryption of a λ-extension of DS and EDX is an encryption of a dictionary with capacity λ. Operations on EDDS are handled as follows:

– *(queries)* to make a query q, the client first executes a get on EDX for q. If this returns $\perp$ (i.e., q has never been issued before) the client queries $\overline{\mathsf{EDS}}$ for q and receives a response r. The client then does a put on EDX to add the query/response pair (q, r). If, on the other hand, the get on the cache returned a response $r \neq \perp$, the client queries $\overline{\mathsf{EDS}}$ for an unused dummy value and puts the query/response pair (q, r) in EDX;

– *(adds)* to add a query/response pair (q, r), the client executes a get on EDX for an arbitrary query and ignores the response. It then queries $\overline{\mathsf{EDS}}$ for an unused dummy and puts (q, r) in EDX;
– *(edits)* to edit the response of an existing query q (e.g., by either adding to it, deleting from it or changing it), the client first executes a get on EDX for q. If this returns $\perp$, the client queries $\overline{\mathsf{EDS}}$ for q and receives a response r. It then edits r, which results in a new response r', and puts (q, r') in EDX. If, on the other hand, the get on the cache returned a response $r \neq \perp$, the client queries $\overline{\mathsf{EDS}}$ for an unused dummy, edits r and puts the edited query/response pair (q, r') in EDX.

Note that for every operation, the dynamic CBC executes a get on EDX, then a query on $\overline{\mathsf{EDS}}$ and, finally, a put on EDX. Furthermore, $\overline{\mathsf{EDS}}$ is never queried for a query q more than once. Intuitively, the first property will guarantee that the scheme suppresses the operation identity while the second will guarantee that it suppresses the query equality.

Every operation executed on EDDS consumes a (unique) dummy item from $\overline{\mathsf{EDS}}$. And since it holds λ dummies, it needs to be rebuilt after λ operations so that it can continue to be used. We now describe how this rebuild is achieved.

Overview of the dynamic RBC. We have two main goals when rebuilding $\mathsf{EDDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$. The first is to build a new $\overline{\mathsf{EDS}}$ structure $\overline{\mathsf{EDS}}'$ that holds the λ dummies. The second is to make sure that $\overline{\mathsf{EDS}}'$ holds the most up-to-date responses for all the queries. Note that the second goal is non-trivial because of the way adds and edits are handled. In particular, the most up-to-date response for a query q can be either in $\overline{\mathsf{EDS}}$ or in EDX depending on whether it has been added, edited or never modified. More precisely, we have hat after λ operations, if a query/response pair (q, r) is in the cache then r is the most up-to-date response for q. On the other hand, if a pair (q, r) is not in the cache then the the main structure $\overline{\mathsf{EDS}}$ holds the most up-to-date response for q. In the following, we refer to a query/response pair (q, r) as *valid* if r is the most up-to-date response for q and as *invalid* if it is not. Our rebuild protocol must then extract the valid query/response pairs from EDX and $\overline{\mathsf{EDS}}$ and add them to $\overline{\mathsf{EDS}}'$ with a minimal amount of leakage. [2]

The protocol consists of five phases: (1) *initialization*, where an array RAM is initialized at the server; (2) *extract-and-tag*, where all the query/response pairs are retrieved from $\overline{\mathsf{EDS}}$ and EDX, tagged according to their validity and stored in an encrypted array at the server; (3) *sort-and-shuffle*, where the encrypted array is (obliviously) sorted to partition the invalid and valid query/response pairs so that the former can be deleted and the latter are randomly shuffled; (4) *update*, where the valid query/response pairs in the array are added to a new $\overline{\mathsf{EDS}}'$ structure; and (5) *cache setup*, where a new cache structure EDX' is created. More precisely, it works as follows:

1. *(initialization):* the server initializes an array RAM.

[2] Note that invalid query/response pairs in $\overline{\mathsf{EDS}}$ result from the pair existing in $\overline{\mathsf{EDS}}$ from setup (i.e., not being added) but being edited during the last λ operations.

2. *(extract-and-tag)* the client sequentially retrieves all the query/response pairs (q, r) in $\overline{\mathsf{EDS}}$ and EDX. For all (q, r) in EDX, it adds an encryption of (q, r, f) to RAM, where f is a random non-zero k-bit value we refer to as a validity tag. If there are less than λ entries in EDX, it queries it on arbitrary values until it reaches λ queries and for each of these arbitrary queries it adds an encryption of $(\perp, \perp, 0)$ to RAM. For all query/response pairs (q, r) in $\overline{\mathsf{EDS}}$, it adds an encryption of (q, r, f) to RAM, where f is set to 0 if q was present in EDX and f is set to a random non-zero k-bit value otherwise. For each dummy in $\overline{\mathsf{EDS}}$, the client adds an encryption of $(\perp, \perp, f)$ to RAM, where f is a random non-zero k-bit value. Throughout this phase, the client also keeps count of the number of entries with 0 tags. Notice that the valid query/response pairs and the dummies are all tagged with random non-zero validity tags whereas the invalid pairs and the entries that result from the "arbitrary" queries on EDX are tagged with 0.
3. *(sort-and-shuffle)* the client obliviously sorts RAM according to the validity tags. Since the valid pairs and the dummies have random non-zero tags and the rest have 0 tags, this step will randomly shuffle the valid pairs and dummies and store the rest at the start of the array. The client then asks the server to delete the first t entries, where t is the number of entries with 0 tags. At this point, the array only holds valid query/response pairs.
4. *(update)* the client creates a new structure $\overline{\mathsf{EDS}'}$ by retrieving the query/response pairs in RAM and adding them to $\overline{\mathsf{EDS}'}$. How exactly this is done depends on the kind of dynamism Σ_{DS} supports:
   - *(semi-dynamic)* if it is semi-dynamic, the client initializes an empty structure DS_0 and encrypts it with Σ_{DS} before storing it at the server. This new encrypted structure is $\overline{\mathsf{EDS}'}$. The client sequentially retrieves the query/response pairs (q, r) from RAM and adds them to $\overline{\mathsf{EDS}'}$.
   - *(mutable)* if Σ_{DS} is mutable we can only use edit operations. The client then sets up "placeholder" structure $\widetilde{\mathsf{DS}}$ that it will encrypt and edit until it holds the necessary data. Note that for this to work, the placeholder must be large enough to hold the latest version of DS (i.e., the structure DS after the λ operations) and it must be "safe" in the sense that encrypting and editing the placeholder must not leak more than operating on the original structure.
5. *(cache setup)* the client generates an empty dictionary with capacity λ and encrypts it with Σ_{DX} and sets it to be EDX'.

Finally, the protocol outputs a rebuilt structure $\mathsf{EDDS}' = (\overline{\mathsf{EDS}'}, \mathsf{EDX}')$.

4.1 Security

We now analyze the security of our dynamic suppression framework. We present two theorems whose proofs are in the full version of this work. Theorem 1 analyzes the case when Σ_{DS} is semi-dynamic and Theorem 2 analyzes the case where

Σ_{DS} is mutable. For Theorem 1, we assume Σ_{DS} has leakage profile

$$\Lambda_{\mathsf{DS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{A}}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, (\mathsf{qeq}^{\mathsf{ds}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}), \mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}}\right),$$

and Σ_{DX} has profile

$$\Lambda_{\mathsf{DX}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{P}}) = \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \perp, \perp\right).$$

Theorem 1 (Semi-dynamic). *If Σ_{DS} is Λ_{DS}-secure, if Ext is $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}})$-safe, and if Σ_{DX} is Λ_{DX}-secure, then Σ_{DDS} is Λ_{DDS}-secure, where*

$$\Lambda_{\mathsf{DDS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = \left(\left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}\right), \mathsf{uniq}, \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\right)\right)$$

and patt_1, patt_2 and patt_3 are defined as,

- $\mathsf{patt}_1(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_0), \mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\right)$
- $\mathsf{patt}_2(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\right)_{q \in \mathbb{Q}_{\mathsf{DS}}}$
- $\mathsf{patt}_3(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}}(\mathsf{DS}_0, a)\right)_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)}$,

where uniq is the non-repeating sub-pattern of $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}$, $\mathsf{DS}_0 \equiv d_0$ and DS_λ is the updated DS after $\lambda \geq 1$ operations.

Before we state our Theorem for mutable schemes, recall that the rebuild protocol needs to setup a placeholder structure that can be edited to realize the new data object. This placeholder must be setup and edited with minimal leakage. We do this with the notion of a safe placeholder which we define below.

Definition 7 (Safe placeholder). *A placeholder structure $\widetilde{\mathsf{DS}}$ is $(\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{Q}}, \mathsf{patt}_{\mathsf{E}})$-safe for a structure DS if, for all queries $q_1, \ldots, q_t$, for all edits $e_1, \ldots, e_t$,*

- $\mathsf{patt}_{\mathsf{S}}(\widetilde{\mathsf{DS}}) \leq \mathsf{patt}_{\mathsf{S}}(\mathsf{DS})$,
- $\mathsf{patt}_{\mathsf{Q}}(\widetilde{\mathsf{DS}}, q_1, \ldots, q_t) \leq \mathsf{patt}_{\mathsf{Q}}(\mathsf{DS}, q_1, \ldots, q_t)$,
- $\mathsf{patt}_{\mathsf{E}}(\widetilde{\mathsf{DS}}, e_1, \ldots, e_t) \leq \mathsf{patt}_{\mathsf{A}}(\mathsf{DS}, e_1, \ldots, e_t)$.

We assume that there exists an efficient algorithm $\mathsf{GenPlaceholder}$ that takes as input some state information and generates a safe placeholder. We now state Theorem 2 whose proof is deferred to the full version of this work. Here, we assume Σ_{DS} has leakage profile

$$\Lambda_{\mathsf{DS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, (\mathsf{qeq}^{\mathsf{ds}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}), \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}\right),$$

and Σ_{DX} has the same profile as above.

Theorem 2 (Mutable). *If Σ_{DS} is Λ_{DS}-secure, if Ext is $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$-safe, if $\widetilde{\mathsf{DS}}$ is an $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$-safe placeholder for DS_λ, and if Σ_{DX} is Λ_{DX}-secure, then Σ_{DDS} is Λ_{DDS}-secure, where*

$$\Lambda_{\mathsf{DDS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = \left(\left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}\right), \mathsf{uniq}, \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\right)\right)$$

and patt_1, patt_2 and patt_3 are defined as,

– $\mathsf{patt}_1(\mathsf{DS}) = \big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_\lambda), \mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\big)$
– $\mathsf{patt}_2(\mathsf{DS}) = \big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\big)_{q \in \mathbb{Q}_{\mathsf{DS}}}$
– $\mathsf{patt}_3(\mathsf{DS}_\lambda) = (\mathsf{patt}_{\mathsf{E}}(\mathsf{DS}_\lambda, e))_{e \in \mathsf{Log}(\mathsf{DS}_\lambda)}$,

where uniq *is the non-repeating sub-pattern of* $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}$*, and* DS_λ *is the updated* DS *after* $\lambda \geq 1$ *operations.*

4.2 Efficiency of the Dynamic Cache-Based Compiler

We now analyze the efficiency of the schemes produced by our suppression framework and compare it to using black-box ORAM simulation.

Operation complexity. The efficiency of Σ_{DDS} clearly depends on the efficiency of its building blocks Σ_{DS} and Σ_{DX}. Recall that for every operation op on $\overline{\mathsf{EDDS}}$, the client executes: one get operation on EDX, one query operation on $\overline{\mathsf{EDS}}$ and one put operation on EDX. This leads to an operation complexity of

$$\mathsf{time}^{\mathsf{dds}}_{\mathsf{O}} = \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}},$$

where $\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}}$ is the query complexity of Σ_{DS}, and $\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}}$ and $\mathsf{time}^{\mathsf{dx}}_{\mathsf{P}}$ are the get and put complexities of Σ_{DX}.

Rebuild complexity. Recall that the Rebuild protocol of Σ_{DDS} executes: (1) λ gets on EDX; (2) $\#\mathbb{Q}_{\mathsf{DS}}$ queries on EDS; (3) an oblivious sort on an array of size $\#\mathbb{Q}_{\mathsf{DS}} + 2 \cdot \lambda$; and (4) $\#\mathbb{Q}_{\mathsf{DS}_\lambda}$ adds or edits on EDS. The complexity of steps (1) and (2) is

$$\lambda \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}}.$$

The complexity of steps (3) and (4) depend on the sorting network used and the storage at the client. Using Batcher's bitonic sort [5] with $O(1)$ client storage [27], steps (3) and (4) have complexity

$$O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda} + \#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{u \in \mathbb{U}} \mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|)\right), \quad (1)$$

where $\mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|)$ is either the add or the edit complexity of Σ_{DS}, $\mathbb{Q}_{\mathsf{DS}_\lambda}$ is the query space of DS_λ, and $\mathbb{R}_{\mathsf{DS}_\lambda}$ is the corresponding response space for the queries $q \in \mathbb{Q}_{\mathsf{DS}_\lambda}$. Note that if $\max_{u \in \mathbb{U}} \mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|) = O\left(\log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right)$, then Equation (1) above is

$$O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right).$$

Adding steps (1) through (4) we have

$$\mathsf{time}^{\mathsf{dds}}_{\mathsf{R}} = \lambda \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \quad (2)$$

Operations & rebuild. It follows from the above that the time $\mathsf{time}^{\mathsf{ds}}_{\lambda\mathsf{O}+\mathsf{R}}$ to execute λ operations and to rebuild the structure is

$$\begin{aligned} \mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= \lambda \cdot \mathsf{time}^{\mathsf{dds}}_{\mathsf{O}} + \mathsf{time}^{\mathsf{dds}}_{\mathsf{R}} \\ &= \lambda \cdot \left(\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + 2 \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}}\right) + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \end{aligned} \tag{3}$$

The complexity above depends in part on the efficiency of the scheme Σ_{DX} used for the underlying cache. Several constructions can be used including the "standard" cache, square-root ORAM or the more efficient tree-based ORAM [43]. In the following, we analyze the complexity of Σ_{DDS} based on different instantiations of Σ_{DX}.

Using the standard cache. The standard (zero-leakage) cache is an array of size λ that stores encryptions of label/value pairs (ℓ, v) where the labels all have the same size and where the values are padded to the maximum value length. To execute a get for a label ℓ, the client retrieves the entire encrypted array, decrypts it and keeps the value associated with ℓ. To insert or edit a label/value pair, the client retrieves the entire encrypted array, decrypts it, inserts the new pair or modifies an existing pair, re-encrypts the array and sends it back to he server. It follows that the get and put complexities of the standard cache are

$$\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} = \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}} = O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w\right),$$

Combining this with Equation (3), we have

$$\begin{aligned} \mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\lambda^2 \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w)\right) \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \end{aligned}$$

Using a tree-based cache. The scheme Σ_{DX} can also be instantiated with a tree-based ORAM like Path ORAM [43] which has get and put complexity

$$\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} = \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}} = O\left(\max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right),$$

where λ is the number of entries stored in the ORAM. Combining this with Equation 3, we have

$$\begin{aligned} \mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right) \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \end{aligned} \tag{4}$$

Comparison to black-box ORAM simulation. With the exception of the construction of [33], ORAM does not traditionally support re-sizing. So to compare our constructions with black-box ORAM simulation based on state-of-the-art ORAMs (e.g., Path ORAM [43]) [3] we have to assume that the ORAM is initialized with some upper-bound on the size. We use an "upper-bound" data structure which we denote DS^*. More precisely, to setup the ORAM simulation for a structure DS, the ORAM is initialized to hold DS^* so that DS can expand to fill the allocated space. The ORAM simulation of one operation on DS using a tree-based ORAM then has complexity,

$$\mathsf{time}^{\mathsf{tree}}_{\mathsf{O}} = \mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} \cdot O\left(\log^2 \frac{|\mathsf{DS}^*|_2}{B}\right) \cdot \frac{B}{w},$$

where $\mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}}$ is the number of blocks that need to be read to answer a query, B is the block size of the ORAM and w is the word length (in bits). Since the ORAM does not have to be rebuilt, $\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}+\mathsf{R}}$ is the same as the time complexity of λ operations. Setting $B = \max_{r \in \mathbb{R}_{\mathsf{DS}^*}} |r|_2$ as an upper limit on possible response length, we have,

$$\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}} = \lambda \cdot \mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} \cdot O\left(\log^2 \frac{|\mathsf{DS}^*|_2}{\max_{r \in \mathbb{R}_{\mathsf{DS}^*}} |r|_2}\right) \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}^*}} |r|_w. \tag{5}$$

To compare the efficiency of our schemes with black-box ORAM simulation, we examine Equation (4). Assuming that $\lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$,[4] and $\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} = O(\log \#\mathbb{Q}_{\mathsf{DS}})$ we have that $\#\mathbb{Q}_{\mathsf{DS}_\lambda} \leq \#\mathbb{Q}_{\mathsf{DS}} + \lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$. Combining the first two terms in Equation (4) we get,

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} = O(\#\mathbb{Q}_{\mathsf{DS}} \cdot \log \#\mathbb{Q}_{\mathsf{DS}}) + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right) \\ + O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}}\right).\end{aligned} \tag{6}$$

From Equation (6), we observe that $\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}}$ is asymptotically dominated by

$$O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}}\right).$$

Comparing Equations (5) and (6), we have the following proposition.

Proposition 1. *If* $\lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$, $\#\mathbb{Q}_{\mathsf{DS}} = O(\#\mathbb{Q}_{\mathsf{DS}^*})$ *and* $\mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} = \omega(1)$, *then*

$$\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} = o\left(\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}}\right).$$

[3] Note that some ORAM constructions can achieve better asymptotic query complexity [38] but we use Path ORAM for its simplicity and real-world practicality.

[4] This is a conservative assumption on λ. In practice, the selection of λ is crucial to the efficiency of the scheme. The question of selecting the optimal λ for efficiency is interesting and can be further explored.

For structures with constant-time queries, $\mathrm{B}_{\mathsf{Q}}^{\mathsf{ds}} = 1$ so our approach improves asymptotically over ORAM simulation whenever

$$\max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w = o\left(\max_{r \in \mathbb{R}_{\mathsf{DS}^*}} |r|_w\right).$$

For a concrete efficiency comparison we refer the reader to Section 5.3.

5 Concrete Instantiations

In this section we show how to apply our framework to two concrete schemes: the piggyback scheme PBS from [27] which is a semi-dynamic construction and the advanced volume-hiding scheme AVLH^d from [26] which is mutable. The leakage profiles of the resulting schemes is minimal and only reveal information pertaining to the total size of the structure.

5.1 Our PBS-Based Constructions

PBS is a non-rebuildable semi-dynamic STE scheme. It is parameterized with a batch size α and supports query and add operations. PBS queries and adds in batches in the sense that when executing a query q_1 it only retrieves a fixed number of batches from q_1' s response and retrieves the next set of batches only when a new query q_2 occurs. In the meantime, q_2 is inserted into a queue until enough queries are made for the client to retrieve q_1's entire response. Adds are handled similarly. When a sequence of queries or adds is complete, all the remaining batches in the queue are retrieved or pushed.

PBS has two variants. The first is a perfectly correct variant which incurs some small amount of query leakage; namely, for sequences of non-repeating queries, it leaks the number of batches required to process the sequence; and for sequences with repeating queries, it reveals the query equality and the response lengths. The second variant achieves only probabilistic correctness but the non-repeating sub-pattern of its query leakage is $\perp$. The application of our framework to the first variant results in a dynamic variant of the AZL construction from [27] whereas applying it to the second variant results in a dynamic variant of the FZL construction from [27].

Leakage profile of PBS. The leakage profile of the perfectly correct variant of PBS is

$$\Lambda_{\mathsf{PBS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{A}}) = (\mathsf{tbrlen}, \mathsf{rqeq}, \mathsf{alen}),$$

where tbrlen, rqeq and alen are defined as follows. The *total batched response length*

$$\mathsf{tbrlen}_{k,\alpha}(\mathsf{DS}) = \mathsf{trlen}(\mathsf{DS}) + \sum_{q \in \mathbb{Q}_{\mathsf{DS}}} \alpha - \left(|\mathsf{qu}(\mathsf{DS}, q)|_w \mod \alpha\right)$$

reveals the number of batches needed to store the responses in the structure. The *repeated query equality* pattern

$$\mathsf{rqeq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } m < t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \gamma_m & \text{if } m = t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{qeq} \times \mathsf{rlen}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise,} \end{cases}$$

where

$$\gamma_m \overset{def}{=} \left(\sum_{i \in [m]} |\mathsf{qu}(\mathsf{DS}, q_i)|_w + \alpha - \big(|\mathsf{qu}(\mathsf{DS}, q_i)|_w \mod \alpha\big) \right) \cdot \alpha^{-1} - (m-1).$$

Note that the non-repeating sub-pattern of rqeq is uniq where

$$\mathsf{uniq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } m < t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \gamma_m & \text{if } m = t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t]. \end{cases}$$

The *add length* pattern

$$\mathsf{alen}_{k,m}(\mathsf{DS}, u_1, \ldots, u_t) = \begin{cases} \perp & \text{if } m < t, \\ \gamma_m & \text{if } m = t, \end{cases}$$

reveals nothing until the last add of the sequence, and then reveals the number of batches required to finish the add sequence.

When PBS is modified to support only probabilistic correctness for queries, the non-repeating sub-pattern of its query leakage is $\perp$. The leakage profile of the probabilistic variant of PBS is therefore $(\mathcal{L}_\mathsf{S}^\mathsf{pbs}, \mathcal{L}_\mathsf{Q}^\mathsf{pbs}, \mathcal{L}_\mathsf{U}^\mathsf{pbs}) = (\mathsf{tbrlen}, \mathsf{patt}_\mathsf{Q}, \mathsf{alen})$ where

$$\mathsf{patt}_\mathsf{Q}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{qeq} \times \mathsf{rlen}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise.} \end{cases}$$

Safe extension for PBS. Let $(\widetilde{q}_1, \cdots, \widetilde{q}_\lambda)$ be dummy queries. For all $1 \leq i \leq \lambda$, compute $\overline{\mathsf{DS}} \leftarrow \mathsf{Add}(\overline{\mathsf{DS}}, (\widetilde{q}_i, \mathbf{0}))$, where $|\mathbf{0}|_w = \max_{r \in \mathbb{R}_\mathsf{DS}} |r|_w$.

Theorem 3. *If λ and α are publicly-known parameters and if all queries in $\mathbb{Q}_\mathsf{DS}$ have the same bit length, the extension scheme described above is* $(\mathsf{tbrlen}, \mathsf{uniq}, \mathsf{alen})$*-safe.*

Dynamic AZL. Let dynamic AZL be the perfectly-correct fully-dynamic rebuildable scheme that results from applying our framework to the perfectly-correct variant of PBS. Its security is stated in the following Theorem whose proof is in the full version.

Theorem 4. *If Σ_DX is Λ_DX-secure where $\Lambda_\mathsf{DX} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{G}, \mathcal{L}_\mathsf{P}) = (\mathsf{mllen}, \perp, \perp)$, then dynamic AZL is Λ_AZL-secure where*

$$\begin{aligned} \Lambda_\mathsf{AZL} &= (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{O}, \mathcal{L}_\mathsf{R}) \\ &= \big((\mathsf{tbrlen}, \mathsf{mllen}), \mathsf{uniq}', (\mathsf{lsize}, \mathsf{tbrlen}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otbrlen})\big) \end{aligned}$$

where $\mathsf{otbrlen}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_\lambda) = \mathsf{tbrlen}_{k,\alpha}(\mathsf{DS}_\lambda)$ *and*

$$\mathsf{uniq}'_{k,m}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \mathsf{uniq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t),$$

where op_i *is either a query* q_i *or an update* $u_i = (q_i, r_i)$.

Efficiency of dynamic AZL. It follows from Equation (4) that the complexity of dynamic AZL when Σ_{DX} is initialized with a tree-based ORAM is

$$\mathsf{time}^{\mathsf{azl}}_{\lambda\mathsf{O+R}} = (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}} + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right)$$
$$+ O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda} |r|_w} \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right),$$

where $\mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}}$ is the query complexity of PBS which is equal to the query complexity of is underlying multi-map encryption scheme. The storage complexity of dynamic AZL is the sum of the storage required for the cache and the storage required for the PBS structure. This results in storage complexity

$$O\bigg(\lambda \cdot (\alpha + \max_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)} |a|_w) + \#\mathbb{Q}_{\mathsf{DS}} \cdot (\alpha + \max_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w)\bigg).$$

Dynamic FZL. Dynamic FZL is the probabilistically-correct fully-dynamic scheme that results from applying our framework to the probabilistically-correct variant of PBS. Its security is analyzed in the following Theorem whose proof is in the full version of this work.

Theorem 5. *If* Σ_{DX} *is* Λ_{DX} *where* $\Lambda_{\mathsf{DX}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{P}}) = (\mathsf{mllen}, \perp, \perp)$*, then dynamic* FZL *is* Λ_{FZL}*-secure where*

$$\Lambda_{\mathsf{FZL}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = ((\mathsf{tbrlen}, \mathsf{mllen}), \perp, (\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otbrlen}))\,.$$

Efficiency of dynamic FZL. The efficiency of dynamic FZL is the same as that of dynamic AZL.

5.2 Our AVLH-Based Construction

We now apply our framework to the mutable variant of the advanced volume-hiding multi-map encryption scheme AVLH^d from [26]. Note that here we do not consider the variant that exploits concentrated components for storage improvements.

Overview of AVLH. At a high level, the scheme uses n bins to store a multi-map of size N, where N is the sum over all labels of the labels' tuple lengths. The scheme uses a random bipartite graph to map labels to bins. More precisely, each label ℓ is mapped at random to t out of n bins, where t is the maximum tuple length. The elements of the tuple corresponding to a label ℓ are placed in each bin mapped to ℓ. If there are more bins mapped than the length of the tuple, some bins are left empty. The bins are then padded to the size of the maximum bin, encrypted and stored on the server. To query for a label ℓ, the client retrieves all the bins mapped to ℓ. The scheme hides the tuple lengths, i.e., the response length rlen. It also supports restricted edits in the sense that one can edit/change the values in a tuple but not add values to it. The leakage profile of AVLH^d is

$$\Lambda_{\mathsf{AVLH}} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{E}) = (\mathsf{trlen}, \mathsf{qeq}, (\mathsf{oid}, \mathsf{uqeq})).$$

Extension. Let $(\widetilde{q}_1, \cdots, \widetilde{q}_\lambda)$ be dummy queries and $(\widetilde{r}_1, \cdots, \widetilde{r}_\lambda)$ be the corresponding dummy responses such that $|\widetilde{r}_i| = 1$. For all $i \in [\lambda]$, compute $\overline{\mathsf{MM}} \leftarrow \mathsf{Add}(\overline{\mathsf{MM}}, (\widetilde{q}_i, \widetilde{r}_i))$. We state the security of this extension in the Theorem below whose proof is deferred to the full version.

Theorem 6. *If λ is a publicly-known parameter and that all queries in the query space $\mathbb{Q}_{\mathsf{DS}}$ have the same bit length, the above extension scheme is $(\mathsf{trlen}, \perp, (\mathsf{oid}, \mathsf{uqeq}))$-safe.*

Safe placeholder. Since AVLH^d is mutable we define a safe placeholder multi-map $\widetilde{\mathsf{MM}}$. Note that the placeholder must have the following properties:

1. $\widetilde{\mathsf{MM}}$ must have enough space to hold the tuples of all the labels $\ell \in \mathbb{L}_{\mathsf{MM}_\lambda}$[5];
2. the setup, query and edit leakages on $\widetilde{\mathsf{MM}}$ must be at most the setup, query and edit leakages on MM.

The placeholder structure is created as follows during rebuilds. During the extract-and-tag phase, the client learns which labels are valid and their tuple lengths. During the update phase it creates, for every valid label ℓ a dummy tuple $\mathbf{t}$ of the same length and inserts $(\ell, \mathbf{t})$ in $\widetilde{\mathsf{MM}}$.We state the security of the placeholder in the Theorem below, whose proof is deferred to the full version.

Theorem 7. *The placeholder above is $(\mathsf{trlen}, \mathsf{qeq}, (\mathsf{oid}, \mathsf{uqeq}))$-safe.*

Zero-leakage advanced volume-hiding. Let ZAVLH be the dynamic rebuildable multi-map encryption scheme that results from applying our framework to AVLH^d with the above placeholder structure and a dictionary encryption scheme Σ_{DX} with leakage profile $\Lambda_{\mathsf{DX}} = (\mathcal{L}^{\mathsf{dx}}_\mathsf{S}, \mathcal{L}^{\mathsf{dx}}_\mathsf{G}, \mathcal{L}^{\mathsf{dx}}_\mathsf{P}) = (\mathsf{mllen}, \perp, \perp)$. Theorem 8 below, whose proof is in the full version of this work, states the security of ZAVLH.

Theorem 8. *If Σ_{DX} is Λ_{DX}-secure, then ZAVLH is Λ_{ZAVLH}-secure where*

$$\Lambda_{\mathsf{ZAVLH}} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{O}, \mathcal{L}_\mathsf{R}) = ((\mathsf{trlen}, \mathsf{mllen}), \perp, (\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otrlen})).$$

Efficiency of ZAVLH. We now analyze the efficiency of our dynamic cache-based compiler with a tree-based cache and the AVLH^d scheme. The query complexity for ZAVLH is

$$\mathsf{time}^{\mathsf{zavlh}}_{\mathsf{Q}} = O(t \cdot N/n)$$

[5] For any multi-map data structure MM, the query space $\mathbb{Q}_{\mathsf{DS}}$ is the label space $\mathbb{L}_{\mathsf{MM}}$.

If $t = O(1)$ and $n = O(N/\log N)$ where t is the maximum tuple length and n is the number of bins, the query complexity is $O(\log N)$ for zero-leakage operations. From Equation (4) we have,

$$\mathsf{time}^{\mathsf{zavlh}}_{\lambda\mathsf{O}+\mathsf{R}} = O\left(\#\mathbb{L}_{\mathsf{MM}} \cdot \log N\right) + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{MM}_\lambda}} |r|_w \cdot \log^2 \lambda\right) \tag{7}$$

$$+ O\left(\#\mathbb{L}_{\mathsf{MM}} \cdot \max_{r \in \mathbb{R}_{\mathsf{MM}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{L}_{\mathsf{MM}_\lambda}\right) \tag{8}$$

5.3 Concrete Comparisons

In Section 4.2, we showed that our framework can asymptotically outperform black-box ORAM simulation under natural assumptions on the data and queries. In this section, we are interested in gaining a better understanding of the practical gains in different settings. Specifically, we compare the concrete efficiency of our ZAVLH scheme to an oblivious multi-map constructed via black-box ORAM simulation and to a standard dynamic encrypted multi-map called $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$ [11]. Since the latter has optimal storage and query complexities, this comparison highlights the cost of leakage suppression.

Parameters and notation. For our comparison, we consider a multi-map MM with t labels and $N = \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$ total values and maximum tuple length l. After λ Add operations on MM, the resulting multi-map is denoted MM_λ. We denote the number of labels in MM_λ as t_λ and the total values in MM_λ as N_λ. The maximum tuple size in MM_λ is denoted by l_λ. All PRF keys and outputs are of length $k = 256$ bits, all values in the multi-maps are 64 bits and N is set to 2^{16}.

Parameters for ZAVLH. The number of bins in AVLH are chosen such that each bin contains $(\log N)/2$ values on average. The tree-based cache used in the dynamic CBC is instantiated with Path ORAM with λ leaf nodes; one for each tuple in the cache. Each block is initialized to hold one tuple and therefore $(l + \lambda)$ values at most. Each node/bucket in the binary tree holds $Z = 5$ blocks. The position map maps every label to a leaf node in the ORAM and has size $\lambda(k + \log \lambda)$. The stash stores at most $\log \lambda$ blocks and therefore $\log \lambda(l + \lambda)$ values. A query to the cache reads and writes a path of $\log \lambda$ buckets in the tree. The multi-map MM stores $t + \lambda$ labels and $N + \lambda$ total values. We summarize the cost of ZAVLH in Table 2 breaking it down into the cost to execute λ operations (OPS) and the costs of the different rebuild phases: extract-and-tag (E&T), sort-and-shuffle (S&S) and update (UP).

Black-box ORAM simulation. To manage the dynamic multi-map MM with Path ORAM, we initialize an *upper-bound* structure MM_* with t_* labels and N_* values.[6] Specifically, we use upper-bound structures that are $25, 50, 150,$ and 1000 times larger than the multi-map's original size (Table 1). The maximum length of a tuple in MM_* is l_*. The Path ORAM that manages MM_* has t_*

[6] This is due to Path ORAM's inability to resize.

Parameters	**Setting 1**	**Setting 2**	**Setting 3**	**Setting 4**
General:				
length of PRF output (bits)	256	256	256	256
length of MM value (bits)	64	64	64	64
cache size (λ)	64	64	64	64
MM:				
max. tuple length (l)	512	512	512	512
total # of labels (t)	256	256	256	256
total # of values (N)	2^{16}	2^{16}	2^{16}	2^{16}
total # of AVLH bins (n)	8192	8192	8192	8192
Updated MM_λ:				
max. tuple length (l_λ)	512	512	512	512
total # of labels (t_λ)	256	256	256	256
total # of values (N_λ)	65600	65600	65600	65600
total # of AVLH bins (n_λ)	8199	8199	8199	8199
Upper-bound MM_*:				
factor of growth	25	50	150	1000
max. tuple length (l_*)	1.28×10^4	2.56×10^4	7.68×10^4	51.2×10^4
total # of labels (t_*)	0.64×10^4	1.28×10^4	3.84×10^4	25.6×10^4
total # of values (N_*)	163.84×10^4	327.68×10^4	983.04×10^4	6553.6×10^4

Table 1. Parameters for the efficiency comparison of dynamic CBC, black-box ORAM simulation, and $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$, given a multi-map MM and a sequence of λ add operations.

Efficiency Measure	ZAVLH (OPS)	ZAVLH (E&T)	ZAVLH (S&S)	ZAVLH (UP)	ZAVLH (Total)	Path ORAM EMM$^{(*)}$	Std EMM ($\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$)
Client State (Mbits)	0.401	0.084	-	0.401	0.486	4.78 10.058 32.539 244.137	0.066
Server Storage (Mbits)	29.704	14.352	-	29.71	44.062	52424.704 209707.008 1887412.224 83885916.16	20.992
Communication (Mbits)	166.739	211.042	1181.008	268.294	1827.084	1995.534 4306.721 14421.059 113419.012	10.485
Leakage	l, N	t	t_λ	l_λ, N_λ	l, N, t $l_\lambda, N_\lambda, t_\lambda$	l_*, t_*	vol, qeq

Table 2. Concrete efficiency comparison. The efficiency numbers shown for ORAM correspond to each of the 4 settings for the ORAM upper-bound data structure.

leaf nodes, one for each label in MM_*. Each block is initialized to hold l_* values and each node/bucket in the binary tree holds $Z = 5$ blocks. This ORAM has a position map of size $t_*(q + \log t_*)$ and a stash that holds at most $\log t_*$ blocks at any given time.

Comparison. Table 2 shows the costs in Mbits for each of the 4 settings for ZAVLH, black-box ORAM simulation, and $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$. We can see that ZAVLH outperforms black-box ORAM simulation in both space and communication for our chosen parameters. In particular, the storage cost of ZAVLH is 3 to 7 orders of magnitude smaller than black-box ORAM simulation and only a factor of 2 larger than $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$. We also observe that the communication cost of ZAVLH is up to 60 times smaller than black-box ORAM simulation, but 180 times larger than $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$ which is optimal but incurs more leakage.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An o(n log n) sorting network. In *ACM Symposium on Theory of Computing (STOC '83)*, pages 1–9, 1983.
2. G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.
3. G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In D. Wichs and Y. Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1101–1114. ACM, 2016.
4. G. Asharov, G. Segev, and I. Shahaf. Tight tradeoffs in searchable symmetric encryption. In *Annual International Cryptology Conference*, pages 407–436. Springer, 2018.
5. K. Batcher. Sorting networks and their applications. In *Proceedings of the Joint Computer Conference*, pages 307–314, 1968.
6. L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.
7. R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.
8. R. Bost and P.-A. Fouque. Thwarting leakage abuse attacks against searchable encryption – a formal approach and applicaitons to database padding. Technical Report 2017/1060, IACR Cryptology ePrint Archive, 2017.
9. 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.
10. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.
11. D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.
12. 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.
13. D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

14. 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.
15. 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.
16. I. Demertzis, D. Papadopoulos, and C. Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 371–406. Springer, 2018.
17. S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.
18. 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.
19. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
20. M. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*, pages 95–100, 2011.
21. P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 315–331. ACM, 2018.
22. P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1067–1083. IEEE, 2019.
23. M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.
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 T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.
26. S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *Advances in Cryptology - Eurocrypt' 19*, 2019.
27. S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.
28. S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.
29. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.
30. G. Kellaris, G. Kollios, K. Nissim, and A. O. Neill. Generic attacks on secure outsourced databases. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

31. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*, pages 143–156, 2012.
32. M. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 297–314. IEEE Computer Society, 2018.
33. T. Moataz, T. Mayberry, E. Blass, and A. H. Chan. Resizable tree-based oblivious RAM. In R. Böhme and T. Okamoto, editors, *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *Lecture Notes in Computer Science*, pages 147–167. Springer, 2015.
34. M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.
35. M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.
36. R. Ostrovsky and V. Shoup. Private information storage. In *ACM Symposium on Theory of Computing (STOC '97)*, pages 294–303, 1997.
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. S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.
39. S. Patel, G. Persiano, K. Yeo, and M. Yung. Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing. In *Conference on Computer and Communications Security (CCS '19)*, pages 79–93, 2019.
40. E. Shi, T.-H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)¡sup¿3¡/sup¿) worst-case cost. In *Advances in Cryptology - ASIACRYPT '11*, pages 197–214. Springer-Verlag, 2011.
41. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.
42. E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.
43. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security (CCS '13)*, 2013.
44. X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.
45. P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS '08)*, pages 139–148, 2008.
46. 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.