Structured Encryption and Leakage Suppression

Seny Kamara[1], Tarik Moataz[1], and Olya Ohrimenko[2]

[1] Brown University, Providence, USA
seny@brown.edu, tarik_moataz@brown.edu
[2] Microsoft Research, Cambridge, UK
oohrim@microsoft.com

Abstract. Structured encryption (STE) schemes encrypt data structures in such a way that they can be privately queried. One aspect of STE that is still poorly understood is its leakage. In this work, we describe a general framework to design STE schemes that do not leak the query/search pattern (i.e., if and when a query was previously made).
Our framework consists of two compilers. The first can be used to make any dynamic STE scheme *rebuildable* in the sense that the encrypted structures it produces can be rebuilt efficiently using only $O(1)$ client storage. The second transforms any rebuildable scheme that leaks the query/search pattern into a new scheme that does not. Our second compiler is a generalization of Goldreich and Ostrovsky's square root oblivious RAM (ORAM) solution but does not make use of black-box ORAM simulation. We show that our framework produces STE schemes with query complexity that is asymptotically better than ORAM simulation in certain (natural) settings and comparable to special-purpose oblivious data structures.
We use our framework to design a new STE scheme that is "almost" zero-leakage in the sense that it reveals an, intuitively-speaking, small amount of information. We also show how the scheme can be used to achieve zero-leakage queries when one can tolerate a probabilistic guarantee of correctness. This construction results from applying our compilers to a new STE scheme we design called the *piggyback scheme*. This scheme is a general-purpose STE construction (in the sense that it can encrypt any data structure) that leaks the search/query pattern but hides the response length on non-repeating queries.

1 Introduction

A structured encryption (STE) scheme encrypts data in such a way that it can be privately queried. An STE scheme is secure if it does not reveal any partial information about the data or query beyond a given leakage profile. Special cases of STE include searchable symmetric encryption (SSE) [37,17,13,26,25,8,38] and graph encryption [10,29]. STE has received attention due to its applications to the design of secure cloud services, secure databases, lawful surveillance [22] and network provenance [42]. In recent years, a lot of progress has been made on improving various characteristics of STE including its efficiency [13], its dynamism [26,25,32,7], its parallelism and locality [25,7,9,3,14], its security [13,38,5] and its expressiveness [10,8,34,15,24].

One aspect that is still poorly understood, however, is its leakage. In the context of SSE, we currently know of four attacks. All of these attacks are query-recovery attacks in the sense that they aim to recover information about the queries. The IKK attack [20] exploits co-occurrence leakage (i.e., how often each pair of queries occur together in a document) assuming knowledge of the client's data collection. The Count attack [6] exploits co-occurrence and response length leakage (i.e., how many documents contain the query) assuming knowledge of the client's data collection and of a subset of its queries. [3] The LZWT attack [28] exploits search pattern leakage. File injection attacks [41] are query-recovery attacks where the adversary needs the ability to inject documents/files.

Oblivious RAM (ORAM). One approach that is often suggested for handling leakage is to avoid STE completely and use one of two ORAM-based approaches. The first, which we refer to as ORAM simulation, is to store the data (represented as an array) in an ORAM and query it by simulating every read and write operation of the query algorithm with an ORAM access. Note that this approach is general-purpose. The second approach is to design a custom oblivious data structure and query it with a dedicated oblivious query algorithm. We briefly note that while ORAM simulation is often cited as a zero-leakage (ZL) solution, [4] its exact query leakage actually depends on the data structure being managed. More precisely, ORAM simulation is only ZL for structures with constant query complexity. For structures that do not satisfy this constraint (e.g., inverted indexes) some form of padding must be applied which increases both the storage and query complexity of the solution.

Leakage suppression. Another direction, which we initiate here, is to focus on designing general tools and techniques to *suppress* the leakage of existing schemes. We focus mainly on two kinds of techniques: *compilers*, which take schemes with a given leakage profile and produce new schemes with an improved profile; and *transforms*, which modify queries and/or data in such a way that they can be safely used with schemes that have a certain leakage profile. Our goal is to find compilers and transforms for as wide a class of schemes as possible and that incur the smallest overhead possible. In this work, we propose a leakage suppression framework (i.e., a set of compilers and transforms) for *query equality* leakage. The query equality, which is typically referred to as the search/query pattern in the encrypted search literature, reveals if and when a query has occurred in the past. Interestingly, our main compiler is a generalization of Goldreich and Ostrovsky's square-root ORAM solution [18] but uses STE to avoid ORAM simulation.

Our leakage suppression framework—which combines both STE and ORAM—can result in STE schemes that are asymptotically more efficient than ORAM

[3] It was shown experimentally in [6] that the IKK and Count attacks need to know at least 90% and 75% of the client's data, respectively. In addition, the Count attack also needs to know at least 5% of the client's queries whenever it knows less than 100% of the client's data.

[4] In this work, a solution is ZL if its leakage reveals only information that is derived from the security parameter or other public parameters.

simulation under certain assumptions on the data and queries which we make precise in Section 8.[5] We also find that these schemes can achieve the same asymptotic efficiency as custom oblivious data structures (specifically, we compare to the case of oblivious trees). While we focus here on query equality, suppression frameworks for other common leakage patterns would be of interest.

1.1 Our Contributions and Techniques

In this work, we consider leakage suppression techniques focusing on query equality. We make several contributions which we summarize below.

Modeling leakage. Because the terminology and formalism used in previous work is sometimes inconsistent and contradictory, we extend the definitional approach of [13,10] with a more intuitive nomenclature and precise descriptions. The details are in Section 5.1 but, as an example, we mention that the search/query pattern is referred to as the *query equality* pattern in our framework and is modeled as a function $\mathsf{qeq} : \mathbb{D} \times \mathbb{Q}^t \to \{0,1\}^{t \times t}$, where $\mathbb{D}$ is a space of data objects, $\mathbb{Q}^t$ is a sequence of queries from a query space $\mathbb{Q}$, and $\{0,1\}^{t \times t}$ is the set of binary $t \times t$ matrices. The function qeq takes a data object and a query sequence and outputs a binary matrix with a 1 at location (i, j) if the i^{th} and j^{th} queries in the sequence are equal and 0 otherwise. We also identify and formalize the notion of *sub-pattern* leakage which captures the behavior of a leakage pattern on a specified subset of query sequences. As we will see, sub-patterns are important in understanding and analyzing our suppression techniques.

Reinterpreting the square-root solution. Our main suppression compiler is based on the seminal square-root ORAM solution of Goldreich and Ostrovsky [18] which works as follows. Items are encrypted and stored in a main memory together with encrypted dummy items after being randomly shuffled. In addition, a cache is maintained in which encrypted items are moved after being accessed. The ORAM structure consists of the main memory and the cache. Reading from the ORAM requires accessing the entire cache to look for the item and retrieving from main memory either a dummy item if the item was found in the cache, or the real item if the item was not found in the cache.

We observe that the square-root solution can be reinterpreted through the lens of STE as follows: the main memory is an encrypted array that leaks the query equality pattern (since reading the same location twice requires sending the same randomly permuted address to the server) and the cache is an encrypted dictionary with no query leakage. The access protocol can then be understood as a mechanism that leverages the ZL queries of the cache to suppress the query equality leakage of the encrypted array.

The cache-based compiler. As we show, the ideas that underlie the square-root solution are not only applicable to encrypted arrays but can be generalized to more complex constructions like encrypted multi-maps and dictionaries. In other words, instead of using a ZL cache to suppress the query equality leakage

[5] When these assumptions do not apply, the schemes are comparable in efficiency to ORAM simulation.

of an encrypted array (i.e., the main memory) we want to use the cache to suppress the query equality leakage of complex encrypted structures. Though there are technical subtleties that must be addressed when moving to more complex structures we describe and analyze this generalization of the square-root solution which we refer to as the *cache-based compiler* (CBC).

The main advantage of using the CBC to suppress query equality leakage is that we can avoid ORAM simulation; that is, we do not have to represent our data structure as an array and simulate every read and write instruction of the query algorithm with an ORAM access. As we show in Section 8, our framework induces an additive overhead over the optimal query complexity. This is in contrast to ORAM simulation which induces a multiplicative overhead. Comparing the efficiency of the two approaches over arbitrary data and queries, however, is not possible so we show that under certain natural conditions (e.g., known to occur in the keyword search setting), our framework results in schemes that are asymptotically faster than ORAM simulation and comparable to dedicated oblivious data structure constructions (here, we consider the case of oblivious trees). While the CBC allows us to avoid ORAM simulation, our framework can still benefit from improvements in ORAM design. The reason is that while ORAM is not used to manage the main data structure, it can (and should) be used to implement the cache. Note also that the CBC yields a static scheme even though it requires a dynamic ZL dictionary. Designing a dynamic variant of the CBC is left as an important open problem.

Non-repeating sub-patterns. In analyzing the security of the schemes that result from the CBC, we find that their query leakage is a *sub-pattern* of the base scheme's query leakage. Specifically, it is what we refer to as the *non-repeating* sub-pattern which is the leakage that occurs on sequences of non-repeating queries. This suggests that a future goal in STE design might be to focus on schemes with low non-repeating sub-pattern leakage as opposed to focusing on schemes with low query leakage directly.

Safe extensions. As mentioned above, there are several technicalities that must be handled when adapting the square-root solution to more complex structures. The first is that the structure must be extendable in the sense that it must be able to hold and query dummy items. We formalize this process as an *extension* scheme, which takes as input a data structure and outputs a new one with the same items plus a given number of dummy items. While, a-priori, this might seem straightforward, one has to handle dummy items with care because the leakage of the scheme (which was not originally designed to handle dummies) could reveal information that enables the adversary to distinguish between real and dummy items. In addition, the way in which dummy items are handled could be correlated with the real items and this could be revealed to the adversary through the leakage of the scheme. In Section 6.1, we formally define the security properties that extension schemes must satisfy in order to be safely used with the CBC.

The rebuild compiler (RBC). Another challenge is that the CBC requires the base scheme to be efficiently rebuildable, i.e, equipped with an efficient proto-

col that can reconstruct the structure with new randomness. Most STE schemes were not designed with this in mind so we describe a general-purpose protocol that can be used to rebuild any dynamic STE scheme. If the base scheme has $O(\log^2 n)$ update complexity, [6] where n is the number of items stored in the structure, then our protocol has computation and communication complexity $O(n \log^2 n)$. In addition, our rebuild protocol does not affect the latency of the scheme in the sense that queries can still be made and answered while a rebuild is taking place. Note, however, that the output of the RBC is a *static* rebuildable scheme, therefore losing the dynamism of the base construction. The question of designing a variant of the RBC that preserves dynamism is left open.

The piggyback scheme (PBS). As discussed, the CBC results in new constructions that leak the non-repeating sub-pattern of their base scheme. Our goal, therefore is reduced to designing schemes with low non-repeating sub-pattern leakage. In the setting of encrypted arrays, this is relatively straightforward because the base scheme that implicitly underlies the square-root solution (i.e., encrypt and randomly shuffle the items, and fetch by reading the permuted location) does not reveal anything when queried on non-repeating sequences. This is not the case, however, for standard encrypted multi-map or dictionary constructions which reveal the response identity (i.e., the plaintext result of the query) if they are response revealing; or the response length if they are response hiding. In particular, this means that these leakages may persist even after applying the CBC.

To address this we design a new scheme called PBS with low non-repeating sub-pattern leakage. There are two variants of the scheme: one that reveals the total sequence response length (i.e., the sum of the response lengths over all queries in the sequence) and another that reveals nothing. The former achieves standard correctness whereas the latter is correct with only a certain probability. At a high level, PBS results from applying a transform to the data and queries so that they can be safely used with an encrypted multi-map that leaks the response length. Our approach is to modify the data in such a way that, at query time, the client can retrieve a fixed number of words per query (which we refer to as a *batch*) no matter how large the response is. To maintain correctness, incoming queries are queued and processed at the next available time. This introduces a delay in the querying process but by carefully tuning the batch size we can ensure the entire response is retrieved in a reasonable amount of time. PBS is general-purpose in the sense that it encrypts *any* data structure. As far as we know, this is the first general-purpose STE scheme and may be of independent interest.

New constructions. Our framework results in several new schemes. First, by applying our compilers to PBS, we get a new general-purpose STE scheme called AZL that is "almost" ZL. Specifically, when used on a sequence of t queries $(q_1, \ldots, q_t)$, its query leakage reveals nothing on queries $(q_1, \ldots, q_{t-1})$ and then reveals the sum of the sequence's response lengths on query q_t. We then show

[6] As far as we know, all dynamic SSE schemes have update complexity ranging from constant to $O(\log^2 n)$.

that by applying our compilers to a variant of PBS, we can get a "fully" ZL construction at the cost of achieving a weaker notion of correctness. As discussed above, the query complexities of both AZL and its fully ZL variant, FZL, are asymptotically smaller than ORAM simulation under natural assumptions.

Of course, our compilers can also be applied to other constructions with query equality leakage, including schemes for single-keyword and boolean SSE [13,10,8,7,5,2,24], encrypted relational databases [23] or encrypted graphs [10,29]. We stress, however, that the resulting schemes may not be ZL (or even almost ZL) since, as discussed above, our framework suppresses query equality leakage but still reveals the base scheme's non-repeating sub-pattern.

2 Related Work

Structured and searchable encryption. Searchable encryption was first considered explicitly by Song, Wagner and Perrig in [37]. In [13], Curtmola, Garay, Kamara and Ostrovsky introduced adaptive security and proposed the first schemes with optimal search complexity $O(\#\mathsf{DB}[w])$, where $\#\mathsf{DB}[w]$ is the number of documents that contain the keyword w. The notion of structured encryption was introduced by Chase and Kamara [10] as a generalization of SSE that supports queries on arbitrarily-structured data. Subsequent works have considered the problems of dynamic [17,26,25,38,5,7], I/O-efficient [7,30], local [9,3,14], more secure [38,16,5], expressive [10,8,34,15,24], and multi-user [13,21] SSE.

Recently, Garg, Mohassel and Papamanthou [16] presented a dynamic SSE construction that hides the query equality pattern by leveraging ORAM and garbled RAM techniques. Their construction has non-optimal search complexity $\widetilde{O}(\#\mathsf{DB}[w] \cdot \log N + \log^3 N)$, where $N = \sum_{w \in \mathbb{W}} \mathsf{DB}[w]$. We note that while this scheme does not reveal the query equality explicitly, it still leaks the response length which is often correlated with the query equality. Our AZL and FZL constructions, on the other hand, hide the query equality and reveal only the *sequence* response length and $\perp$, respectively. In addition, they achieve this without the multiplicative $\log N$ overhead and with less computation on the client side.

Oblivious RAM. The seminal work of Goldreich and Ostrovsky [18] introduced the notion of ORAM and described the Square-Root and Hierarchical solutions. Many subsequent constructions improved ORAM upon several dimensions including communication complexity, number of rounds, client storage and storage overhead [33,40,19,27,36,39,16].

3 Preliminaries and Notation

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 $x \xleftarrow{\$} X$ 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 $a_i < a_j$ 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 [4] 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*. 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. We model a dynamic data type $\mathbf{T}$ as a collection of four spaces $\mathbb{D} = \{\mathbb{D}_k\}_{k\in\mathbb{N}}$, $\mathbb{Q} = \{\mathbb{Q}_k\}_{k\in\mathbb{N}}$, $\mathbb{R} = \{\mathbb{R}_k\}_{k\in\mathbb{N}}$ and $\mathbb{U} = \{\mathbb{U}_k\}_{k\in\mathbb{N}}$ and two maps $\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}$ and $\mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D}$, where $\mathbb{D}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{U}$ are, respectively, $\mathbf{T}$'s object, query, response and update spaces. In Section 9, we make the additional assumption that $\mathbb{U} = \mathbb{Q} \times \mathbb{R}$, i.e., an update can be written as a pair composed of a query and its response. 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 $\log(n)$ 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,

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

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

Definition 2 (Correctness). *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is 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. Throughout, we will make black-box use of these data types which means that they can be instantiated with any appropriate data structure. To highlight this black-box usage, we refer to the data structure by its type's name. For example, we will write

RAM, DX and MM to refer to some arbitrary array, dictionary and multi-map [7] data structures.

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.

Data structure logs. Given a structure DS that instantiates an object d, we will be interested in the shortest sequence of update operations needed to create a new structure DS' that also instantiates d. We refer to this as the *update log* of DS and assume the existence of an efficient algorithm Log that takes as input DS and outputs a sequence $(u_1, \ldots, u_n)$ 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* 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 λ-extendable, for $\lambda \geq 1$, if there exists a query set $\overline{\mathbb{Q}} \supset \mathbb{Q}$ of size $\#\mathbb{Q} + \lambda$ and a probabilistic polynomial time algorithm $\mathsf{Ext}_{\mathbf{T}}$ that takes as input DS and λ and returns a new structure $\overline{\mathsf{DS}}$ of the same type $\mathbf{T}$ such that: (1) $\overline{\mathsf{DS}} \equiv d$; and (2) for all $q \in \overline{\mathbb{Q}} \setminus \mathbb{Q}$, $\mathsf{Query}(\overline{\mathsf{DS}}, q) = \perp$. [8] 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. We sometimes write $\Pi_{A,\mathcal{A}}$ to denote an execution of Π where the first party follows the protocol and the second party is some adversary $\mathcal{A}$. Similarly we sometimes write $\Pi_{\mathcal{S},\mathcal{A}}$ to denote an execution of Π between a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$. We quantify the round complexity of a protocol in either *moves* (i.e., messages sent between the parties) or *rounds* (i.e., pairs of messages exchanged between the parties).

[7] 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).

[8] Note that we make the implicit assumption that adding dummy queries to the query space of some data type does not change the type.

4 Re-Defining Structured Encryption

An STE scheme can be roughly viewed as a data structuring scheme that works over encrypted data. Several types of STE schemes were described in [11] (the full version of [10]) but here we consider structure-only schemes. This variant only encrypts objects as opposed to standard schemes which encrypt both a data structure and data items (e.g., documents, emails, user profiles). At a high-level, the formulation proposed in [10] works as follows. During a setup phase, the client constructs an encrypted data structure EDS under a key K. The client then sends EDS to the server. During the query phase, the client constructs and sends a token tk generated from its query q and secret key K. The server then uses the token tk to query EDS and recover a response r. Below, we formally describe our notion of STE. Our definition generalizes that of [10] in several respects.

Interaction. In the standard variant of STE, the query phase is *non-interactive*; that is, it requires only a single round that consists of the client sending a token and the server returning an encrypted data item. All the constructions proposed in [10] are non-interactive and many SSE constructions are as well. There are, however, several constructions that are interactive including [35,25,8]. The use of interaction in STE provides a lot of power and most interactive constructions are able to improve on the leakage of non-interactive schemes. For example [25] uses interaction during the update phase to leak less than [26], and [8] uses interaction to leak less than the naive boolean SSE construction which consists of the server taking intersections and unions of results.

Rebuilding. Since previous notions of STE did not consider rebuilding, the standard security notions of [13,10] have to be augmented appropriately. In particular, the definition has to properly capture the effect of rebuilding operations on the security of the scheme. Functionally, the result of rebuilding an encrypted structure EDS should be equivalent to re-running the scheme's Setup algorithm (with new coins) on the structure underlying EDS. From a security perspective, the purpose of rebuilding is to reduce the scheme's leakage.

4.1 Syntax and Correctness

In Definition 3 below we extend the syntax of STE to include interactive operations and rebuilding. We do this by adding an additional protocol for rebuilding operations. When using data structures, it is sometimes convenient to build a structure with a Setup operation that takes as input a data object. Other times, it is more convenient to build an empty structure with an Init operation and add items subsequently. Here, we only define a Setup algorithm but capture Init operations by inputting an empty structure $\mathsf{DS}_0 \equiv d_0$.

Definition 3 (Structured encryption). *A type-***T** *interactive structured encryption scheme* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Update}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *consists of an algorithm and three two-party protocols 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*. We sometimes write this as* $\mathsf{Setup}(1^k, \lambda, \perp)$*.*

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

For visual clarity, we sometimes omit the subscripts of the protocols when the parties involved are clear from the context.

We say that a type-$\mathbf{T}$ encrypted structure EDS instantiates a data object $d \in \mathbb{D}$ if for all $q \in \mathbb{Q}$, $\mathsf{Query}\big((K, st, q), \mathsf{EDS}\big)$ outputs $\big((st, r), \perp\big)$ such that $r = \mathsf{qu}(d, q)$, where K and st are the key and state of EDS. We write this as $\mathsf{EDS} \equiv d$ and sometimes write $\mathsf{EDS} \equiv \mathsf{DS}$ to mean that EDS and DS instantiate the same data object.

Definition 4 (Correctness). *A type-*$\mathbf{T}$ *structured encryption scheme* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Update}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *is correct if it satisfies the following properties:*

- *(static correctness) for all* $k \in \mathbb{N}$*, for all* $d \in \mathbb{D}$*, for all* DS *that instantiate* d*, for all* $\lambda \geq 1$*,*

$$\Pr\left[\, \mathsf{EDS} \equiv \mathsf{DS} : (K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \lambda, \mathsf{DS}) \,\right] \geq 1 - \mathsf{negl}(k),$$

  where the probability is over the coins of Setup *and of* Query*.*
- *(dynamic correctness) for all* $k \in \mathbb{N}$*, for all* $d \in \mathbb{D}$*, for all* EDS *that instantiate* d*, for all* $u \in \mathbb{U}$*, for all* $\lambda \geq 1$*,*

$$\Pr\left[\, \mathsf{EDS}' \equiv \mathsf{up}(d, u) : (st, \mathsf{EDS}') \leftarrow \mathsf{Update}\big((K, st, u), \mathsf{EDS}\big) \,\right] \geq 1 - \mathsf{negl}(k),$$

  where K *and* st *are the key and state of* EDS *and the probability is over the coins of* Update*.*
- *(rebuild correctness) for all* $k \in \mathbb{N}$*, for all* $d \in \mathbb{D}$*, for all* EDS *that instantiate* d*,*

$$\Pr\left[\, \mathsf{EDS}' \equiv d : \big((st, K'), \mathsf{EDS}'\big) \leftarrow \mathsf{Rebuild}\big((K, st), \mathsf{EDS}\big) \,\right] \geq 1 - \mathsf{negl}(k),$$

  where K *and* st *are the key and state of* EDS *and the probability is over the coins of* $\mathsf{Rebuild}$*.*

Structured encryption variants. The syntax of the different variants of STE can all be recovered from Definition 3. Stateless schemes can be recovered by omitting the state from the inputs and outputs of the algorithms. Schemes with non-interactive queries and/or updates can be recovered by requiring that the $\mathsf{Query}_{\mathbf{C},\mathbf{S}}$ or $\mathsf{Update}_{\mathbf{C},\mathbf{S}}$ protocols have only one message referred to as the search and update tokens, respectively. Response-revealing schemes have the following query syntax $\big(st, r\big) \leftarrow \mathsf{Query}_{\mathbf{C},\mathbf{S}}\big((K, st, q), \mathsf{EDS}\big)$.

5 Defining Security

As discussed in the previous Section, the standard notion of security for STE guarantees that an encrypted structure reveals no information about its underlying structure beyond the setup leakage $\mathcal{L}_{\mathsf{St}}$, that the query protocol reveals no information about the structure and queries beyond the query leakage $\mathcal{L}_{\mathsf{Qr}}$, and that the update protocol reveals no information about the structure and updates beyond the update leakage $\mathcal{L}_{\mathsf{Up}}$. If this holds for non-adaptively chosen operations then this is referred to as non-adaptive semantic security. If, on the other hand, the operations are chosen adaptively, this leads to the stronger notion of adaptive semantic security [13]. This notion of security was first proposed and formalized by Curtmola *et al.* in the context of SSE [13] and later generalized to STE in [10].

5.1 Modeling Leakage

We use the approach of [13,10] to capture leakage in STE. Every STE operation is associated with leakage which itself can be composed of multiple *leakage patterns*. The collection of all of these leakage functions is the scheme's *leakage profile*. Leakage patterns are (families of) functions over the various spaces associated with the underlying data type.

Leakage patterns. For concreteness, we describe several well-known leakage patterns. Because the terminology used in previous work to describe leakage is very inconsistent, we propose new terminology and nomenclature. Our goal here is to provide a nomenclature for leakage patterns that gives names that are precise, concise, unique and intuitive. We refer to any leakage pattern that reveals an item completely as an *identity pattern*, any leakage pattern that reveals whether two items are equal as an *equality pattern*, any leakage pattern that reveals the size of a set as a *size pattern* and any leakage pattern that reveals the length of an item as a *length pattern*. 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 data type and consider the following leakage patterns:

- the *query equality pattern* is the function family $\mathsf{qeq} = \{\mathsf{qeq}_{k,t}\}_{k,t \in \mathbb{N}}$ with $\mathsf{qeq}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \rightarrow \{0,1\}^{t \times t}$ such that $\mathsf{qeq}_{k,t}(d, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix such that $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ if $q_i \neq q_j$. The query equality pattern is referred to as the search pattern in the SSE literature;
- the *response identity pattern* is the function family $\mathsf{rid} = \{\mathsf{rid}_{k,t}\}_{k,t \in \mathbb{N}}$ with $\mathsf{rid}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \rightarrow \mathbb{R}_k$ such that $\mathsf{rid}_{k,t}(d, q_1, \ldots, q_t) = \big(\mathsf{qu}(d, q_1), \cdots, \mathsf{qu}(d, q_t)\big)$.

The response identity pattern is referred to as the access pattern in the SSE literature;

- the *data identity pattern* is the function family $\mathsf{did} = \{\mathsf{did}_k\}_{k\in\mathbb{N}}$ with $\mathsf{did}_k : \mathbb{D}_k \to \mathbb{D}_k$ such that $\mathsf{did}_k(d) = d$.
- the *response equality pattern* is the function family $\mathsf{req} = \{\mathsf{req}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{req}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{req}_{k,t}(d, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix such that $M[i, j] = 1$ if $\mathsf{qu}(d, q_i) = \mathsf{qu}(d, q_j)$.

Note that the patterns described above can be defined over any data type. Some leakage patterns, however, can only be defined over data types with spaces that have additional structure. As examples, consider the following patterns where we assume that the underlying type is defined over data, query and response spaces that are equipped with "length functions" $|\cdot|_{\mathbb{D}} : \mathbb{D} \to \mathbb{N}$, $|\cdot|_{\mathbb{Q}} : \mathbb{Q} \to \mathbb{N}$ and $|\cdot|_{\mathbb{R}} : \mathbb{R} \to \mathbb{N}$ (we drop the subscripts for visual clarity since the space is clear from the context):

- the *query length pattern* is the function family $\mathsf{qlen} = \{\mathsf{qlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{qlen}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}$ such that $\mathsf{qlen}_{k,t}(d, q_1, \ldots, q_t) = \big(|q_1|, \cdots, |q_t|\big)$;
- the *response length pattern* is the function family $\mathsf{rlen} = \{\mathsf{rlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{rlen}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}$ such that $\mathsf{rlen}_{k,t}(d, q_1, \ldots, q_t) = \big(|\mathsf{qu}(d, q_t)|, \cdots, |\mathsf{qu}(d, q_t)|\big)$;
- the *maximum query length pattern* is the function family $\mathsf{mqlen} = \{\mathsf{mqlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{mqlen}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}$ such that $\mathsf{mqlen}_{k,t}(d, q_1, \ldots, q_t) = \max_{q\in\mathbb{Q}_k} |q|$;
- the *maximum response length pattern* is the function family $\mathsf{mrlen} = \{\mathsf{mrlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{mrlen}_{k,t} : \mathbb{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}$ such that $\mathsf{mrlen}_{k,t}(d, q_1, \ldots, q_t) = \max_{q\in\mathbb{Q}_k} |\mathsf{qu}(d, q)|$;
- the *total response length pattern* is the function family $\mathsf{trlen} = \{\mathsf{trlen}_k\}_{k\in\mathbb{N}}$ with $\mathsf{trlen}_k : \mathbb{D}_k \to \mathbb{N}$ such that $\mathsf{trlen}_k(d) = \sum_{q\in\mathbb{Q}_k} |\mathsf{qu}(d, q)|$;
- the *data size pattern* is the function family $\mathsf{dsize} = \{\mathsf{dsize}_k\}_{k\in\mathbb{N}}$ with $\mathsf{dsize}_k : \mathbb{D}_k \to \mathbb{N}$ such that $\mathsf{dsize}_k(d) = |d|$.

We say that a pattern is ZL if it depends only on the security parameter and other public parameters. Note that this does not imply that no leakage occurred but rather that whatever leakage did occur is not useful since it could have been derived solely from the public parameters. For example, the maximum query length is a ZL pattern since it can be derived from the security parameter. Given some query leakage pattern $\mathsf{patt} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$, we will often abuse notation and write $\mathsf{patt}(\mathsf{DS}, q_1, \ldots, q_t)$ to mean $\mathsf{patt}(d, q_1, \ldots, q_t)$ where $d \equiv \mathsf{DS}$. Similarly, for some setup leakage pattern $\mathsf{patt} : \mathbb{D} \to \mathbb{X}$, we sometimes write $\mathsf{patt}(\mathsf{DS})$ to mean $\mathsf{patt}(d)$ where $d \equiv \mathsf{DS}$. We use the same notation for update and rebuild leakage patterns.

Leakage sub-patterns. Given a leakage pattern patt we can decompose it into sub-patterns that capture its behavior on restricted classes of query sequences. In this work, we are particularly interested in how certain schemes behave when used on non-repeating query sequences—as opposed to arbitrary sequences. We refer to this as patt's *non-repeating sub-pattern*.

Definition 5 (Non-repeating sub-patterns). *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R})$ *be a static data type and* $\mathsf{patt} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$ *be a query leakage pattern. We say that* $\mathsf{nrp} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$ *is* patt*'s non-repeating sub-pattern if there exists some*

function $\mathsf{other} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$ *such that for all* DS *of type* $\mathbf{T}$ *and all sequences* $(q_1, \ldots, q_t) \in \mathbb{Q}^t$,

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

Definition 5 can be extended to any other operation in the natural way.

Operational leakage. Each operation of an STE scheme (e.g., setup, query, update) generates some leakage which is the *direct product* of one or more leakage patterns. As an example, consider the setup and query leakage of typical static SSE schemes (e.g., [13,7]). The setup leakage is $\mathcal{L}_{\mathsf{St}} = \mathsf{trlen}$ and the query leakage is $\mathcal{L}_{\mathsf{Qr}} = (\mathsf{qeq}, \mathsf{rid}) = \mathsf{qeq} \times \mathsf{rid}$. Note that during the Ideal experiment used to formalize SSE security, the simulator will receive $\mathsf{trlen}(\mathsf{DB}) = \sum_{w \in \mathbb{W}} \#\mathsf{DB}(w)$ in order to simulate EDB and

$$\mathsf{qeq} \times \mathsf{rid}(\mathsf{DB}, w_1, \ldots, w_t) = \big(\mathsf{qeq}(\mathsf{DB}, w_1, \ldots, w_t), \mathsf{rid}(\mathsf{DB}, w_1, \ldots, w_t)\big),$$

in order to simulate the t^{th} search token. We say that an operation is ZL if its leakage includes only ZL patterns.

Leakage profiles. A leakage profile is a collection of leakages for a set of operations. For example, the standard leakage profile for static response-revealing SSE schemes like [13,7] is

$$\Lambda_{\mathsf{RR}} = (\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Qr}}) = \Big(\mathsf{trlen}, \Big(\mathsf{qeq}, \mathsf{rid}\Big)\Big).$$

The response-hiding variants of these constructions, however, have leakage profile

$$\Lambda_{\mathsf{RH}} = (\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Qr}}) = \Big(\mathsf{trlen}, \Big(\mathsf{qeq}, \mathsf{rlen}\Big)\Big).$$

Leakage upper bounds. Another useful notion for our purposes is that of a leakage upper bound which allows us to argue that some leakage pattern reveals nothing beyond some other operational leakage.

Definition 6. *Let* patt_1 *and* patt_2 *be two query leakage patterns. We say that* patt_1 *leaks at most* patt_2 *if there exists a probabilistic polynomial time simulator* $\mathcal{S}$ *such that for all probabilistic polynomial time distinguishers* $\mathcal{D}$*, for all* $d \in \mathbb{D}$*, for all* $\mathsf{DS} \equiv d$*, for all* $t \in \mathbb{N}$*, for all sequences* $(q_1, \ldots, q_t) \in \mathbb{Q}^t$*, the following expression is negligible in* k*,*

$$\left|\Pr\left[\mathcal{D}\Big(\mathsf{patt}_1\big(\mathsf{DS}, q_1, \ldots, q_t\big)\Big) = 1\right] - \Pr\left[\mathcal{D}\Big(\mathcal{S}\Big(\mathsf{patt}_2\big(\mathsf{DS}, q_1, \ldots, q_t\big)\Big)\Big) = 1\right]\right|.$$

We write this as $\mathsf{patt}_1 \leq \mathsf{patt}_2$.

Similar notions can be defined for Setup, $\mathsf{Rebuild}$ and Update operations in the natural way.

5.2 Adaptive Semantic Security

In this Section, we extend the notion of adaptive semantic security for STE from [13,10]. Obviously, since we consider interactive Query and Update protocols, we require that the entire interaction between the adversary and the challenger be simulatable (with appropriate leakage) as opposed to just the tokens as is the case in the non-interactive definitions. Also, to capture the effect of rebuilding, the adversary is allowed to execute rebuild operations.

Definition 7 (Adaptive semantic security). *Let* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}_{\mathbf{C},\mathbf{S}}, \mathsf{Update}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ *be a type-*$\mathbf{T}$ *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,* $\Lambda = (\mathsf{patt}_{\mathsf{St}}, \mathsf{patt}_{\mathsf{Qr}}, \mathsf{patt}_{\mathsf{Up}}, \mathsf{patt}_{\mathsf{Rb}})$ *is a leakage profile,* $\lambda \geq 1$ *is a query capacity and* $z \in \{0,1\}^*$*:*

$\mathbf{Real}_{\mathsf{STE},\mathcal{C},\mathcal{A}}(k)$*: given* z *and* λ *the adversary* $\mathcal{A}$ *outputs a structure* DS *of type* $\mathbf{T}$ *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* $t \in [m]$ *the challenger and adversary do the following:*

1. *if* op_t *is a query operation* $q \in \mathbb{Q}$*, they execute* $\mathsf{Query}_{\mathcal{C},\mathcal{A}}\big((K, st, q), \mathsf{EDS}\big)$*;*
2. *if* op_t *is an update operation* $u \in \mathbb{U}$*, they execute* $\mathsf{Update}_{\mathcal{C},\mathcal{A}}\big((K, st, u), \mathsf{EDS}\big)$*;*
3. *if* op_t *is a rebuild operation, 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.*

$\mathbf{Ideal}_{\mathsf{STE},\mathcal{A},\mathcal{S}}(k)$*: given* z *and* λ *the adversary* $\mathcal{A}$ *outputs a structure* DS *of type* $\mathbf{T}$*. Given* $\mathsf{patt}_{\mathsf{St}}(\mathsf{DS})$*, 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* $t \in [m]$*, the challenger, simulator and adversary do the following:*

1. *if* op_t *is a query operation* $q \in \mathbb{Q}$*, they execute* $\mathsf{Query}_{\mathcal{S},\mathcal{A}}\big(\mathsf{patt}_{\mathsf{Qr}}(\mathsf{DS}, q), \mathsf{EDS}\big)$*;*
2. *if* op_t *is an update operation* $u \in \mathbb{U}$*, they execute* $\mathsf{Update}_{\mathcal{S},\mathcal{A}}\big(\mathsf{patt}_{\mathsf{Up}}(\mathsf{DS}, u), \mathsf{EDS}\big)$*;*
3. *if* op_t *is a rebuild operation, they execute* $\mathsf{Rebuild}_{\mathcal{S},\mathcal{A}}\big(\mathsf{patt}_{\mathsf{Rb}}(\mathsf{DS}), \mathsf{EDS}\big)$*.*

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

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

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

Connection to ORAM and PIR. STE captures other primitives like ORAM and PIR. In particular, the syntax and security definitions of both primitives can be recovered from Definition 7 as follows. ORAM can be viewed as an adaptively Λ_{ORAM}-secure array encryption scheme with

$$\Lambda_{\mathsf{ORAM}} = \big(\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Rd}}, \mathcal{L}_{\mathsf{Wr}}\big) = \big(\mathsf{dsize}, \bot, \bot\big),$$

where $\mathcal{L}_{\mathsf{St}}$, $\mathcal{L}_{\mathsf{Rd}}$ and $\mathcal{L}_{\mathsf{Wr}}$ are the setup, read and write leakages. Similarly, PIR can be viewed as an adaptively Λ_{PIR}-secure array encryption scheme where

$$\Lambda_{\mathsf{PIR}} = \big(\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Rd}}\big) = \big(\mathsf{did}, \bot\big).$$

where $\mathcal{L}_{\mathsf{St}}$ and $\mathcal{L}_{\mathsf{Rd}}$ are the setup and read leakages.

6 The Cache-Based Compiler

STE provides a natural way to understand the square-root solution of Goldreich and Ostrovsky [18]. More precisely, the construction consists of two components: a main memory in which the encrypted data and dummy items are stored and a cache in which items are moved after being accessed. Access to this ORAM structure requires constantly accessing the cache to look for the desired item and either retrieving a dummy item (in case the item was in the cache) or the real item from main memory (in case the item was not in the cache).

We observe that this ORAM structure can be viewed through the lens of structured encryption as follows: the main memory is an encrypted array that leaks the query equality pattern and the cache is a ZL encrypted dictionary. The access protocol can then be understood as a mechanism that leverages the ZL property of the cache to suppress the query equality leakage of the encrypted array. We now describe this view in more detail.

A structured view of the square-root solution. We assume familiarity with the square-root solution and refer the reader to [18] for a detailed exposition. Given an array RAM of N items the square-root solution produces a structure $\mathsf{ORAM} = (\overline{\mathsf{ERAM}}, \mathsf{EDX})$ which consists of an encrypted array $\overline{\mathsf{ERAM}}$ and an encrypted dictionary EDX. $\overline{\mathsf{ERAM}}$ is an encryption of a $\sqrt{N}$-extension $\overline{\mathsf{RAM}}$ of RAM, where $\sqrt{N}$ is the capacity with which RAM has been extended. Concretely, it consists of encryptions of the data items in RAM and of $\sqrt{N}$ dummy items all permuted at random.[9] We refer to an item's location in RAM as its virtual address and to its location in $\overline{\mathsf{ERAM}}$ as its real address. To allow for space-efficient rebuilding, the permutation is instantiated by sorting on random tags that are associated to each item and that are generated by evaluating a PRF on the item's virtual address. To access the item with virtual address i, one executes a Read protocol which re-computes the item's random tag and performs a binary search to find it. Since the tags are deterministic the locations accessed by the binary search are also deterministic and, therefore, the Get protocol reveals the query equality (but nothing else since the labels are pseudo-random). The cache simply consists of encryptions of elements of the form $\langle i, v \rangle$, where i is the virtual address of item v. To retrieve the item with virtual address i, one executes a protocol Get which retrieves and decrypts each element of the cache and returns to the client the one with prefix i. The purpose of concatenating virtual addresses i to items v is to allow for retrievals based on virtual address as opposed to based on location in the cache. More abstractly, it instantiates a dictionary with pairs that consist of data items labeled with their virtual address. Finally, by retrieving the entire cache every time a query is made to EDX, we ensure that the Get protocol for EDX is ZL and that nothing is revealed about the query or response.

So we have two structures: $\overline{\mathsf{ERAM}}$, which holds $N + \sqrt{N}$ items (i.e., the real items plus the dummy items) and has query leakage qeq; and EDX, which holds $\sqrt{N}$ items and has query leakage $\perp$. Clearly, accessing $\overline{\mathsf{ERAM}}$ directly more than

[9] Note that after $\sqrt{N}$ queries, the entire ORAM needs to be rebuilt.

once leaks information so the goal is to leverage the obliviousness of EDX to *suppress* the leakage of $\overline{\mathsf{ERAM}}$. At a high-level, Goldreich and Ostrovsky's idea is as follows. To retrieve the item at virtual address i, the client executes a $\mathsf{Get}(i)$ operation on EDX to check if the item is in the cache. If so, the client executes a $\mathsf{Read}(j)$ operation on $\overline{\mathsf{ERAM}}$, where j is the virtual address of a dummy item, followed by a Put operation on EDX to store the dummy item in the cache. If the ith item was not in EDX, then the client executes a $\mathsf{Read}(i)$ operation on $\overline{\mathsf{ERAM}}$ followed by a Put operation on EDX to store the item just retrieved from $\overline{\mathsf{ERAM}}$. This protocol has several properties: (1) the client always retrieves the desired item; (2) for any two virtual addresses accessed, the view of the server is identically distributed; and (3) $\overline{\mathsf{ERAM}}$ is never queried more than once on the same address. The first property guarantees correctness. The second guarantees that no partial information is revealed about the address queried. The third property guarantees queries cannot be linked; effectively suppressing the leakage of $\overline{\mathsf{ERAM}}$.

Overview of the CBC. As argued above, the square-root solution can be seen as an instantiation of a more general approach that consists of using a ZL encrypted dictionary to suppress the query equality pattern of an encrypted RAM. We observe that this approach is not only applicable to encrypted RAMs (as in the case of the square-root solution) but to a larger class of encrypted structures. We formalize this by abstracting and generalizing this approach. The result is a compiler that, given a structured encryption scheme $\mathsf{STE}_{\mathsf{EDS}}$ with query leakage $\mathsf{qeq} \times \mathsf{patt}$ and a dictionary encryption scheme $\mathsf{STE}_{\mathsf{EDX}}$, with query leakage $\perp$, yields a new structured encryption scheme $\mathsf{STE}_{\mathsf{SDS}}$ with query leakage nrp, where nrp is the non-repeating sub-pattern of patt. If $\mathsf{nrp} = \perp$, then the resulting scheme has ZL queries.

The CBC works as follows. Given a data structure DS of type $\mathbf{T}$ and a query capacity $\lambda \geq 1$, it creates a new structure $\mathsf{SDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$ which consists of: (1) an encryption $\overline{\mathsf{EDS}}$ of a λ-extension of DS; and (2) an encrypted dictionary EDX with capacity λ. To perform a query q on SDS, the client executes a Get on the cache EDX for q. If this results in $\perp$ (i.e., there is no value in the cache with label q) the client queries the main structure $\overline{\mathsf{EDS}}$ with q and updates EDX with the pair (q, r), where r is the result of the query. If, on the other hand, the initial EDX query resulted in a value $v \neq \perp$, the client queries the main structure $\overline{\mathsf{EDS}}$ with an unused dummy. It then updates EDX with the pair (q, v). Rebuilding is handled by creating a new encrypted dictionary EDX and executing the $\mathsf{Rebuild}$ protocol of $\mathsf{STE}_{\mathsf{EDS}}$. Due to space limitations, we defer a more detailed/pseudo-code description to the full version of this work.

Correctness is easy to verify and, intuitively, one can see that EDS will not leak the query equality because it will be queried with any q at most once. There are, however, some subtleties that come up when trying to apply the CBC to structures other than encrypted RAMs. We discuss some of these challenges below.

6.1 Safe Extensions

As highlighted above, the CBC relies on the ability to query the main encrypted structure EDS on dummy values. In other words, EDS must be an encryption of an *extension* $\overline{\mathsf{DS}}$ of the underlying structure DS. In particular, this means that the setup and query leakage of $\mathsf{STE}_{\mathsf{EDS}}$ will be on the extension $\overline{\mathsf{DS}}$ as opposed to the original structure DS. This creates some technical problems that have to be treated carefully.

Extension leakage. The first difficulty is that leakage on $\overline{\mathsf{DS}}$ could reveal useful information about its sub-structure DS. As a concrete example, consider an array encryption scheme with the setup leakage $\mathcal{L}_{\mathsf{St}} = \mathsf{dsize}$ which, in this case, reveals the size of the array. Let $\lambda \geq 1$, $s = \mathsf{dsize}(\mathsf{RAM})$ and consider an extended array $\overline{\mathsf{RAM}}$ with size $2 \cdot (s + \lambda)$ if the first element of the sub-array is even and size $2 \cdot (s + \lambda) + 1$ otherwise. Clearly, the size (i.e., the setup leakage) of the extension $\overline{\mathsf{RAM}}$ reveals a bit of information about the first element of its sub-array.

Definition 8 (Safe extensions). *Let $\Lambda = (\mathsf{patt}_{\mathsf{St}}, \mathsf{patt}_{\mathsf{Qr}}, \mathsf{patt}_{\mathsf{Rb}})$ be a type-$\mathbf{T}$ 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 $(q_1, \ldots, q_t) \in \mathbb{Q}_k^t$, $\mathsf{patt}_{\mathsf{St}}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_{\mathsf{St}}(\mathsf{DS})$, $\mathsf{patt}_{\mathsf{Qr}}(\overline{\mathsf{DS}}, q_1, \ldots, q_t) \leq \mathsf{patt}_{\mathsf{Qr}}(\mathsf{DS}, q_1, \ldots, q_t)$, and $\mathsf{patt}_{\mathsf{Rb}}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_{\mathsf{Rb}}(\mathsf{DS})$.*

6.2 Security of the Cache-Based Compiler

We are now ready to analyze the security of the CBC. In Theorem 1 below, we precisely describe the leakage of the supressed scheme as a function of the leakage of the base scheme, of the extension and of the underlying cache.

Theorem 1. *If $\mathsf{STE}_{\mathsf{EDS}}$ is a static and rebuildable $\big(\mathsf{patt}_{\mathsf{St}}, \mathsf{qeq} \times \mathsf{patt}, \mathsf{patt}_{\mathsf{Rb}}\big)$-secure scheme of type $\mathbf{T}$, if Ext is an $(\mathsf{patt}_{\mathsf{St}}, \mathsf{nrp}, \mathsf{patt}_{\mathsf{Rb}})$-safe extension scheme, and if $\mathsf{STE}_{\mathsf{EDX}}$ is a $(\mathsf{patt}'_{\mathsf{St}}, \bot, \bot)$-secure dictionary encryption scheme, then $\mathsf{STE}_{\mathsf{SDS}}$ is a*

$$\Big(\mathsf{patt}_{\mathsf{St}}, \mathsf{nrp}, \mathsf{patt}_{\mathsf{Rb}}\Big)\text{-secure}$$

scheme of type $\mathbf{T}$, where nrp is the non-repeating sub-pattern of patt.

The proof of Theorem 1 is deferred to the full version of the paper.

7 The Rebuild Compiler

In this section, we describe a compiler that turns any dynamic STE scheme into a rebuildable *static* STE scheme. Recall that for most applications of STE, the client outsources its data to the server. The client, therefore, does not have a local copy of the data from which it can build a new encrypted structure. One possible solution is to have the client retrieve the encrypted structure, "extract" the underlying data structure and set up a new one. This naive approach, however, does not always work as there are many STE schemes that do not support a

form of extraction in the sense above. This is the case, for example, for the SSE constructions of Goh [17] and the ZMF construction of Kamara and Moataz [24]. [10] Another issue occurs if the client does not have enough local storage to store the encrypted structure. In such a case, the rebuild protocol has to be space-efficient for the client and, preferably, make use of only $O(1)$ space.

Overview of the RBC. There are three main challenges in making an encrypted structure rebuildable. The first is that our approach needs to be general-purpose; that is, it should work for any dynamic encrypted structure. Second, the rebuild operation should be time-efficient for the server and client, and space-efficient for the client. The third is that the rebuild operation's leakage should be minimal.

At a high-level, our approach works as follows. When the client constructs an encrypted structure EDS from a plaintext structure DS, it also builds what we refer to as an "encrypted log" RAM. This log is an array that holds encryptions of all the add operations necessary to build the structure DS. The log is stored at the server with EDS. To rebuild EDS, the client will use a sorting network to randomly shuffle the encrypted log at the server. The client and server will then initialize a new (empty) encrypted structure EDS_N. The client then retrieves each ciphertext in the log, decrypts it to recover an update u and executes with the server an add operation on EDS_N for u. After processing every element of the log, EDS_N becomes the new structure. We note that our approach works for both response-hiding and response-revealing constructions.

Detailed description. Let $\mathsf{STE}_{\mathsf{EDS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$ be a dynamic type-**T** STE scheme. Our compiler converts $\mathsf{STE}_{\mathsf{EDS}}$ into a new *static* rebuildable scheme $\mathsf{RSTE}_{\mathsf{EDS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Rebuild})$.

Setup takes as input a static data structure DS and encrypts it using $\mathsf{STE}_{\mathsf{EDS}}.\mathsf{Setup}$. This results in a key K_M and an encrypted structure EDS_M. It then creates an array RAM that stores encryptions of the updates needed to build DS. That is, it computes $(u_1, \ldots, u_m) := \mathsf{Log}(\mathsf{DS})$ and, for all $i \in [m]$, it sets

$$\mathsf{RAM}[i] := \mathsf{SKE.Enc}\big(K_L, u_i\big),$$

where K_L is a symmetric key. Setup outputs $\mathsf{EDS} = (\mathsf{EDS}_M, \mathsf{RAM})$, the keys K_M and K_L for EDS_M and RAM, respectively, and state that includes the state of EDS_M and a counter cnt that will be used to keep track of the number of queries executed.

Query takes as input the secret key, the state and a query from the client, and the encrypted structure from the server. It uses the counter cnt to check if the number of queries since the last $\mathsf{Rebuild}$ has not exceeded λ. If so it executes the query protocol of $\mathsf{STE}_{\mathsf{EDS}}$ and increments cnt. If not, it aborts.

The $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}$ protocol takes as inputs the secret key from the client and the encrypted structure $\mathsf{EDS} = (\mathsf{EDS}_M, \mathsf{RAM})$ from the server. First, the server creates a copy RAM' of RAM. The client and server then obliviously permute

[10] Technically, this is also true for the schemes in [13,10,26,8,7] but they can be easily modified to achieve this property.

RAM'. To do this, the client samples a random permutation π over $[m]$ and the client and server choose a sorting network for $[m]$ items. For each gate $g = (i, j)$ of the network, the server sends the ciphertexts $\mathrm{ct}_i = \mathsf{RAM}'[i]$ and $\mathrm{ct}_j = \mathsf{RAM}'[j]$ to the client. The client decrypts them and swaps them as follows: if $\pi(i) < \pi(j)$, then it returns the pair $(\mathrm{ct}'_i, \mathrm{ct}'_j)$ otherwise it returns the pair $(\mathrm{ct}'_j, \mathrm{ct}'_i)$, where ct'_i and ct'_j are re-encryptions of ct_i and ct_j under the same key K_L. The server then stores the first element of the pair at $\mathsf{RAM}'[i]$ and the second at $\mathsf{RAM}'[j]$. At the end of this phase, RAM' holds a set of randomly permuted and re-encrypted ciphertexts. Next, the client and server initialize a new encrypted structure $((K_N, st_N), \mathsf{EDS}_N) \leftarrow \mathsf{STE}_{\mathsf{EDS}}.\mathsf{Setup}(1^k, \perp)$. The client sequentially retrieves and decrypts all the elements in RAM' and uses the result to update EDS_N. More precisely, for all retrieved ciphertexts ct_i, it computes $u_i := \mathsf{Dec}(K_L, \mathrm{ct}_i)$ and executes $(st_N, \mathsf{EDS}_N) \leftarrow \mathsf{Add}((K_N, st_N, u_i), \mathsf{EDS}_N)$. Finally, it sets the counter cnt back to 0 and sets EDS_M to be the new encrypted structure EDS_N. Due to space constraints, we provide a more detailed description in the full version of this work.

Remark on amortization and latency. The encrypted structures that result from our rebuild compiler are "amortized" in the sense that an entire $\mathsf{Rebuild}$ operation needs to be executed after every λ queries. We note, however, that $\mathsf{Rebuild}$ executions do not affect the latency of Query executions because the two operate on different structures: namely, $\mathsf{Rebuild}$ works on RAM and EDS_N whereas Query works on EDS_M.

Security. We prove the security of our compiler in Theorem 2 below. We give a black-box leakage analysis and later discuss specific instantiations. We show that the resulting scheme is adaptively-secure with a slightly augmented setup leakage, the same query leakage, and rebuild leakage that depends on the underlying scheme's add leakage.

Theorem 2. *If* $\mathsf{STE}_{\mathsf{EDS}}$ *is a dynamic and non-rebuildable* $(\mathsf{patt}_{\mathsf{St}}, \mathsf{patt}_{\mathsf{Qr}}, \mathsf{patt}_{\mathsf{Ad}})$*-secure scheme of type* $\mathbf{T}$*, then* $\mathsf{RSTE}_{\mathsf{EDS}}$ *is a static and rebuildable* $(\mathsf{patt}_{\mathsf{St}} \times \mathsf{lsize} \times \mathsf{mllen}, \mathsf{patt}_{\mathsf{Qr}}, \mathsf{patt}_{\mathsf{Rb}})$*-secure scheme of type* $\mathbf{T}$ *where,*

$$\mathsf{patt}_{\mathsf{Rb}}(\mathsf{DS}) = \Big(\mathsf{patt}_{\mathsf{Ad}}(\mathsf{DS}, u)\Big)_{u \in \mathsf{Log}(\mathsf{DS})}.$$

Due to space limitation, the proof of Theorem 2 is deferred to the full version of the paper.

Efficiency. The resulting scheme produces encrypted structures of size $O(\mathrm{S}^{\mathsf{eds}}(\mathsf{DS}) + |\mathsf{Log}(\mathsf{DS})|_w)$, where $\mathrm{S}^{\mathsf{eds}}(\mathsf{DS})$ is the space complexity of the underlying STE scheme. The query complexity is the same as the underlying scheme's. The complexity of the rebuild operation depends on the sorting network used and the amount of local storage at the client. Using Batcher's bitonic sort [4] with $O(1)$ client local storage, $\mathsf{Rebuild}$ has communication complexity

$$O\left(\sum_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}} + \#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{u \in \mathbb{U}} \mathrm{T}^{\mathsf{eds}}_{\mathsf{Ad}}(u)\right)$$

where $\mathrm{T}^{\mathsf{eds}}_{\mathsf{Ad}}(u)$ is the add complexity of $\mathsf{STE}_{\mathsf{EDS}}$ and $r = \mathsf{qu}(q)$. Note that if $\max_{u\in\mathbb{U}} \mathrm{T}^{\mathsf{eds}}_{\mathsf{Ad}}(u) = O\left(\log^2 \#\mathbb{Q}_{\mathsf{DS}}\right)$, then the rebuild communication complexity is

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

The round complexity of $\mathsf{Rebuild}$ is $O\big(\#\mathbb{Q}_{\mathsf{DS}}\cdot\log^2 \#\mathbb{Q}_{\mathsf{DS}}+\#\mathbb{Q}_{\mathsf{DS}}\cdot\max_{u\in\mathbb{U}} \mathrm{T}^{\mathsf{eds}}_{\mathsf{Ad}}(u)\big)$.

8 Efficiency of the Cache-Based Compiler

In this section, we give the asymptotic overhead of the constructions that result from both the CBC and ORAM simulation (when using tree-based ORAM) and provide a comparison of the two. We defer a more detailed analysis and additional comparisons (e.g., to ORAM simulation with the square-root solution and to custom oblivious data structures) to the full version of this work.

Recall that $\mathsf{STE}_{\mathsf{SDS}}.\mathsf{Query}$ executes: (1) $\mathsf{STE}_{\mathsf{EDS}}.\mathsf{Query}$ in order to query the main structure EDS; (2) $\mathsf{STE}_{\mathsf{EDX}}.\mathsf{Get}$ to query the cache EDX; and (3) $\mathsf{STE}_{\mathsf{SDS}}.\mathsf{Rebuild}$ to rebuild the cache when the counter reaches capacity λ. The *un-amortized* query complexity of the suppressed structure over a query sequence $(q_1, \ldots, q_\lambda)$ is therefore

$$\mathrm{T}^{\mathsf{sds}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda) = \sum_{i=1}^{\lambda} \mathrm{T}^{\mathsf{eds}}_{\mathsf{Qr}}(q_i) + \sum_{i=1}^{\lambda} \mathrm{T}^{\mathsf{edx}}_{\mathsf{Qr}}(q_i) + \mathrm{T}^{\mathsf{eds}}_{\mathsf{Rb}}(\lambda) + \mathrm{T}^{\mathsf{edx}}_{\mathsf{Rb}}(\lambda), \qquad (1)$$

where $\mathrm{T}^{\mathsf{sds}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda)$ is the query complexity of SDS, $\mathrm{T}^{\mathsf{eds}}_{\mathsf{Qr}}(q_i)$ is the query complexity of EDS, $\mathrm{T}^{\mathsf{edx}}_{\mathsf{Qr}}(q_i)$ is the query complexity of the cache EDX, and $\mathrm{T}^{\mathsf{eds}}_{\mathsf{Rb}}(\lambda)$ and $\mathrm{T}^{\mathsf{edx}}_{\mathsf{Rb}}(\lambda)$ are the rebuild complexities of EDS and EDX, respectively.

CBC with a tree-based cache. If the CBC is instantiated with tree-based cache, then we have $\mathrm{T}^{\mathsf{edx}}_{\mathsf{Qr}}(q_1, \ldots, q_i) = O\left(\max_{j\in[i]} |r_j|_w \cdot \log^2 i\right)$, where $r_j = \mathsf{qu}(\mathsf{DS}, q_j)$. Replacing the rebuild cost in Eq. (1) with the cost of the RBC, we have

$$\mathrm{T}^{\mathsf{sds}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda) = \sum_{i=1}^{\lambda} \mathrm{T}^{\mathsf{eds}}_{\mathsf{Qr}}(q_i) + O\left(\lambda \cdot \max_{q\in\mathbf{q}} |r|_w \cdot \log^2 \lambda\right) + O\left(\sum_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}}\right)$$

where $\mathbf{q} = (q_1, \ldots, q_\lambda)$.

ORAM simulation with the tree-based ORAM. ORAM simulation of a structure DS using tree-based ORAM has the following complexity.

$$\mathrm{T}^{\mathsf{tree}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda) = \sum_{i=1}^{\lambda} \mathrm{B}(q_i) \cdot O\left(\log^2 \frac{|\mathsf{DS}|_2}{B}\right) \cdot \frac{B}{w},$$

where B is the block-size of the ORAM and $\mathrm{B}(q_i)$ denotes the number of blocks that need to be read to answer query q_i. Setting $B = \max_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_2$, we have

$$\mathrm{T}^{\mathsf{tree}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda) = \sum_{i=1}^{\lambda} \mathrm{B}(q_i) \cdot O\left(\log^2 \frac{|\mathsf{DS}|_2}{B}\right) \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w.$$

CBC vs. ORAM simulation. In the following proposition, we compare the efficiency of the CBC with the efficiency of ORAM simulation.

Proposition 1. *Let* DS *be a type-*$\mathbf{T}$ *data structure and* $\mathbf{q} = (q_1, \ldots, q_\lambda)$ *be a query sequence with* $1 \leq \lambda \leq \#\mathbb{Q}_{\mathsf{DS}}$*. If*

$$\sum_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w = o\left(\sum_{i=1}^{\lambda} \mathrm{B}(q_i) \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w\right) \quad \text{and} \quad \lambda \cdot \max_{q \in \mathbf{q}} |\mathsf{qu}(\mathsf{DS}, q)|_w = O\left(\sum_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w\right)$$

then

$$\mathrm{T}^{\mathsf{sds}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda) = o\left(\mathrm{T}^{\mathsf{tree}}_{\mathsf{Qr}}(q_1, \ldots, q_\lambda)\right).$$

Note that for structures with constant-time queries, $\mathrm{B}(q_i) = 1$, our approach improves asymptotically over ORAM simulation whenever

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

However, for structures with non-constant query complexity (e.g. search trees, graphs), our approach improves over ORAM simulation whenever

$$\sum_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w = o\left(\omega(1) \cdot \lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w\right).$$

A note on our assumptions. We note that the assumptions in Proposition 1 are natural and are satisfied in many scenarios of interest. For example, if the response lengths of DS are distributed according to a power law (a common assumption in the context of keyword search), there always exists a λ for which the first assumption holds. Furthermore, the second assumption follows whenever queries with small responses are more likely than queries with large responses. Again, this is a common assumption in keyword search where users are more likely to search for keywords contained in smaller numbers of documents than keywords that are stored in large number of documents.

9 PBS: The Piggyback Scheme

We describe a general-purpose STE scheme that reveals the query equality and response length on arbitrary query sequences, but only the total response length on sequences of distinct queries. As we will see in Section 10, this construction, combined with the RBC and the CBC, results in a scheme that only leaks the total response length on arbitrary sequences. The main idea behind the scheme is to trade-off query latency for leakage. [11] At a high level, our approach is to hide response lengths by ensuring the client retrieves a fixed number of words per query (a batch); no matter what the response length. To maintain correctness,

[11] This approach was first suggested in [23] but never described or analyzed formally.

incoming queries are queued and processed at the next available time. Naturally, this introduces a delay/latency in the querying process but by carefully tuning the batch size we can ensure that the entire response is retrieved in a reasonable amount of time. For ease of exposition, we describe a slightly simpler variant of our construction which achieves correctness under some assumptions (which we describe below).

Overview. Our scheme makes black-box use of a dynamic dictionary encryption scheme $\mathsf{STE}_{\mathsf{EDX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Put})$. Given a batch size $\alpha \geq 1$ and a data structure DS, if $\mathsf{DS} \equiv d_0$, the Setup initializes an empty encrypted dictionary EDX. Otherwise, for every query $q \in \mathbb{Q}_{\mathsf{DS}}$, it does the following. It divides q's response r into N words $(r_1, \ldots, r_N)$ and pads it with enough $\perp$ symbols to make it a multiple of the batch size α. It then adds the pairs $\big((q\|1, r_1), \ldots, (q\|N+p, r_{N+p})\big)$ to DX, where p is the number of $\perp$ symbols. It also keeps track of q's batch size $(N+p)/\alpha$ in a dictionary DX_{st}. After processing every query in $\mathbb{Q}_{\mathsf{DS}}$, it encrypts DX with $\mathsf{STE}_{\mathsf{EDX}}$. The output includes a key K_D, the encrypted dictionary EDX and a state st_D. The state of scheme st includes the batch size α, a timeout parameter θ assumed to be larger than the maximum time gap between updates, the encrypted dictionary state st_D, the dictionary DX_{st} and two empty queues Q_s and Q_u. Note that the reason we pad is to guarantee the ability to retrieve α words even when the queue contains a single query. For example, if we did not pad and the first query's response consisted of less than α words, the server would clearly learn the response length of that query.

Query 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 a query q and from the server $\mathsf{EDS} = \mathsf{EDX}$. The client starts by adding the pair $(q, 0)$ to Q_s. It then peeks at Q_s to recover the pair (q', c) and retrieves α words by executing $\mathsf{STE}_{\mathsf{EDX}}.\mathsf{Get}$ on labels $q'\|c \cdot \alpha + 1, \ldots, q'\|c \cdot \alpha + \alpha$. It uses DX_{st} to check if this was the last batch of words for q' and if so it removes (q', c) from Q_s. If not, it updates c to $c + 1$.

Add 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 from the server $\mathsf{EDS} = \mathsf{EDX}$. It checks if the last update occurred more than θ time ago. If so, it flushes Q_u by executing $\mathsf{STE}_{\mathsf{EDS}}.\mathsf{Put}$ on all the remaining updates in Q_u and aborts. If not, it parses the update u as a pair composed of the query q and its response r. Similar to Setup, it divides q's response r into N words $(r_1, \ldots, r_N)$ and pads it with enough $\perp$ symbols to make it a multiple of the batch size α. The padded response now has length $c = (N+p)/\alpha$, where p is the number of $\perp$ symbols added. It also keeps track of q's batch size $(N+p)/\alpha$ in a dictionary DX_{st}. The client then adds the pair $((q, r), c-1)$ to the queue Q_u. It then peeks at Q_u to recover the pair $((q', r'), c')$ and updates EDX by executing $\mathsf{STE}_{\mathsf{EDX}}.\mathsf{Put}$ on the update sequence $(q'\|c' \cdot \alpha + 1, r'_1), \ldots, (q'\|c' \cdot \alpha + \alpha, r'_\alpha)$. It removes all r'_i from r', for $i \in [\alpha]$. Finally, if the counter c' is equal to 0, then it removes the pair (u', c) from Q_u, otherwise, it updates c' by $c' - 1$.

Note that, as described, the construction will be correct as long as: (1) the updates $u = (q, r)$ are only for new queries; and that (2) we never query on

queries that are still being updated (i.e., that are still in Q_u). In the full version of this work, we show how to lift these restrictions and provide a detailed description of the scheme.

9.1 Security of PBS

In this Section, we analyze the security of PBS. Even though the scheme makes black-box use of an encrypted dictionary we find that here a black-box leakage analysis is not as informative as a concrete leakage analysis. Therefore, in Theorem 3 below we consider the security of PBS instantiated with any response-hiding dynamic encrypted dictionary that has the following leakage profile

$$\Lambda_{\mathsf{EDX}} = \big(\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Gt}}, \mathcal{L}_{\mathsf{Pt}}\big) = \Big(\mathsf{trlen}, \mathsf{qeq}, \bot\Big).$$

This profile can be achieved by extending well-known constructions [13,7]. We give such an example in the full version of the paper.

Setup leakage. The setup leakage of PBS is the *total batched response length* which reveals the total number of padded word batches needed to store the responses in the structure. More precisely, this is defined as $\mathsf{tbrlen} = \{\mathsf{tbrlen}_{k,\alpha}\}_{k,\alpha\in\mathbb{N}}$, where $\mathsf{tbrlen}_{k,\alpha} : \mathbb{D}_k \to \mathbb{N}$ with:

$$\begin{aligned}\mathsf{tbrlen}_{k,\alpha}(\mathsf{DS}) &= \sum_{r\in\mathbb{R}_{\mathsf{DS}}} |r| + \alpha - \big(|r| \mod \alpha\big)\\ &= \mathsf{trlen}(\mathsf{DS}) + \sum_{r\in\mathbb{R}_{\mathsf{DS}}} \alpha - \big(|r| \mod \alpha\big).\end{aligned}$$

Query leakage. The query leakage of PBS is the *repeated query equality pattern* $\mathsf{rqeq} = \{\mathsf{rqeq}_{k,m}\}_{k,m\in\mathbb{N}}$, where $\mathsf{rqeq}_{k,m} : \mathbb{D}_k \times \mathbb{Q}_k^t$ is defined as:

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

where

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

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

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

The non-repeating sub-pattern reveals nothing except on the last query where it reveals γ_m, i.e., the total number of batches needed to finish retrieving the entire sequence. For repeated sequences, rqeq reveals the query equality and the response length patterns.

Note that, intuitively speaking, it seems that PBS leaks "less" than rqeq. Specifically, it doesn't leak qeq $\times$ rlen on every repeating sequence. Nevertheless, the scheme's leakage on non-repeating patterns is captured precisely by nrp which is ultimately what is relevant for use with the CBC.

Add leakage. The add leakage of PBS is the *add length pattern* $\mathsf{alen} = \{\mathsf{alen}_{k,m}\}_{k,m\in\mathbb{N}}$, where $\mathsf{alen}_{k,m} : \mathbb{D}_k \times \mathbb{U}_k^t$ is defined as:

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

The add length pattern reveals nothing except on the last update where it reveals γ_m, i.e., the total number of batches needed to finish the update sequence.

Theorem 3. *If* $\mathsf{STE}_{\mathsf{EDX}}$ *is* $(\mathsf{trlen}, \mathsf{qeq}, \perp)$*-secure, then* PBS *is* $(\mathsf{tbrlen}, \mathsf{rqeq}, \mathsf{alen})$*-secure.*

The proof of Theorem 3 is deferred to the full version of the paper.

9.2 Latency of PBS

We now analyze the latency of our construction; that is, how long the client has to wait until it receives the entire response for its query. For a query sequence $(q_1, \ldots, q_t)$, the client's waiting time at time t is equal to the number of queries left in the queue at that time. In the worst-case, this is

$$t \cdot \left(\frac{\max_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w}{\alpha} - 1 \right).$$

Note that if α is set to $\max_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w$, the scheme does not introduce any latency. This, of course, comes at the cost of a large amount of padding which translates to storage and communication overhead.

The above bound on the latency helps us understand the limitations of the scheme in the worst case but it does not tell us much about its latency in general. Given that, in practice, a client is very unlikely to exclusively search for queries with maximum response length, we are interested in more likely scenarios where client queries and their response lengths follow some known distributions.

The Zipf distribution. To get a more interesting bound on latency, we have to make assumptions on how queries are sampled and how the response lengths are distributed. Here, we will assume queries are sampled from a Zipf distribution $\mathcal{Z}_{n,s}$ with probability mass function $f_{n,s} : [n] \to [0, 1]$, $f_{n,s}(r) = r^{-s}/H_{n,s}$, where r is the rank of the query and $H_{n,s}$ is the harmonic number $H_{n,s} = \sum_{i=1}^{n} i^{-s}$. Recall that the Zipf distribution is defined over ranks so we assume an implicit ranking function that maps queries to their rank.

We also assume that the lengths of the responses are Zipf distributed by which we mean that the r^{th} response has word length

$$\frac{r^{-s}}{H_{n,s}} \cdot \sum_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w.$$

Here again, we assume the existence of a ranking function that maps responses to their rank. From our second assumption, it follows that the set of all response lengths is

$$L = \left\{ \frac{T}{1 \cdot H_{n,s}}, \ldots, \frac{T}{n^s \cdot H_{n,s}} \right\},$$

where $T \stackrel{def}{=} \sum_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w$. In our analysis, we will set $s = 1$. All these assumptions are inspired from the information retrieval literature [12,43] where it is common to assume that keyword search queries are sampled from a distribution $\mathcal{Z}_{n,s}$ and that the number of documents in which keywords appear is distributed according to $\mathcal{Z}_{n,s}$. Furthermore, for English language queries and datasets, it common to set $s = 1$.

Before we can finish our analysis, we need to make a third assumption. Specifically, we have to choose a mapping between the r^{th} ranked query and a response. Here, we will assume that high-rank queries have low-rank responses. The intuition is that, in the context of keyword search, we tend to search more often for keywords that appear less frequently in the dataset. Alternatively, we tend to search less for keywords that appear frequently in the data. In our analysis, we will refer to this as the *inverted query hypothesis*.

In the following Theorem we bound the probability that the client will retrieve all of its responses as a function of the number of *additional* query operations it executes, i.e., the number of operations beyond the minimal t.

Theorem 4. *If the rank of the client's queries is sampled i.i.d. from $\mathcal{Z}_{n,1}$ and if the lengths of the responses are distributed according to the $\mathcal{Z}_{n,1}$ distribution then, under the inverted query hypothesis, the client will retrieve all of its responses after an additional $\varepsilon \cdot t$ query operations with probability at least*

$$1 - \exp\left(- 2t \left(\varepsilon \cdot \frac{\alpha}{\mu} \right)^2 \right),$$

where $\mu \stackrel{def}{=} \max_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w$.

The proof of Theorem 4 is deferred to the full version of the paper.

Correctness vs. leakage. As described above, PBS achieves perfect correctness and the client will retrieve the responses for all its queries. For this, however, the client needs to perform additional queries (i.e., more than the t queries in its sequence) in order to empty its queue Q_s.

The scheme, however, can be used differently. Specifically, if the client is willing to trade correctness for lower leakage, it can stop querying after m query operations. Theorem 4 shows that after a sequence of t queries, with probability that is a function of ε, the client needs to perform an additional $\varepsilon \cdot t$ query operations to empty its queue (of course assuming the queries are sampled according to a Zipf distribution). Assuming the client sets the size of the queue to meet its requirements, if it stops querying after m query operations, the leakage profile of PBS becomes

$$\Lambda_{\mathsf{PBS}} = \left(\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Qr}}, \mathcal{L}_{\mathsf{Ad}} \right) = \left(\mathsf{tbrlen}, \mathsf{rqeq}', \perp \right),$$

where

$$\mathsf{rqeq}'_{k,m}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \bot & \text{if } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{qeq} \times \mathsf{rlen}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise.} \end{cases}$$

In this case, the scheme's non-repeating sub-pattern leakage is just $\bot$.

10 (Almost) Zero-Leakage Structured Encryption

We now describe an almost zero-leakage STE scheme, AZL, followed by a fully ZL variant we refer to as FZL. AZL results from first applying the RBC to PBS, and then applying CBC to the result. In the following, we describe the leakage profiles of the intermediate constructions that result from this process.

Applying the RBC to PBS. Let RPBS be the scheme that results from applying the RBC to PBS. It follows by Theorem 2 that the concrete leakage profile of this scheme is,

$$\Lambda_{\mathsf{RPBS}} = (\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Qr}}, \mathcal{L}_{\mathsf{Rb}}) = \left(\left(\mathsf{tbrlen}, \mathsf{lsize}, \mathsf{mllen}\right), \mathsf{rqeq}, \left(\mathsf{ulen}, \mathsf{lsize}, \mathsf{mllen}\right)\right),$$

where $\mathsf{lsize} = \{\mathsf{lsize}_k\}_{k\in\mathbb{N}}$ is defined as $\mathsf{lsize}_k(\mathsf{DS}) = \#\mathsf{Log}(\mathsf{DS})$, $\mathsf{mllen} = \{\mathsf{mllen}_k\}_{k\in\mathbb{N}}$ is defined as $\mathsf{mllen}_k(\mathsf{DS}) = \max_{u\in\mathsf{Log}(\mathsf{DS})} |u|$, and $\mathsf{ulen} = \{\mathsf{ulen}_{k,m}\}_{k,m\in\mathbb{N}}$ is defined as

$$\mathsf{ulen}_{k,m}(\mathsf{DS}) = \Big(\mathsf{alen}_{k,m}(u)\Big)_{u\in\mathsf{Log}(\mathsf{DS})}.$$

Safely extending RPBS. We now show how to safely extend RPBS so that it can be used as the base scheme of the CBC. Here, we assume that λ and α are publicly-known parameters and that all queries in the query space $\mathbb{Q}_{\mathsf{DS}}$ have the same bit length. Let $(\widetilde{q}_1, \cdots, \widetilde{q}_\lambda)$ be dummy queries. For all $i \in [\lambda]$, compute $\overline{\mathsf{DS}} \leftarrow \mathsf{Update}(\overline{\mathsf{DS}}, (\widetilde{q}_i, \mathbf{0}))$, where $|\mathbf{0}|_w = \max_{r\in\mathbb{R}_{\mathsf{DS}}} |r|_w$.

Theorem 5. *The extension scheme described above is*

$$\left(\left(\mathsf{tbrlen}, \mathsf{lsize}, \mathsf{mllen}\right), \mathsf{nrp}, \left(\mathsf{ulen}, \mathsf{lsize}, \mathsf{mllen}\right)\right)\text{-safe.}$$

The proof of Theorem 5 is deferred to the full version of the paper.

Applying the CBC. Let AZL be the scheme that results from applying the CBC to RPBS using the extension scheme described above. It follows by Theorem 1 that the concrete leakage profile of AZL is

$$\Lambda_{\mathsf{AZL}} = (\mathcal{L}_{\mathsf{St}}, \mathcal{L}_{\mathsf{Qr}}, \mathcal{L}_{\mathsf{Rb}}) = \left(\left(\mathsf{tbrlen}, \mathsf{lsize}, \mathsf{mllen}\right), \mathsf{nrp}, \left(\mathsf{ulen}, \mathsf{lsize}, \mathsf{mllen}\right)\right),$$

where

$$\mathsf{nrp}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \bot & \text{if } t < m \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \gamma_\lambda & \text{if } t = \lambda \text{ and } q_i \neq q_j \text{ for all } i, j \in [t]. \end{cases}$$

Note that the setup leakage of the cache is mllen which is already included in the setup leakage of RPBS.

Efficiency. AZL has query complexity

$$\mathrm{T}^{\mathsf{SDS}}_{\mathsf{Qr}}(q_1,\ldots,q_\lambda)=\sum_{i=1}^{\lambda}\mathrm{T}^{\mathsf{EDS}}_{\mathsf{Qr}}(q_i)+O\left(\lambda\cdot\max_{q\in\mathbf{q}}|r|_w\cdot\log^2\lambda\right)+O\left(\sum_{r\in\mathbb{R}_{\mathsf{DS}}}|r|_w\cdot\log^2\#\mathbb{Q}_{\mathsf{DS}}\right),$$

and storage complexity

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

If $\max_{u\in\mathsf{Log}(\mathsf{DS})}|u|_w=O(\max_{r\in\mathbb{R}_{\mathsf{DS}}}|r|_w)$, the storage overhead simplifies to

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

Achieving zero-leakage. As discussed in Section 9, the non-repeating sub-pattern leakage of PBS is $\perp$ if we are willing to tolerate probabilistic correctness. In such a case, applying the RBC and then the CBC results in a scheme FZL with query leakage,

$$\Lambda_{\mathsf{FZL}}=\left(\mathcal{L}_{\mathsf{St}},\mathcal{L}_{\mathsf{Qr}},\mathcal{L}_{\mathsf{Rb}}\right)=\left(\left(\mathsf{tbrlen},\mathsf{lsize},\mathsf{mllen}\right),\perp,\left(\mathsf{ulen},\mathsf{lsize},\mathsf{mllen}\right)\right).$$

The efficiency of FZL is the same as AZL.

An improved extension for probabilistic correctness. We briefly note that under probabilistic correctness, we can extend RPBS more efficiently than described above. The extension works as follows. Let $(\widetilde{q}_1,\ldots,\widetilde{q}_\lambda)$ be dummy queries. For all $i\in[\lambda]$, compute $\overline{\mathsf{DS}}\leftarrow\mathsf{Update}(\overline{\mathsf{DS}},(\widetilde{q}_i,\mathbf{0}))$, where $|\mathbf{0}|_w=\alpha$. Note that the setup and rebuild leakage of this variant are the same as those considered in Theorem 5 so they can be simulated exactly as in the proof of that Theorem. The non-repeating query sub-pattern is $\mathsf{nrp}(\overline{\mathsf{DS}},q_1,\ldots,q_t)=\mathsf{nrp}(\mathsf{DS},q_1,\ldots,q_t)=\perp$ which can be simulated trivially.

11 Conclusions and Future Directions

In this work, we introduced a new framework to cope with leakage based on compilers and transformations that suppress the leakage of STE schemes. Our work motivates several interesting directions for future work. The most immediate is the design of a query equality suppression framework for dynamic STE schemes. Another interesting challenge is to design compilers with lower computational overhead. Here, trying to improve the cost of our rebuild compiler—even for restricted classes of encrypted structures—might be a good start. As far as we know, our PBS construction is the first STE scheme to hide the response length without naive padding (i.e., padding to the maximum response length). To achieve this, we used queuing techniques which introduce a delay in the query process. Can the latency of PBS be improved? Can response lengths be suppressed without introducing delays at all? Finally, in this work we focused on suppressing query equality and response length leakage but an important direction for future work is to find suppression techniques and frameworks for other common leakage patterns.

Acknowledgments. We are grateful to Hajar Alturki for useful feedback on the PBS construction and to the anonymous reviewers for helpful suggestions.

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. *IACR Cryptology ePrint Archive*, 2018:195, 2018.
3. G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *(STOC '16)*, pages 1101–1114, New York, NY, USA, 2016. ACM.
4. K. Batcher. Sorting networks and their applications. In *Proceedings of the Joint Computer Conference*, pages 307–314, 1968.
5. R. Bost. Sophos - forward secure searchable encryption. In *ACM (CCS '16)*, 20016.
6. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM (CCS '15)*, pages 668–679. ACM, 2015.
7. 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 *(NDSS '14)*, 2014.
8. 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.
9. D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.
10. M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT '10*, pages 577–594. Springer, 2010.
11. M. Chase and S. Kamara. Structured encryption and controlled disclosure. Technical Report 2011/010.pdf, IACR Cryptology ePrint Archive, 2010.
12. S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *ACM SIGIR 2007*.
13. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *(CCS '06)*, 2006.
14. I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD'17*, 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. S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO'16*, 2016.
17. E.-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.
18. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
19. M. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *(CCSW '11)*, 2011.
20. M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *(NDSS '12)*, 2012.
21. S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM (CCS '13)*, 2013.

22. S. Kamara. Restructuring the NSA metadata program. In *Workshop on Applied Homomorphic Cryptography*, Lecture Notes in Computer Science. Springer, 2014.
23. S. Kamara and T. Moataz. SQL on structurally-encrypted databases. *IACR Cryptology ePrint Archive*, 2016:453, 2016.
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 (CCS '12)*, 2012.
27. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *(SODA '12)*, 2012.
28. C. Liu, L. Zhu, M. Wang, and Y. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.
29. X. Meng, S. Kamara, K. Nissim, and G. Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *(CCS 15)*, 2015.
30. 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.
31. M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.
32. R. Ostrovsky and V. Shoup. Private information storage. In *ACM Symposium on Theory of Computing (STOC '97)*, pages 294–303, 1997.
33. 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.
34. S. Sedghi, P. van Liesdonk, J. M. Doumen, P. H. Hartel, and W. Jonker. Adaptively secure computationally efficient searchable symmetric encryption. Technical Report TR-CTIT-09-13, 2009.
35. E. Shi, T. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with o((logn)3) worst-case cost. In *ASIACRYPT*, 2011.
36. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE S&P*, pages 44–55. IEEE Computer Society, 2000.
37. E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *(NDSS'14)*, 2014.
38. 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 *(CCS)*, 2013.
39. P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *(CCS '08)*, 2008.
40. 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*, 2016.
41. Y. Zhang, A. O'Neill, M. Sherr, and W. Zhou. Privacy-preserving network provenance. *Proc. VLDB Endow.*, 10(11):1550–1561, Aug. 2017.
42. G. K. Zipf. The psycho-biology of language. 1935.