SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage

Ioannis Demertzis
yannis@umd.edu
University of Maryland

Dimitrios Papadopoulos
dipapado@cse.ust.hk
Hong Kong University of Science and Technology

Charalampos Papamanthou
cpap@umd.edu
University of Maryland

Saurabh Shintre
Saurabh_Shintre@symantec.com
Symantec Research Labs

ABSTRACT

Building expressive encrypted databases that can scale to large volumes of data while enjoying formal security guarantees has been one of the holy grails of security and cryptography research. Searchable Encryption (SE) is considered to be an attractive implementation choice for this goal: It naturally supports basic database queries such as *point*, *join* and *range*, and is very practical at the expense of well-defined leakage such as *search* and *access* pattern. Nevertheless, recent attacks have exploited these leakages to recover the plaintext database or the posed queries, casting doubt to the usefulness of SE in encrypted systems. Defenses against such leakage-abuse attacks typically require the use of Oblivious RAM or worst-case padding—such countermeasures are however quite impractical. In order to efficiently defend against leakage-abuse attacks on SE-based systems, we propose SEAL, a family of new SE schemes with *adjustable leakage*. In SEAL, the amount of privacy loss is expressed in leaked bits of search or access pattern and can be defined at setup. As our experiments show, when protecting only a few bits of leakage (e.g., three to four bits of access pattern), enough for existing and even new more aggressive attacks to fail, SEAL query execution time is within the realm of practical for real-world applications (a little over one order of magnitude slowdown compared to traditional SE-based encrypted databases). Our findings therefore show that SEAL could be a promising approach for building efficient and robust encrypted databases.

1 INTRODUCTION

Encrypted databases enable a data owner to outsource a database to a server in a private manner, so that the server can still answer database queries (e.g., point, join, range) on the underlying encrypted data. Initially implemented with weak primitives like order-preserving (OPE) and deterministic (DET) encryption (e.g., [5, 6, 36, 40])[1], encrypted databases have now moved to more "secure" implementations through other primitives like searchable or structured encryption (SE) [12], offering support for a plethora of queries such as point queries [16, 17], range queries [14, 15, 18], and SQL queries [26] (e.g., join and group-by queries).

SE-based encrypted databases are quite practical at the expense of well-defined *leakage*. This leakage information includes the *search pattern* (whether a query q has been made in the past or not) and the *access pattern* that consists of the *volume pattern* (number of database tuples contained in the query result) and the *overlapping pattern* (which database tuples, if any, in the result for query q appeared in the result of a previous query).

Leakage-abuse attacks. Unfortunately the aforementioned leakages exposed by SE can be quite harmful, enabling the recovery of the encrypted database or/and the posed queries. In particular, the works of Islam et al. [24] and Cash et al. [10] were the first to exploit access pattern leakage and prior knowledge about the dataset to recover the queried keywords. Zhang et al. [43] propose file injection attacks for encrypted email applications to improve the recovery rate of queried keywords. For the private range search problem, effective access pattern and volume pattern attacks through which the attacker learns the plaintext order and value of all encrypted records, without any prior knowledge, have been proposed [13, 22, 28–30, 32]. This ever-growing body of leakage-abuse attacks has already alerted the community about using SE for implementing encrypted database systems [1].

Current defenses. To provably defend against leakage-abuse attacks on SE-based systems one has to (i) use expensive cryptographic tools to eliminate the search/overlapping patterns, i.e., Oblivious RAM (ORAM) [38] (introducing a polylogarithmic search overhead) and (ii) perform worst-case padding (resulting in worst-case linear search time [25] or quadratic index size) for eliminating the volume pattern. Both approaches above incur large overheads leading to quite impractical protocols. We present other, more practical, but less effective defenses in our prior work section.

Our contributions. In light of the above, we ask in this paper whether practical SE primitives can still somehow be used to implement secure encrypted databases. Towards this goal, we propose SEAL[2], a family of new SE schemes with *adjustable leakage* which allow the client to define a trade-off between efficiency and leaked information. We show that hiding *only a few bits* of the search/overlapping/volume pattern significantly reduces the success of existing as well new, even more aggressive, leakage-abuse attacks. At the same time SEAL's practical performance is close to traditional SE. In particular our contributions are as follows:

[1] Note that such implementations have been shown to be susceptible to inference attacks [34] since they leak statistical and order information allowing an attacker to decrypt the actual encrypted records.

[2] **SEAL** stands for **S**earchable **E**ncryption with **A**djustable **L**eakage.

.

(1) To better motivate SEAL, we first present new attacks on existing SE-based encrypted databases. In particular, we show that the same inference attacks [34] on DET systems can be used by a persistent adversary to recover the database values in SE-based systems, such as those implementing point queries (e.g., [14]), and group-by and join queries (e.g., [26]). The high-level reason is that after the adversary observes a certain number of SE queries in these constructions, tuples with the same values are revealed and therefore frequency information is readily available to the adversary. Even for more robust SE-based range query schemes [14, 18], we present new attacks that can work under certain assumptions about the dataset (see Section 3).
(2) We present SEAL(α, x), a family of SE schemes with adjustable leakage. SEAL is based on two other "adjustable" primitives, an adjustable ORAM, parameterized by a value α and an adjustable padding algorithm, parameterized by a value x. The adjustable ORAM, ADJ-ORAM-α, hides only α bits of the access pattern by partitioning the accessed N-sized array into $N/2^\alpha$ regions of 2^α size each and by applying an individual standard ORAM per region. The adjustable padding algorithm, ADJ-PADDING-x, reduces the volume pattern leakage by padding every list to the the closest power of x, leading to a dataset with at most $\log_x N$ distinct sizes. Clearly, larger values for α and x yield slower but more secure SEAL (see Section 4).
(3) We use SEAL to build encrypted databases with adjustable leakage. We first present three new constructions POINT-ADJ-SE-(α, x) (for point queries), JOIN-ADJ-SE-(α, x) (for join queries) and RANGE-ADJ-SE-(α, x) (for range queries) that use SEAL(α, x) as black box, instead of plain SE. Finally, we present a more efficient adjustable construction for ranges, RANGE-SRC-SE, that reduces access pattern leakage and volume pattern leakage *implicitly* by modifying an existing construction [15] and not by using our (more expensive) SEAL(α, x). (see Sections 4.4 and 4.5).
(4) We evaluate the robustness of our SEAL-based encrypted databases for various values α and x against particularly powerful adversaries that observe the leaked search/overlapping and volume patterns and *have plaintext access to the entire input dataset*. Such strong threat model offers additional credibility to our proposed mitigation techniques. We consider two new attacks. The first is a *query recovery attack* that aims at decrypting the encrypted queries posed by the client. The second is a *database recovery attack* that aims at mapping encrypted database tuples to (decrypted) client queries[3]. We observe that these attacks are related. For example, in the case of point queries, if an attacker recovers the plaintext value v of a query q and additionally figures out *which* tuples this query returns, then the adversary can infer that the value of the queried attribute of the returned tuples is v (see Section 5).
(5) We observe that for all the above attacks we can find certain values for α and x that reduce the attacker's success rate significantly while maintaining good performance. For instance we show that if we use SEAL to hide three bits of access pattern while at the same time pad the keyword lists to powers of 4 (thus hiding a few bits of volume pattern as well), we can defend against our all-powerful attackers only at the expense of an acceptable slowdown from plain SE—around 32×.

[3]We note here that this attack is trivial in the case of plain SE since this mapping is part of the leakage.

Prior work. Wagh et al. [41] introduces an ORAM with a tunable trade-off between the search/storage efficiency and security. This trade-off is controlled by an (ϵ, δ)-differential privacy modification of PathORAM [38]. Their construction could potentially be used as a drop-in replacement in our proposed encrypted database algorithms (instead of our adjustable ORAM). It would be interesting to explore how different choices of ϵ and δ affect the performance of existing leakage-abuse attacks—we leave this as future work.

The works of Cash et al. [10], and Bost and Fouque [8] propose padding techniques for keyword search that can hide a portion of the volume pattern. Unlike our proposed padding in Section 4.2, their padding depends on the distribution of the input dataset, which results in leakage even prior to query execution. Bost and Fouque [8] also propose new security definitions for SE aiming at capturing existing leakage abuse attacks. These definitions could potentially provide intuition on how we can modify existing schemes in order to make them robust against such attacks.

Recently, Kamara et al. [27] showed how to suppress the search pattern leakage without using ORAM. However suppressing only the search pattern leakage is not enough for mitigating leakage-abuse attacks. Kamara and Moataz [25] showed theoretically how to perform worst-case padding without requiring quadratic index size, while sometimes assuming certain properties for the input dataset, such as a Zipf distribution or highly-concentrated multimaps. Finally, Markatou and Tamassia [31] have recently presented new mitigation techniques for range queries based on various range query transformations.

2 PREMILIMINARIES

We now provide some notation, definitions and background that we use throughout the paper. We write $out \leftarrow \mathsf{Alg}(in)$ to indicate the output of an algorithm Alg and $(client_{out}, server_{out}) \leftrightarrow \mathsf{Prot}(client_{in}, server_{in})$ to indicate the execution of a protocol Prot between a client and a server.

Negligible Function. A function $\nu\colon \mathbb{N} \rightarrow \mathbb{R}$ is negligible in λ, denoted by $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

Oblivious RAM (ORAM). Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [20], is a compiler that encodes the memory such that accesses on the compiled memory do not reveal access patterns on the original memory. An ORAM scheme consists of two algorithms/protocols ORAM = (OramInitialize, OramAccess), where OramInitialize initializes the memory, and OramAccess performs the oblivious accesses. We provide the formal definition in Section 4.3.

Oblivious Dictionary (ODICT). An oblivious dictionary is an oblivious data structure that can support oblivious queries from an arbitrary domain. ODict offers the following protocols (see [42] for a detailed description):

- $(T, \sigma) \leftarrow$ ODictSetup$(1^\lambda, N)$: Given a security parameter λ, and an upper bound N on the number of elements, it creates an oblivious data structure T. The client sends T to the server and maintains locally the state σ.
- $((\texttt{value}, \sigma'), T') \leftrightarrow$ ODictSearch$((\texttt{key}, \sigma), T)$: Given the search key key and σ, returns the corresponding value value, the updated T' and σ'.
- $(\sigma', T') \leftrightarrow$ ODictInsert$((\texttt{key}, \texttt{value}, \sigma), T)$: Given a key-value pair key, value and σ, it inserts this entry in the dictionary. It returns the updated T' and σ'.

Searchable Encryption (SE). Let $\mathcal{D}$ be a collection of *documents*. Each document $D \in \mathcal{D}$ is assigned a unique document identifier and contains a set of keywords from a dictionary Δ. Let $\mathcal{D}(w)$ denote the identifiers of documents containing keyword w. SE schemes build an *encrypted index* $\mathcal{I}$ on the document identifiers which can be queried using keyword "tokens". Note that we do not store encrypted documents in the index, just their identifiers. Encrypted documents can be retrieved in an extra round. We denote with N the data collection size, i.e., $N = \sum_{w \in \Delta} |\mathcal{D}(w)|$. An SE *protocol* involves two parties, a *client* and a *server* and consists of the following algorithms/protocols [12]:

- $(st_C, \mathcal{I}) \leftarrow$ Setup$(1^\lambda, \mathcal{D})$: is a probabilistic algorithm performed by the client prior to sending any data to the server. It receives the security parameter as input and the data collection $\mathcal{D}$, and outputs an encrypted index $\mathcal{I}$ which is sent to the server. st_C is sent to the client and it contains the secret key k.
- $((\mathcal{X}, st_C), \mathcal{I}) \leftrightarrow$ Search$((st_C, w), \mathcal{I})$: is a protocol executed between the client and the server. The client inserts the secret state st_C and a keyword w, while the server inserts an encrypted index $\mathcal{I}$. At the end of the protocol the client learns $\mathcal{X}$, the set of all document identifiers $\mathcal{D}(w)$ corresponding to keyword w and the updated secret state st_C, while the server's output is the updated encrypted index $\mathcal{I}$.

The security of the above SE scheme is captured by the following definition, using the standard SE's real/ideal security game [11] (also provided in Appendix—see Figure 14).

Definition 2.1. Suppose (KeyGen, Setup, Search) is a SE scheme based on the above definition, let $\lambda \in \mathbb{N}$ be the security parameter and consider experiments $\mathbf{Real}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{L}_1, \mathcal{L}_2}(\lambda)$ presented in Figure 14, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are leakage functions. SE is $(\mathcal{L}_1, \mathcal{L}_2)$-secure if for all polynomial-size adversaries $\mathcal{A}$ there exist polynomial-time simulators SimSetup and SimSearch, such that for all polynomial-time algorithms Dist:

$$|\Pr[\text{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \mathbf{Real}(\lambda)] -$$
$$\Pr[\text{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \mathbf{Ideal}_{\mathcal{L}_1, \mathcal{L}_2}(\lambda)]| \leq negl(\lambda),$$

where probabilities are taken over the coins of *KeyGen* and *Setup* algorithms .

The above definition captures strong adversarial capabilities, i.e., even adaptive adversaries that can select their new queries based on previous ones cannot learn anything more than the specified leakage functions $\mathcal{L}_1, \mathcal{L}_2$ ([11]). Next, we discuss these leakage functions in more detail.

Leakage Functions. Leakage $\mathcal{L}_1$ is associated with information that is leaked from the index alone (before any queries have been executed) and typically contains the size of the data collection N. Leakage $\mathcal{L}_2$ represents the information leaked during a query. It typically consists of the *search pattern* that indicates whether the client searches for a particular w, and the *access pattern* that contains the document identifiers matching the queried keyword w, namely $\mathcal{L}_2(\mathcal{D}, w) = (id(w), \mathcal{D}(w))$.

In the above, $id : \Delta \rightarrow \{0,1\}^\lambda$ is a mapping of keywords to λ-bit numbers. We refer to $id(w)$ as the *alias* of w. In practice, this will be a random allocation of keywords to aliases that is used to capture the search pattern leakage. That is, while $id(w)$ does not directly reveal w, when querying for the same keyword repeatedly the server observes the same $id(w)$. Recall that $\mathcal{D}(w)$ contains the document identifiers[4] matching the queried keyword w and this captures the *access pattern* leakage. More specifically, the access pattern consists of (i) the size of the result which we call *volume pattern*, and (ii) the document overlaps between previously queried keywords, which we call *overlapping pattern*.

SE through ORAM. One way to reduce the SE query leakage would be to replace all the memory accesses performed with oblivious memory accesses using an ORAM as a black box. In that case, the only leaked information during queries is the result size.

Attacks on deterministically-encrypted systems. Naveed et al. [34] proposed the *frequency analysis* and $\ell_\mathbf{p}$*-optimization* attacks that apply to databases encrypted with the use of deterministic schemes such as CryptDB [36].

The *frequency analysis* attack is the most basic and well-known inference attack in the area of cryptography. We define C_k and M_k to be the ciphertext and message spaces, respectively of the deterministic encryption scheme. Given a deterministically encrypted column $\mathbf{c}$ over C_k and an auxiliary dataset $\mathbf{z}$ over M_k, the attack works by assigning the i-th most frequent element of $\mathbf{c}$ to the i-th most frequent element of $\mathbf{z}$.

The $\ell_\mathbf{p}$*-optimization* attack is a family of attacks against deterministic encryption. The main goal is to find an assignment from ciphertexts to plaintexts that minimizes a given cost function, e.g., the $\ell_\mathbf{p}$ distance between the histograms of the dataset. This attack minimizes the total mismatch identified in frequencies across all plaintext and ciphertext pairs.

3 ENCRYPTED DATABASES FROM SEARCHABLE ENCRYPTION & ATTACKS

In this section we first show how SE can be used to support various queries on encrypted databases, such as point/group-by/join/range queries and then show various attacks (some existing and some

[4]We assume that the order of the documents does not reveal any significant information. This can be achieved by assigning a random λ-bit number to each document

new) on these constructions. Our findings systematically reestablish that using SE to implement encrypted databases [14, 18, 26] is particularly risky when the adversary is persistent and also has access to prior information about the underlying encrypted database (e.g., distribution of first names/gender). For snapshot adversaries that have no prior information about the encrypted database, there could be value in SE-based systems, however these are assumptions that are unlikely to hold in the real world [23, 34].

3.1 SE-based Point Queries

The most basic database query is the *point* query for a value v. A point query retrieves all the tuples from table $\mathcal{T}$ that contain value v in attribute x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x = v;$$

We can use an SE scheme to implement private point queries (e.g., see Demertzis et al. [14], and Kamara and Moataz [26]) by viewing attribute values as keywords, and database tuples as document identifiers. In this case an SE-based point query will return the encrypted tuples that match this value. We call this scheme POINT-SE. Note that POINT-SE can also be used to implement *group-by* queries (e.g., see Kamara and Moataz [26]), where the client can compute the group-by query through point queries for all distinct values of attribute x.

Attacks on POINT-SE. When using POINT-SE, the attacker can identify which encrypted tuples have the same value v, after he observes the execution of a query. Finally, after he observes the execution of all queries, the attacker can group the encrypted database tuples by value, and can therefore can compute the size of each group. By running a frequency analysis attack or an ℓ_{p}-optimization attack (described in Section 2), it is easy to map plaintext values to encrypted tuples. Note that the above attack requires the attacker to see all queries. However, in the case of group-by queries, the very nature of the query reveals all possible point queries, resulting in total leakage exposure with just a single query.

To conclude, observing all possible results from point queries (either one by one or via a group-by query) turns an SE-implemented database into a deterministically-encrypted database, making it vulnerable to simple attacks as described before.

3.2 SE-based Join Queries

A fundamental query type for relational databases is the *join* query. A simple join of two tables $\mathcal{T}$ and $\mathcal{R}$ on attribute x returns all pairs of tuples from $\mathcal{T}$ and $\mathcal{R}$ that agree on x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T}, \mathcal{R} \texttt{ WHERE } \mathcal{T}.x = \mathcal{R}.x;$$

A simple approach that allows us to support private join queries using SE is the following: We encrypt $\mathcal{T}$ with a semantically-secure encryption scheme and $\mathcal{R}$ with POINT-SE for private point queries on attribute x. Then we stream all the tuples of $\mathcal{T}$ to the client. Then the client decrypts each tuple $\mathtt{t}$ in $\mathcal{T}$ and queries the SE index for $\mathcal{R}$ (on attribute x) to retrieve the matching tuples of $\mathcal{R}$. Clearly this approach has high bandwidth since it requires streaming a large number of tuples to the client. We call this scheme JOIN-SE. To address the above bandwidth issue, Kamara and Moataz [26] propose a construction that, in the case of two tables $\mathcal{T}$ and $\mathcal{R}$, precomputes the answers to join queries on each possible attribute x. Then they store with SE a mapping from "keyword" x to the precomputed answer (i.e., pairs of pointers to tuples from $\mathcal{T}$ and $\mathcal{R}$ that have the same value on attribute x). This approach requires both significant amount of storage and setup time. We call this scheme JOIN-SE-PRECOMPUTE.

Attacks on JOIN-SE and JOIN-SE-PRECOMPUTE . It is easy to see that both schemes JOIN-SE and JOIN-SE-PRECOMPUTE leak the *encrypted join graph*. That is, for each encrypted tuple $\mathtt{t}$ of $\mathcal{T}$, the respective encrypted tuples $\mathtt{t}'$ of $\mathcal{R}$ that have the same value on x with $\mathtt{t}$ are leaked (if such tuples exist).

We propose a simple attack that recovers the values of the encrypted tuples: Assuming we have access to (part of) the plaintext dataset, we can compute the *plaintext join graph* by connecting with an edge tuples from $\mathcal{T}$ and tuples from $\mathcal{R}$ that have the same plaintext value on attribute x. If all tuples in $\mathcal{T}$ and $\mathcal{R}$ have at least one incident edge the attacker can perform the frequency analysis attack on both $\mathcal{T}$ and $\mathcal{R}$ and recover the plaintext values for the encrypted values of attribute x. In this case JOIN-SE and JOIN-SE-PRECOMPUTE provide exactly the same security properties for joins as more efficient encrypted systems based on deterministic encryption (e.g., CryptDB [36]). Otherwise the attack can be performed only on the leaked frequencies and JOIN-SE and JOIN-SE-PRECOMPUTE have potentially less leakage than systems based on deterministic encryption.

3.3 SE-based Range Queries

In the case of range queries, we want to retrieve all tuples from table $\mathcal{T}$ that contain value $v \in [l, u]$ in attribute x, i.e.,

$$\texttt{SELECT * FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x \geq l \text{ and } \mathcal{T}.x \leq u;$$

One way to support private range queries is to treat each numeric value of attribute x as a keyword and use SE. Then, private range queries can be supported by transforming the range $[l, u]$ to series of private point queries, i.e., searching for the individual values $l, l+1, \ldots, u-1, u$. We call this scheme RANGE-SE. Many attacks that exploit the overlapping and volume patterns exist against RANGE-SE—see [13, 22, 28–30, 32]. In general, these attacks first compute an ordering of the encrypted tuples and then retrieve the actual values after observing a certain number of queries.

To address this leakage, Faber et al. [18] and Demertzis et al. [14, 15] have proposed new private range constructions that use SE and are *response-hiding*, in that they do not leak overlaps between different range queries. Their main idea, called LOGARITHMIC-SRC in [14], builds a binary-tree data structure with some extra "internal" nodes (see Figure 1) on top of the database. Each leaf corresponds to a value $k \in \{0, 1, \ldots, M-1\}$ (where M is the size of the domain of attribute x) and stores *all* tuples that have value k at attribute x (i.e., a leaf can store more than one tuples). Data stored in a leaf is also copied to its parents. To answer a range search query, we select the root of the smallest subtree fully covering the query. The above data structure defines a natural key-value relationship, where each tree node is a key with the value being its respective database tuples. This allows us to query the data structure privately using SE.

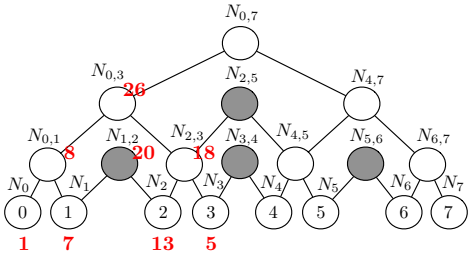


Figure 1: LOGARITHMIC-SRC [14, 15] consists of a full binary tree over the domain with an extra internal node between every two cousins. Red values denote the number of tuples each node contains (used for the proposed attack).

LOGARITHMIC-SRC yields up to $O(N)$ *false positives* where N is the size of the database table. For example, if the range $[3, 5]$ is being queried in Figure 1 and there is a single tuple in the range but the rest of the dataset has value 2, node $N_{2,5}$ will be returned and therefore the response will be the entire dataset. LOGARITHMIC-SRC-i, proposed for this problem [14], maintains two LOGARITHMIC-SRC-type binary trees, one on the domain $\{0, \ldots, M-1\}$ that stores constant-size metadata in the leaves (let us call this tree T_1) and one on the domain $\{0, \ldots, N-1\}$ that stores the actual database tuples in the leaves (one per leaf) sorted by the search attribute (let us call this tree T_2). In particular, for every value of the domain $i \in \{0, \ldots, M-1\}$, T_1 stores the subrange of $\{0, \ldots, N-1\}$ that corresponds to database tuples with value i in T_2. Therefore, a range query $[a, b]$ is transformed into two queries: One range query $[a, b]$ in T_1 that returns information that allows one to reconstruct the range $[a', b']$ of T_2 that contains the desired tuples, and finally one range query $[a', b']$ in T_2 that returns those tuples. This approach brings down the worst-case query cost from $O(N)$ to $O(R + r)$, where R is the size of the queried range (and is due querying T_1) and r is the size of the returned result (and is due to querying T_2).

Do existing attacks apply? Most existing attacks on RANGE-SE [13, 22, 28–30] do not apply to the above, response-hiding, schemes. However we must note that LOGARITHMIC-SRC and LOGARITHMIC-SRC-i leak the volume pattern and they may be vulnerable to volume pattern attacks despite the fact that none of the prior proposed volumetric attacks work for these schemes. Below, we propose such a new attack for LOGARITHMIC-SRC that could be extended also for LOGARITHMIC-SRC-i.

New attacks on LOGARITHMIC-SRC. The main idea is that if the attacker observes the volumes of all queries, then she could potentially reconstruct the tree and map encrypted database tuples to plaintext values.

For simplicity, let us focus on a small LOGARITHMIC-SRC tree with Dom = $\{0, 1, 2, 3\}$ (and therefore 8 nodes, including the one "extra" internal node—see Figure 1). Assume the adversary observes the following sizes of results (he actually sees the respective encrypted tuples as well): 20, 1, 26, 18, 8, 5, 7 and 13. His goal is to map these sizes (and the respective encrypted tuples) to the nodes N_0, N_1, N_2, N_3, N_{01}, N_{12}, N_{23} and N_{03} of the tree. The tuples that map to leaf i will therefore have value i!

To do the mapping the adversary exploits the fact that the size of a parent is equal to the sum of the sizes of its children and therefore sets up 4 linear equations with 8 unknowns $|N_0|$, $|N_1|$, $|N_2|$, $|N_3|$, $|N_{01}|$, $|N_{12}|$, $|N_{23}|$ and $|N_{03}|$. Of course these equations have an infinite number of solutions but the one we are interested in is a permutation of the observed sizes 20, 1, 26, 18, 8, 5, 7 and 13. In our example, due the fact that all pairwise sums are different, there is a unique assignment (up to a mirror arrangement), in particular the assignment $|N_0| = 1, |N_1| = 7, |N_2| = 13, |N_3| = 5, |N_{01}| = 8, N_{12} = 20, N_{23} = 18$ and $N_{03} = 26$. We note here that the described attack would not work in the case where pairwise-sums are not unique (e.g., when all leaves have size 1) but other information could be potentially used in that case. To conclude, this simple attack shows that concealing the overlapping pattern (as LOGARITHMIC-SRC is doing) is not enough for fully defending against range attacks.

Generalization of attack to LOGARITHMIC-SRC-i. Recall that in LOGARITHMIC-SRC-i we maintain two LOGARITHMIC-SRC-type trees: one for the metadata (T_1) and one for the actual data (T_2). Every leaf in T_1 has size *at most one* since a specific domain value may not be present at all in the database. Thus the above attack that exploits distinct sizes of leaves might not work very well.

However there are still ways to launch an attack. Coming back to Figure 1, consider the tree T_1 on the domain $\{0, 1, 2, 3\}$, with the difference that all leaf nodes have size either zero or one. Suppose after all queries have been issued on T_1 the adversary observes only three nodes of size one (and all other nodes have size zero). Looking into this information carefully, one can tell that these nodes have to be either N_0, $N_{0,1}$ and $N_{0,3}$ or N_3, $N_{2,3}$ and $N_{0,3}$ which implies that all database tuples have the same value *and* this value is either 0 or 3. Note that at that point, it will be easy to recover the topology of T_2 as well since for each range query one node of T_1 and one for T_2 will be accessed together. Exploring these attacks against such schemes in more detail is left as future work.

4 SEAL: ADJUSTABLE SEARCHABLE ENCRYPTION & DERIVED CONSTRUCTIONS

Most of the attacks on SE-based encrypted databases that were presented in the previous section exploit the leakage of SE such as the *search pattern*, the *overlapping pattern* and the *volume pattern*. In this section we propose SEAL, a family of new SE schemes with reduced leakage with the hope that these can be used to implement more secure (yet efficient) encrypted databases that withstand leakage-abuse attacks. Our main building blocks are an *adjustable ORAM*, an ORAM that allows one to define the bits of leakage of the index being accessed in a tunable manner, as well a *an adjustable padding* algorithm that adds noise to the actual size of the list being accessed.

4.1 Adjustable Oblivious RAM

An adjustable ORAM scheme (ADJ-ORAM-α) is parameterized by a parameter α that defines the number of leaked bits of the accessed memory location (for traditional ORAM it is $\alpha = 0$). We now

$bit \leftarrow \mathbf{Real}^{\text{ADJ-ORAM-}\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(\sigma_0, \mathsf{EM}_0) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}_0, \alpha), \perp)$.
3: **for** $k = 1$ to q **do**
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad ((v_{i_k}, \sigma_k), \mathsf{EM}_k) \leftrightarrow$ ADJ-OramAccess$((\mathsf{op}, i_k, v_{i_k}, \sigma_{k-1}), \mathsf{EM}_{k-1})$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

$bit \leftarrow \mathbf{Ideal}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(st_{\mathcal{S}}, \mathsf{EM}_0) \leftarrow$ ADJ-SimOramInitialize$(1^\lambda, \mathcal{L}_1^\alpha)$.
3: **for** $k = 1$ to q **do**
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad (st_{\mathcal{S}}, \mathsf{EM}_k) \leftrightarrow$ ADJ-SimOramAccess$(st_{\mathcal{S}}, \mathsf{EM}_{k-1}, \mathcal{L}_2^\alpha(i_k))$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

Figure 2: ADJ-ORAM-α real-ideal security experiments. With $m_0, m_1, \ldots$, we denote the messages exchanged at Line 5 of both experiments.

formally define the ADJ-OramInitialize and ADJ-OramAccess protocols of our ADJ-ORAM-α scheme:

- $(\sigma, \mathsf{EM}) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}, \alpha), \perp)$, takes as input a security parameter λ, a memory array M of n values (without loss of generality lets assume n is a power of 2) $(1, v_1), \ldots, (n, v_n)$, a parameter $\alpha \in \{0, 1, \ldots, \log n\}$ and outputs secret state σ (for client), and encrypted memory EM (for server).
- $((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-OramAccess$((\mathsf{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$ is a protocol between the client and the server, where the client's input is the type of operation op (read/write), an index i and the value v_i—for $\mathsf{op} = \texttt{read}$ client sets $v_i = \perp$. Server's input is the encrypted memory EM. Client's output consists of the updated secret state σ and the value v_i assigned to the i-th value of M if $\mathsf{op} = \texttt{read}$ (for $\mathsf{op} = \texttt{write}$ the returned value is $\perp$). Server's output is the updated encrypted memory EM.

Next, we define the security of ADJ-ORAM-α in the real/ideal game of Figure 2 parametrized by leakage functions $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$.

Definition 4.1. ADJ-ORAM-α is $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$-secure if for any PPT adversary Adv, there exists a PPT simulator containing algorithms (ADJ-SimOramInitialize, ADJ-SimOramAccess) such that

$$|\Pr[\mathbf{Real}^{\text{ADJ-ORAM-}\alpha}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda) = 1]|$$

is at most $neg(\lambda)$, where the above experiments are defined in Figure 2 and where the randomness is taken over the random bits used by the algorithms of the ADJ-ORAM-α scheme, the algorithms of the simulator and Adv.

The leakages $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$ are defined in a manner similar to those of SE, i.e., $\mathcal{L}_1^\alpha(\mathsf{M}) = (n, \alpha)$ and $\mathcal{L}_2^\alpha(i) = id^\alpha(i)$, where $id^\alpha(i)$ returns the α most significant bits of a random $\log n$-bit alias assigned to tuple (i, v_i). Intuitively, if two queries for index i are made on an ADJ-ORAM-α, the adversary should only figure out that the α most significant bits of the queried index are the same—but nothing else.

$(\sigma, \mathsf{EM}) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}, \alpha), \perp)$

1: Let M be in the form $(1, v_1), \ldots, (n, v_n)$ and $\mu = 2^\alpha$.
2: Sample a secret key $k \xleftarrow{\$} \{0,1\}^\lambda$.
3: Let π_k be a PRP: $\{0,1\}^\lambda \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^{\log_2 n}$.
4: Create $S_1, \ldots, S_\mu$ empty arrays of size $\frac{n}{\mu}$.
5: **for** $i = 1, \ldots, n$ **do**
6: $\quad$ Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and ϕ be the integer representation of the remaining bits of $\pi_k[i]$.
7: $\quad S_{\ell+1}[\phi + 1] = v_i$.
8: **for** $i = 1, \ldots, \mu$ **do**
9: $\quad (\sigma_i, \mathsf{EM}_i) \leftrightarrow$ OramInitialize$((1^\lambda, S_i), \perp)$.
10: Let EM to be $\mathsf{EM}_1, \ldots, \mathsf{EM}_\mu$ and σ to $(\sigma_1, \ldots, \sigma_\mu)$.
11: **return** (σ, EM).

$((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-OramAccess$((\mathsf{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$

1: Parse σ as $(\sigma_1, \ldots, \sigma_\mu)$ and EM as $(\mathsf{EM}_1, \ldots, \mathsf{EM}_\mu)$ where $\mu = 2^\alpha$.
2: Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and ϕ be the integer representation of the remaining bits of $\pi_k[i]$.
3: $\ell = \ell + 1$ and $\phi = \phi + 1$.
4: $((v_i, \sigma_\ell), \mathsf{EM}_\ell) \leftrightarrow$ OramAccess$((\mathsf{op}, \phi, v_i, \sigma_\ell), \mathsf{EM}_\ell)$.
5: **return** $(v_i, \sigma, \mathsf{EM})$.

Figure 3: ADJ-ORAM-α using any ORAM as a black box.

Construction of ADJ-ORAM-α. The main idea behind our approach is that the memory array will not be stored in one ORAM, but it will be partitioned into multiple disjoint subsets, each of which will then be stored in a separate smaller ORAM. We use as a black box any secure ORAM= (OramInitialize, OramAccess) to store each subset. Our construction works by building 2^α different ORAMs $\text{ORAM}_1, \ldots, \text{ORAM}_{2^\alpha}$, each of which will store a part of M of size $n/2^\alpha$.

One possible way to partition M into these ORAMs would be to deterministically assign (i, v_i) based on their location in M, i.e., the first 2^α entries will be stored in ORAM_1, the next 2^α entries will be stored in ORAM_2 and so on. However, this might reveal sensitive information for certain application settings, e.g., if the server knows that M stores v_i in a sorted manner, then accessing ORAM_1 reveals that one of the smallest values in M was accessed. Hence, before performing the partitioning, we randomly permute M using a PRP P over $[1, n]$ (implemented with a small-domain PRP [21, 33, 37]), for which the key k is chosen and stored by the client. Let π_k be the corresponding mapping after k has been chosen. Then, the partitioning of M is performed using the integer representation of the α most significant bits of the permuted index and the remaining bits of $\pi_k(i)$ correspond to the index $\pi_k(i)$ of tuple (i, v_i) inside the small ORAM. Our construction is given in Figure 3.

Theorem 4.2. *Assuming* (OramInitialize, OramAccess) *is a secure ORAM and π_k is a secure PRP, then* ADJ-ORAM-α *presented above is $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$-secure, according to Definition 4.1.*

Proof. The ORAM scheme used is secure and therefore we use its algorithms SimOramInitialize and SimOramAccess. In particular, the ADJ-SimOramInitialize takes as an input $\mathcal{L}_1^{\alpha} = (n, \alpha)$ and the security parameter λ, and it creates $\mathsf{EM}_1, \ldots \mathsf{EM}_{\mu}$ and $\sigma_1, \ldots, \sigma_{\mu}$ using SimOramInitialize$(1^{\lambda}, \frac{n}{\mu})$ for $\mu = \frac{n}{2^{\alpha}}$. The ADJ-SimOramAccess takes as an input $id^{\alpha}(i)$, from $\mathcal{L}_2$ leakage, which determines in which encrypted memory EM_i must be accessed, and performs a random access using SimOramAccess$(\sigma_i, \mathsf{EM}_i)$. Then, the simulator properly updates EM_i and σ_i. □

Performance and leakage of ADJ-ORAM-α. The higher the value of α is, the more efficient ADJ-ORAM is (ORAM is applied on a smaller parts of the array) and the larger the leakage becomes (more accesses will be made on the same small parts of the array). Concretely, if we assume that the ORAM used as a building block has $T(n)$ access overhead (e.g., $T(n) = O(\log n)$ for the most efficient ORAM [35]), then ADJ-ORAM-α has an improved $T(n/2^{\alpha})$ overhead. In Section 4.3 we discuss how ADJ-ORAM-α can be instantiated using [38] and oblivious data structures [42] and we provide a more concrete performance analysis.

4.2 Adjustable Padding

In this section we propose *adjustable padding*, another primitive that will help us build more secure SE schemes. Recall that existing SE schemes leak the query result size, i.e., $|\mathcal{D}(w)|$. In particular in a dataset with size N a keyword list can have N different sizes. One way to eliminate this leakage is by *padding* all the keyword lists $D(w)$ to the same size N (worst-case padding). However, this would introduce a prohibitive storage/search overhead. To avoid this overhead, one could pad to the closest power of two, forcing the adversary to observe at most $\log N + 1$ sizes—thus leaking $\log \log N + 1$ bits, at most doubling the search and storage overhead.

Our proposal is a generalization of the above idea. Our padding can be parameterized by a value x that defines the number of different sizes (which are exactly $\lceil \log_x N \rceil + 1$) that the adversary can observe. Our padding algorithm works as follows (see Figure 4). Given a keyword list $\mathcal{D}(w)$ of size, we find the integer i such that $x^{i-1} < |\mathcal{D}(w)| \leq x^i$. Then we pad the list $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy entries. Note that this padding strategy can increase the space and search overhead by a factor of x and yields leakage of $\log \log_x N + 1$ bits! In other words the larger x is, the less efficient the scheme becomes and the less leakage the adversary observes. We note here that for simulation purposes, after all lists are padded, our algorithm pads the dataset to a total of $x \cdot N$ entries so that to avoid leaking any information about the dataset.

We note here that padding techniques have been used before for concealing the size of the accessed result (e.g., see Cash et al. [10] and Bost and Fouque [8]). However, these approaches depend on the distribution of the input dataset, which leads to more leakage, even prior to query execution. Instead our padding algorithm is *distribution-agnostic* and can thus be simulated only by knowing the size of the dataset N and the padding parameter x.

4.3 SEAL

We now present SEAL(α, x), our adjustable SE construction that uses ADJ-ORAM-α, ADJ-PADDING-x and an oblivious dictionary

$\mathcal{D} \leftarrow$ ADJ-Padding$(x, \mathcal{D})$

1: $N = |\mathcal{D}|$.
2: **for** each keyword w in $\mathcal{D}$ **do**
3: Find the smallest i: $x^{i-1} < |\mathcal{D}(w)| \leq x^i$.
4: Pad $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy values.
5: Pad $\mathcal{D}$ with dummy records so that the total size is $x \cdot N$.
6: **return** the padded dataset.

Figure 4: ADJ-Padding-x leading to $\log_x N$ different sizes.

ODict described in Section 2 as a black boxes. We recall that parameter α is defined in the range $[0, \log N]$ and that for $\alpha = 0$ all the search/overlapping pattern bits are protected, and for $\alpha = \log N$ all bits are leaked. Also for larger x values, less volume pattern bits are leaked—e.g., for value $x = N$ no volume pattern bits are leaked.

Construction of SEAL(α, x). SEAL(α, x) is defined similarly with SE (see Section 2) and it contains algorithms/protocols Setup and Search. Our construction is described in Figure 5.

SEAL's setup takes as input dataset $\mathcal{D}$. Parameters α and x are considered public and we do not provide them as input explicitly. First, it uses ADJ-PADDING$(x, \mathcal{D})$ in order to transform $\mathcal{D}$ to a new dataset with at most $\log_x N + 1$ distinct results sizes (see Line 2 of setup). Then, it sorts all the (w, id) pairs in lexicographical order (see Line 3 of setup) and places them sequentially in a memory array M which is then given as input to the ADJ-OramInitialize algorithm (see Line 8 of setup). The sorting guarantees that all (w, id) for the same keyword w will be placed in consecutive memory locations. All entries for w can then be retrieved if one knows the index of the first appearance of w and the size of the padded list $|\mathcal{D}(w)|$. For every keyword w, this information is stored in an oblivious dictionary T (see Line 7 of setup).

SEAL's search takes as input the queried keyword w, client's secret state st_C and the encrypted index $\mathcal{I}$, which contains the small oblivious memories $\mathsf{EM}_1, \ldots$ as well as the oblivious dictionary T. For a given queried keyword w, the client first performs an access to the oblivious dictionary to retrieve the index of the first appearance of w in M and the padded result size (cnt_w) (see Lines 2-3 of search). Then, it performs cnt_w accesses in the ADJ-ORAM-α in order to retrieve the result $\mathcal{X}$ (see Lines 4-7 of search). Note that, due to padding, $\mathcal{X}$ may contain "dummy" records which will be filtered out by the client afterwards.

Leakage Definition for SEAL(α, x). SEAL(α, x) is secure according to the standard SE/OSE definition described in Section 2 with the following leakage functions

$$\mathcal{L}_1^{\alpha, x}(\mathcal{D}) = (N, \alpha, x) \text{ and } \mathcal{L}_2^{\alpha, x}(\mathcal{D}, w) = \mathcal{D}_{\alpha}^{x}(w),$$

where $\mathcal{D}_{\alpha}^{x}(w)$ contains the α most significant bits of the aliases of the document identifiers in the padded list $\mathcal{D}(w)$ as output by algorithm ADJ-PADDING$(x, \mathcal{D})$. For the rest of the paper we will simply denote these leakages as $\mathcal{L}_1$ and $\mathcal{L}_2$.

Theorem 4.3. *Assuming that* ODict *is a secure oblivious data structure according to [42] (Def. 1)*, ADJ-ORAM-α *is secure according to Def. 4.1,* SEAL(α, x) *is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure according to Def. 2.1.*

$(st_C, \mathcal{I}) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{D})$

1: Set $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$, where $\mathbf{W}$ is the set of keywords in $\mathcal{D}$.
2: $\mathcal{D} \leftarrow$ ADJ-PADDING$(x, \mathcal{D})$. ▷ Parameter x is public.
3: Let M be an array of N entries storing (w, id) pairs of $\mathcal{D}$ in lexicographic order and i_w be the index of w's first occurrence in M.
4: $(T, \sigma_{\text{odict}}) \leftarrow$ ODictSetup$(1^\lambda, N)$.
5: **for** all $w \in \mathbf{W}$ **do**
6: Let $cnt_w = |\mathcal{D}(w)|$.
7: $(\sigma_{\text{odict}}, T) \leftrightarrow$ ODictInsert$((w, i_w||cnt_w, \sigma_{\text{odict}}), T)$.
8: $(\sigma_{\text{oram}}, \mathsf{EM}) \leftarrow$ ADJ-OramInitialize$(1^\lambda, \mathsf{M}, \alpha)$. ▷ Parameter α is public.
9: $st_C = (\sigma_{\text{oram}}, \sigma_{\text{odict}})$ and $\mathcal{I} = (\mathsf{EM}, T)$.
10: **return** $(st_C, \mathcal{I})$.

$((\mathcal{X}, st_C), \mathcal{I}) \leftrightarrow \mathsf{Search}((st_C, w), \mathcal{I})$

1: Parse $\mathcal{I}$ as (EM, T) and st_C as $(\sigma_{\text{odict}}, \sigma_{\text{oram}})$ and let $\mathcal{X}$ be empty.
2: $((\mathtt{value}, \sigma_{\text{odict}}), T) \leftrightarrow$ ODictSearch$((w, \sigma_{\text{odict}}), T)$.
3: Parse value as $(i_w||cnt_w)$.
4: **for** $i = i_w, \ldots, i_w + cnt_w$ **do**
5: $((v_i, \sigma_{\text{oram}}), \mathsf{EM}) \leftrightarrow$ ADJ-OramAccess$((read, i, \perp, \sigma_{\text{oram}}, \alpha), \mathsf{EM})$. ▷ Parameter α is public.
6: $\mathcal{X} \leftarrow \mathcal{X} \cup v_i$.
7: **return** $(\mathcal{X}, st_C, \mathcal{I})$.

Figure 5: Our SEAL(α, x) scheme using ADJ-ORAM-α, ADJ-PADDING-x, and an oblivious dictionary as black boxes.

Proof. ADJ-ORAM-α is secure—our proof uses simulator algorithms ADJ-SimOramInitialize and ADJ-SimOramAccess. The security parameter λ is given. The SimSetup takes as an input $\mathcal{L}_1 = (N, \alpha, x)$. SimSetup initializes $(T, \sigma_{\text{odict}}) \leftarrow$ ODictSetup$(1^\lambda, N)$ and it inserts N random entries of the form $(w, i_w||cnt_w)$ in the oblivious dictionary T using ODictInsert. Then, it computes $N' = x \cdot N$. Finally, it uses ADJ-SimOramInitialize$(1^\lambda, N', \alpha)$ to create the encrypted memory EM and state σ_{oram}. The SimSearch algorithm takes as an input $\mathcal{L}_2$ and performs one random access in the oblivious dictionary T using ODictSearch, and calls $|\mathcal{D}^x_\alpha(w)|$ times the ADJ-SimOramAccess with input the α-bit identifiers in $\mathcal{D}^x_\alpha(w)$ ($\mathcal{D}^x_\alpha(w)$ has the required leakage for ADJ-SimOramAccess). Then, the simulator updates EM, T and the states σ_{odict}, and σ_{oram}. □

Asymptotic Performance. Let $(T(n), C(n), S(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying ORAM used and let $(t(n), c(n), s(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying oblivious dictionary used. The server space required is always $S(x \cdot N) + s(N)$. Now, assuming the client keeps, along with the oblivious dictionary state, the ORAM states locally, the search complexity for a keyword w is

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot T\left(\frac{x \cdot N}{2^\alpha}\right)$$

and the client space is $2^\alpha \cdot C(x \cdot N/2^\alpha) + c(N)$. Assuming the client does not keep ORAM states locally and just downloads and re-encrypts to the server, the search complexity for w becomes

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot \max\left\{T\left(\frac{x \cdot N}{2^\alpha}\right), C\left(\frac{x \cdot N}{2^\alpha}\right)\right\}$$

and the client space is just $c(N)$. Whether one chooses to store the local states locally or oursource them depends on the parameter α. For example, for small values of α it is better to keep them locally, while for larger values of α it might worth outsourcing.

Implementing ADJ-ORAM-α. We implement each small ORAM in ADJ-ORAM-α with Path-ORAM [38]. Recall that the cost of Path-ORAM for accessing n blocks of size B is $B \log n$ for accessing the path and $O(\log^3 n)$ for recursively updating the position map. In our case we apply Path-ORAM on $N/2^a$ blocks of size around $2 \log N$ bits ($\log N$ bits for storing keyword w and $\log N$ bits for storing the id) and therefore our total cost is $O(\log N \log(N/2^a) + \log^3(N/2^a))$.

Implementing SEAL(α, x). For SEAL(α, x), apart from ADJ-ORAM-α as described above, we also use an oblivious dictionary ODict (for storing $i_w||cnt_w$) implemented with an oblivious AVL tree [42] (this requires $b \log^2 N$ additional additive cost where b is the bit-size of $i_w||cnt_w$). In case the number of keywords/attributes $|\mathbf{W}|$ in small, we choose to keep the dictionary locally—this requires around $3|\mathbf{W}| \log N$ bits which in practice is a few megabytes and is a common assumption in Dynamic SE [7, 9, 19, 39]. Our experiments in the next section assume the dictionary is kept locally. Note that even if we do not keep the dictionary locally, we only require one oblivious access to it per query w. This is most of the times subsumed by the required $|\mathcal{D}(w)|$ ADJ-ORAM-α queries, especially when $|\mathcal{D}(w)|$ is large (e.g., $\Omega(\log^2 N)$). In any case we can always reduce the above cost with an adjustable oblivious dictionary at the expense of leaking α bits of the search pattern. Finally, in case the worst-case overhead of SEAL(α, x) becomes higher than sequential scan (which has no leakage), we perform a sequential scan.

4.4 New Constructions for Point & Join Queries

In Section 3 we presented/reviewed three constructions for point and join queries on encrypted databases that use *SE as a black box*:

(i) POINT-SE, a construction for point queries on encrypted data; (ii) JOIN-SE and JOIN-SE-PRECOMPUTE, two constructions for join queries on encrypted data.

Our proposed new constructions reduce the leakage of the above constructions by using SEAL(α, x), instead of simple SE. By doing this replacement we have the following constructions, for various parameters of α and x,

(1) POINT-ADJ-SE;
(2) JOIN-ADJ-SE.

Note that JOIN-ADJ-SE can be instantiated either by using JOIN-SE or JOIN-SE-PRECOMPUTE as basis.

4.5 New Constructions for Range Queries

The first adjustable construction that we propose for range queries, RANGE-ADJ-SE-(a, x), is based on the "naive" construction RANGE-SE from Section 3.3, where instead of simple SE we use SEAL(a, x).

Our second construction, RANGE-SRC-SE-(a, x) comprises two modifications LOGARITHMIC-SRC-i [14] so that the potential attack presented in Section 3.3 is mitigated. Recall the attack works by exploiting volumes exposed by tree T_1 which (the tree T_1) stores metadata required to search tree T_2.

Our first modification of LOGARITHMIC-SRC-i is a simple one: Instead of outsourcing tree T_1 using SE, keep tree T_1 locally unencrypted and therefore previously exposed volume information will not be available. The only downside is the $O(|\mathbf{W}|)$ client storage that is required to store T_1, where $\mathbf{W}$ is the set of values of the range attribute. In practice this storage is minimal, e.g., none of the ranges of the attributes shown in Table 1 of our evaluation exceed 1MB. (Of course, if strictly necessary, we can outsource tree T_1 to the server via an oblivious dictionary without any leakage, increasing the search time by a polylog factor.)

RANGE-SRC-SE-(α, x). However, the above modification addresses the leakage only in T_1. But T_2 can also leak information. For example, (a) if the same tree node is accessed twice, there is nonzero probability that the same range is being queried, and (b) the result size (or an upper bound of it) is leaked from accessing T_2.

To reduce the effect of leakages (a) and (b), one could reduce the number of sizes observed by the adversary by implementing the encrypted index for T_2 using SEAL(α, x) instead of simple SE.

Our second modification that yields our final scheme RANGE-SRC-SE-(α, x) does almost that, but it does *not* use ADJ-Padding for reducing the volume pattern leakage—this would blow up the space to $O(xN \log(xN))$. Instead RANGE-SRC-SE-(α, x) reduces the number of sizes that are being observed to $\log_x N + 1$ by storing only as many *equally distributed* levels from T_2. E.g., for $x = 2$ all levels are stored, for $x = 4$ half of the levels are stored, while for $x = 16$ one fourth of the levels are stored. Note that by this approach the search complexity becomes $O(x \cdot r)$ and the space becomes $O(N \log_x N)$.

5 EVALUATION AGAINST ATTACKS

To benchmark the effectiveness of our proposed adjustable constructions POINT-ADJ-SE, JOIN-ADJ-SE and RANGE-SRC-SE, we could use existing state-of-the-art leakage-abuse attacks [10, 13, 22, 24, 28, 30]. However, these attacks are very sensitive to the *exact* overlapping or volume pattern leakage (e.g., for ordering the records in range queries), which is not available in our adjustable constructions.

We introduce instead a new class of attacks where the adversary tries to work with only the available bits of leakage, and at a high level, tries to guess the rest of the bits. Also, our adversary is *all-powerful*, having plaintext access to the input dataset. We stress that this is a "heavy" benchmark that already covers known attacks [10, 13, 22, 24, 28, 30]. This is because if our adjustable constructions reduce the success rate of such a powerful attacker, a more realistic attacker with partial knowledge of the dataset would perform even worse (assuming the same attack strategy is followed). We now describe the attacker model in detail.

5.1 Attacker Model

Our adversary has two goals:

(1) First, to perform a *query recovery attack*, namely decrypting the client encrypted queries;
(2) Second, to perform a *database recovery attack*, namely mapping encrypted tuples to the (decrypted) client queries.[5]

We note here that a database recovery attack in the case of SE (where $\alpha = \log N$) is trivial, since which encrypted records mapping to which client queries is information that is contained in the leakage itself. This task becomes more challenging for smaller values of α where this information is not given in its entirety.

For our experiments, we define the query recovery success rate QR_{SR} as the ratio of the number of correctly decrypted queries over the total number of considered queries. We also define the database recovery success rate DR_{SR} as the ratio of the number of tuples that have been correctly mapped to (decrypted) client queries over the total number of considered tuples.

5.2 Experimental Setup

Our experiments were conducted on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto and the bouncy castle library [2] for the cryptographic operations. Our java implementation does not use hardware supported cryptographic operations. However, this does not affect our conclusions. The use of hardware supported cryptographic operations can further improve the absolute time for construction and search, but it will not affect the comparison for different parameters α and x.

We consider the following two datasets in our experimental evaluation. For attacking POINT-ADJ-SE-(α, x), we use a real dataset consisting of 6,123,276 tuples with 22 attributes of reported incidents of crime in Chicago [3]. For attacking POINT-ADJ-SE-(α, x), JOIN-ADJ-SE-(α, x), and RANGE-ADJ-SE-(α, x), we used the TPC-H benchmark [4] with scaling factor 0.1 which is widely used by the database community[6]. TPC-H consists of eight separate tables (PART, SUPPLIER, PARTSUPP, CUSTOMER, NATION, LINEITEM, REGION, ORDERS). Our attacks take as input the leakage of all

[5] Note here that figuring out the mapping from encrypted tuples to decrypted client queries allows the adversary to learn information about the content of the encrypted tuples. E.g., in the case of point queries once you map the encrypted tuples to a recovered query q then you know that one attribute of the encrypted tuples is q.
[6] We do not provide an evaluation for group-by queries since the results are identical to those for point queries (after observing all the distinct queries).

$QR_{SR} \leftarrow$ QueryRecoveryAttack$(\mathcal{T}, \{t_q, |q|\}_{q \in Q})$

Input: Plaintext database table $\mathcal{T}$ and set of tokens t_q to be decrypted along with their volumes $|q|$.
Output: The success rate QR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow$ ADJ-Padding$(x, \mathcal{T})$.
2: Set correct = 0.
3: **for** each token t_q **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |q|\}$.
5: Remove q' from $\mathcal{T}$.
6: **if** q' is the correct value for t_q **then**
7: correct++.
8: **return** correct$/|Q|$.

Figure 6: Query Recovery Attack for Point Queries.

$DR_{SR} \leftarrow$ DatabaseRecoveryAttack$(\mathcal{T}, \{t_q, S_q\}_{q \in Q})$

Input: Plaintext database table $\mathcal{T}$ and set of tokens t_q to be decrypted along with their set S_q of α-bit identifiers of encrypted tuples.
Output: The success rate DR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow$ ADJ-Padding$(x, \mathcal{T})$.
2: Set correct = 0.
3: **for** each pair of token/id-set (t_q, S_q) **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |S_q|\}$.
5: Remove q' from $\mathcal{T}$.
6: **for** each $id \in S_q$ **do**
7: Choose random `suffix` such that id||`suffix` has not been used before with q'.
8: **if** q' is the correct value for t_q and `suffix` are the correct remaining bits of id **then**
9: correct++.
10: **return** correct$/\sum |S_q|$.

Figure 7: Database Recovery Attack for Point Queries.

possible queries (worst-case leakage). The same attacks can be run with less queries, leading to lower success rate. When evaluating the performance of SEAL(α, x) we store the oblivious dictionary locally.

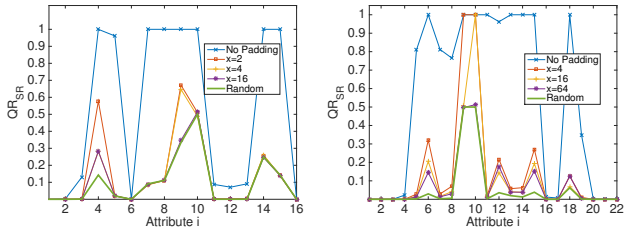
We denote with $x = \perp$ the lack of padding, where the attacker can observe up to N distinct result sizes.

5.3 Attacking POINT-ADJ-SE

We evaluate the effectiveness of POINT-ADJ-SE-(α, x) against our new query/database recovery attacks presented in Figures 6 and 7. In both attacks we consider one attribute of one table at a time.

Our *query recovery* attack (see Figure 6) is very simple and uses only volume pattern leakage. Having access to the plaintext table $\mathcal{T}$, the adversary computes the new padded table for the queried attribute (Line 1 in Figure 6) using the padding parameter x. Now, for a given encrypted query q with size $|q|$ the adversary uses $\mathcal{T}$ to find the candidate plaintext values which have size $|q|$, and chooses one of them at random (see Line 4 in Figure 6). Note that the higher the value of x is, the larger the set of possible values in Line 4 is therefore reducing the success rate of the attack.

The *database recovery* (see Figure 7) works as follows. First the adversary is using the x-padded volume pattern to decrypt which keyword we are querying, as before. Then, for each returned α-bit tuple identifier id the adversary picks the remaining suffix bits at random and returns the respective tuple (see Lines 6-9 in Figure 7).

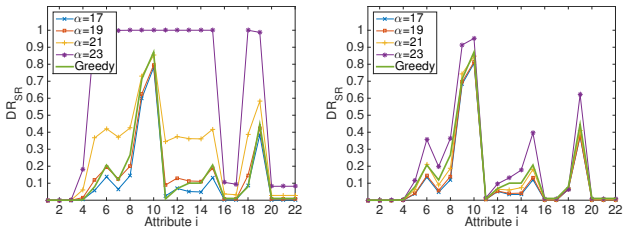


(a) LINEITEM (TPC-H) (b) Crime Dataset

Figure 8: Query Recovery Attack against POINT-ADJ-SE for various x.

If both the keyword is correctly recovered and the suffix is correctly guessed (the probability of which drops as α decreases), then the mapping of the specific tuple to the recovered keyword is correct.

Query Recovery Attack Evaluation. Figures 8(a), and 8(b) show the evaluation of POINT-ADJ-SE-(α, x) against the query recovery attack. We only vary x since α does not affect the effectiveness of the attack. Figure 8(a) demonstrates the evaluation for the LINEITEM table (TPC-H benchmark), while Figure 8(b) presents the results for the Crime dataset. In all figures, we additionally report the attacker's query recovery success rate if she just maps encrypted queries to plaintext values at random, i.e., $1/|\mathbf{W}|$—ideally, the success rate of our attack should be as close as possible to this "Random" approach.



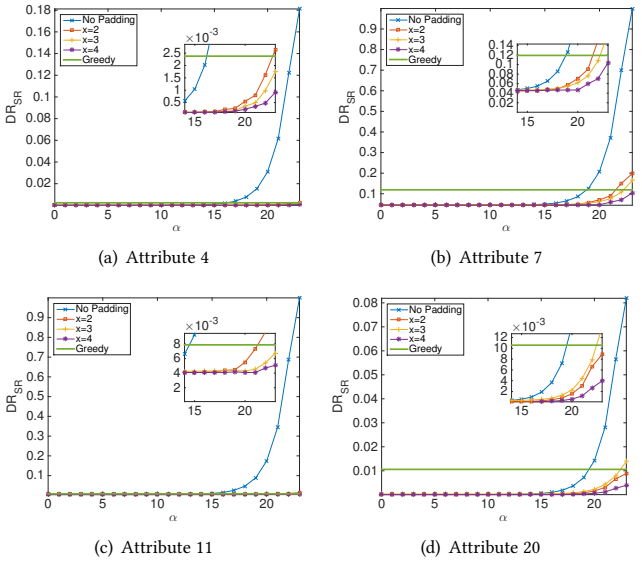
(a) $x = \perp$ (b) $x = 2$

Figure 9: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. We show all attributes.

In Figure 8(a), for $x = 2$ (only a 2× overhead in search time and storage), we see that our scheme forces the attacker to perform very close to "Random" for 14 out of 16 attributes. We observe that QR_{SR} for attribute 8 is close to Random for $x = 16$, while for attribute 4 greater values of x are needed. Let us look why this is the case for, say, attribute 8: There are only three values that can be queried with highly-skewed result sizes $|q_1|$ =1, $|q_2|$ =1,000 and $|q_3|$ =100,000. Therefore the larger the number of padded sizes is, the more likely it is that each q_i will be mapped to a distinct padded size, allowing the attacker to still distinguish between these queries. We observe similar patterns for the other tables of TPC-H and we report in Appendix the results for tables ORDERS and PART (Figure 15).

In Figure 8(b) we repeat the same experiment for the 22 attributes of the crime dataset, and we observe that in 17 out of 22 attributes for $x = 4$ (up to 4× performance degradation) the attacker's QR_{SR} significantly drops and is close to the Random approach. For attributes 6, 8, 10, 12, 15 greater values of x are needed again due to the small number of values that these attributes have. Finally, we observe that in attributes 15 and 18, QR_{SR} is higher for $x = 64$ than for $x = 4$, which is counterintuitive. This is because the query sizes of the values in these attributes are distributed in a way that for $x = 4$ there are less distinct sizes than for $x = 64$.

Database Recovery Attack Evaluation. As discussed above the database recovery attack is based on the query recovery one. Thus, due to lack of space we focus on the 22 attributes of the crime dataset in which QR_{SR} is higher than the one in the TPC-H dataset. Figure 9 shows the attacker's success rate for the database recovery attack (DR_{SR}) for $\alpha = (17, 19, 21, 23)$ ($\alpha = 23$ corresponds to SEAL($\log N, x$)) and for $x = \perp$ and $x = 2$. Recall that in our threat model the attacker has plaintext access to the input dataset, so for the database recovery attacks we report as a reference point a greedy strategy that the adversary may follow, in which she maps all encrypted tuple/tuple-ids to the most frequent plaintext value (guessing heuristically). E.g., for a binary attribute if the most frequent value appears in the 70% of the tuples/tuple-ids then the adversary achieves $DR_{SR} = 70\%$ by following the greedy strategy. Ideally, the goal is to find α as close as possible to $\log N$ and the smallest possible value of x, while DR_{SR} is below the greedy strategy. As is shown in Figure 9 for $\alpha = \log N - 2 = 21$ and $x = 2$ the attacker's success rate is always below the success rate of the greedy strategy. In Figure 10, we provide a more detailed evaluation for 4 specific attributes of the crime dataset for $\alpha \in [0, \log N]$ and $x = \perp, 2, 3, 4$.



(a) Attribute 4 (b) Attribute 7

(c) Attribute 11 (d) Attribute 20

Figure 10: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. Attributes 4,7,11,20.

5.4 Attacking JOIN-ADJ-SE

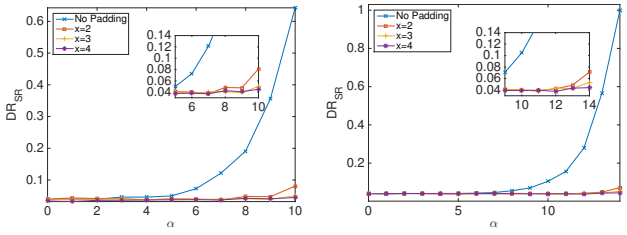
We evaluate the effectiveness of JOIN-ADJ-SE-(α, x) using the database recovery attack proposed for point queries (see Figures 7). Since the database schema and the size of each table are usually not considered private information, we do not consider join query recovery attacks.

Attack Evaluation. Figure 11 demonstrates the database recovery attack for foreign-key join queries. We consider foreign-key joins between tables (i) SUPPLIER and NATION—Figure 11(a), and (ii) CUSTOMER and NATION; the TPC-H benchmark contains only foreign-key joins. We observe in Figure 11(b) the DR_{SR} for $\alpha = [0, \log N]$, and $x = \perp, 2, 3, 4$. For $\alpha = 0$ and $x = \perp$, DR_{SR} is 65% in Figure 11(a) and 97% in Figure11(b), but for $\alpha = \log N - 1$ and $x = 2$, DR_{SR} drops below 6%. We conducted all the possible foreign-key joins and we observe the same pattern.

5.5 Attacking RANGE-SRC-SE

We evaluate the effectiveness of RANGE-SRC-SE-(α, x) scheme for various x against slightly modified versions of the attacks for point queries (Figures 6 and 7). In particular in Line 2 of both Figure 6 and 7, we do not perform padding but we recreate T_2 in plaintext with only $\log_x N + 1$ evenly distributes levels. We report as a baseline a scheme that does not perform padding but hides the entire overlapping pattern leakage. For the case of query recovery attack we set $\alpha = \log N$ for RANGE-SRC-SE-(α, x), since varying α does not affect the effectiveness of the attack.

Attack Evaluation. We focus on numeric attributes PS_SupplyCost from table PARTSUPP; P_Size and P_RetailPrice from table PART; L_TAX, L_QUANTITY, L_DISCOUNT from table LINEITEM. The attributes PS_SupplyCost and P_RetailPrice were transformed from floating point decimal numbers to integers with rounding. Table 1 presents for each attribute the number of all possible range queries



(a) SUPPLIER ⋈NATION (b) CUSTOMER ⋈NATION

Figure 11: Database Recovery Attack for Foreign-key Join Queries for the TPC-H Benchmark.

Attribute	#Queries	# Correctly Decrypted Queries			
		Baseline	RANGE-SRC-SE		
			$x = 2$	$x = 4$	$x = 16$
PS_SupplyCost	500500	73446	14	6	2
P_Size	1275	1184	10	5	2
P_RetailPrice	519690	19555	18	5	2
L_Tax	45	45	8	5	3
L_Quantity	1275	1263	10	4	3
L_Discount	66	66	8	4	1

Table 1: Query Recovery Attack for Range Queries

and the number of the correctly decrypted ones using the baseline (Column 3 of Table 1), and RANGE-SRC-SE for $x = 2$, $x = 4$ and $x = 16$ (Columns 4, 5, 6 of Table 1). We observe that $x = 16$ drastically reduces the number of correctly decrypted queries. We omit the presentation of the database recovery attacks for ranges, since DR_{SR} is primarily based on the result of the query recovery attack, and we see in Table 1 that even for $x = 2$ QR_{SR} is small.

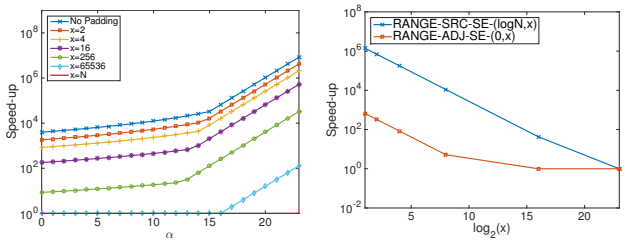
5.6 Efficiency of adjustable constructions

Figure 12(a) shows the smallest speedup achieved by our construction SEAL(α, x) compared to an approach that performs sequential scan and has no leakage, and for various values of α and x. Similarly in Figure 13(a) we show the largest slowdown of SEAL(α, x) compared to simple SE which has the maximum leakage. We do an analysis of these plots in the next section.

Figure 12(b) and 13(b) evaluate RANGE-ADJ-SE-$(0, x)$ and RANGE-SRC-SE-$(\log N, x)$. Note that both schemes hide the overlapping pattern, the first by using ORAM, the second by construction. Also both schemes are using the same x, allowing the adversary to observe the same number of different sizes (but not necessarily the same sizes). Note that RANGE-SRC-SE performs much better than RANGE-ADJ-SE. This is to be expected given RANGE-SRC-SE has more leakage—the search pattern, which however we do not know how to use in an attack here.[7]

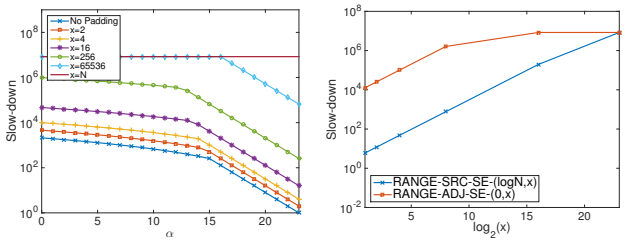
In Appendix, we provide additional experiments regarding the performance of our SEAL scheme. We show experiments for values of α and x that significantly mitigate the proposed attacks and achieve good performance (as we also discuss in the next section). In Figure 16, we evaluate the required index size and construction time of SEAL for a dataset with 8 million tuples and $x = 1, 2, 3, 4$. Finally, in Figures 17 and 18 we evaluate the search time of our SEAL

[7] Although the search pattern (combined with the access pattern) has been used in recent work by Kornaropoulos et al. [29] to attack RANGE-SE, it is not clear how it can be used for RANGE-SRC-SE-(α, x).



(a) SEAL(α, x) (b) Range Schemes

Figure 12: Speedup from sequential scan.



(a) SEAL(α, x) (b) Range Schemes

Figure 13: Slowdown from SE.

scheme for two attributes of the crime dataset for $\alpha = 20, 21, 22, 23$ and $x = \perp, 2, 3, 4$.

5.7 Setting parameters α and x in practice

We observe that for point and join queries setting $\alpha = \log N - 3$ and $x = 4$ significantly reduces both QR_{SR} and DR_{SR}, while for these values the smallest speedup from sequential scan is more than $262,000\times$ and the maximum slow-down from SE is $32\times$. There rare cases that attributes with skewed distribution and small number of distinct values, e.g., binary attributes, require higher values of x, e.g., $x = 16$ or $x = 64$. In the cases of range queries, we observe that our RANGE-SRC-SE-$(\log N, x)$ for $x = 16$ significantly mitigates our all-powerful query recovery attack and achieves a maximum $32\times$ slowdown from insecure RANGE-SE.

6 CONCLUSION

In this work we show the necessity of new defense mechanisms (beyond SE) for encrypted databases. We propose SEAL a family of new SE schemes with adjustable leakage which can be used for building efficient encrypted databases (for point, range, group-by and joins queries) that are robust against all-powerful attacks. We hope that SEAL be used as benchmark for measuring the robustness of previous/future leakage-abuse attacks.

REFERENCES

[1] [n.d.]. Attack of the week: searchable encryption and the ever-expanding leakage function. https://blog.cryptographyengineering.com/. Accessed: 2019-06-06.
[2] [n.d.]. Bouncy Castle. http://www.bouncycastle.org. ([n. d.]).
[3] [n.d.]. Crimes 2001 to present (City of Chicago). https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2. ([n. d.]).
[4] [n.d.]. TPC-H benchmark. http://www.tpc.org/tpch. ([n. d.]).
[5] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order Preserving Encryption for Numeric Data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 563–574.

[6] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 205–216.

[7] Raphael Bost. 2016. Sofos: Forward Secure Searchable Encryption. In *CCS*.

[8] Raphael Bost and Pierre-Alain Fouque. [n.d.]. *Thwarting Leakage Abuse Attacks against Searchable Encryption–A Formal Approach and Applications to Database Padding*. Technical Report. Cryptology ePrint Archive, Report 2017/1060.

[9] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *CCS*.

[11] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. [n.d.]. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011* ([n. d.]).

[12] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*.

[13] Jonathan L Dautrich Jr and Chinya V Ravishankar. 2013. Compromising Privacy in Precise Query Protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 155–166.

[14] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD*.

[15] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *TODS* (2018).

[16] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast Searchable Encryption With Tunable Locality. In *SIGMOD*.

[17] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. 2018. Efficient searchable encryption through compression. *PVLDB* (2018).

[18] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*.

[19] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *CCS*.

[20] Oded Goldreich and Rafail Ostrovsky. [n.d.]. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996* ([n. d.]).

[21] Louis Granboulan and Thomas Pornin. 2007. Perfect block ciphers with small blocks. In *International Workshop on FSE*.

[22] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenny Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range series. In *CCS*.

[23] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Why Your Encrypted Database Is Not Secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*. 162–168.

[24] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2014. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 235–246.

[25] Seny Kamara and Tarik Moataz. 2019. Encrypted Multi-Maps with Computationally-Secure Leakage. (2019).

[26] Seny Kamara and Tarik Moataz. 2019. SQL on Structurally-Encrypted Databases. *ASIACRYPT* (2019).

[27] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. 2018. Structured encryption and leakage suppression. In *CRYPTO*.

[28] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic attacks on secure outsourced databases. In *CCS*.

[29] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution. IEEE SSP 2020.

[30] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*.

[31] Evangelia Anna Markatou and Roberto Tamassia. [n.d.]. Mitigation Techniques for Attacks on 1-Dimensional Databases that Support Range Queries. ISC 2019.

[32] Evangelia Anna Markatou and Roberto Tamassia. 2019. Full Database Reconstruction with Access and Search Pattern Leakage. ISC 2019.

[33] Ben Morris and Phillip Rogaway. 2014. Sometimes-Recurse Shuffle - Almost-Random Permutations in Logarithmic Expected Time. In *EUROCRYPT*.

[34] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*.

[35] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*.

[36] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*.

[37] Emil Stefanov and Elaine Shi. 2012. FastPRP: Fast Pseudo-Random Permutations for Small Domains. *IACR* (2012).

[38] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*.

[39] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *CCS*.

[40] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *PVLDB* 6, 5 (2013), 289–300.

[41] Sameer Wagh, Paul Cuff, and Prateek Mittal. 2018. Differentially Private Oblivious RAM. *Proceedings on Privacy Enhancing Technologies* (2018).

[42] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *CCS*.

[43] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. [n.d.]. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX 2016*.

Real(λ)

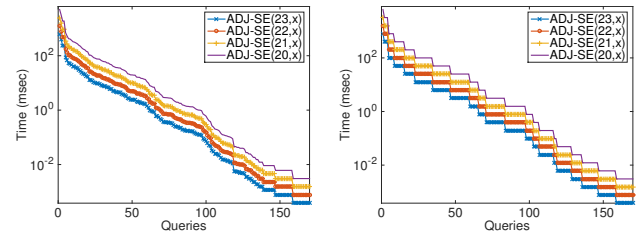
1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda)$
2: $(st_C, \mathcal{I}_0) \leftarrow \mathsf{Setup}(1^\lambda, \mathcal{D})$
3: **for** $1 \le i \le q$ **do**
4: $\quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, \mathcal{I}_{i-1}, M_1, \ldots, M_{i-1})$*
5: $\quad (\mathcal{X}_i, st_C, \mathcal{I}_i) \leftrightarrow \mathsf{Search}(st_C, w_i, \mathcal{I}_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $\mathcal{I} = \mathcal{I}_0 \ldots \mathcal{I}_q$, $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: **return** $v = (\mathcal{I}, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$

Ideal $_{\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}}}(\lambda)$

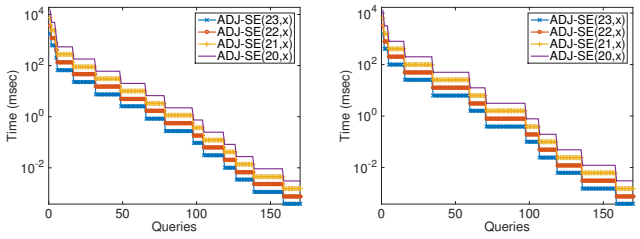
1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda)$
2: $(st_{\mathcal{S}}, \mathcal{I}_0) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$
3: **for** $1 \le i \le q$ **do**
4: $\quad (w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, \mathcal{I}_{i-1}, M_1, \ldots, M_{i-1})$*
5: $\quad (\mathcal{X}_i, st_{\mathcal{S}}, \mathcal{I}_i) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w_i), \mathcal{I}_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $\mathcal{I} = \mathcal{I}_0 \ldots \mathcal{I}_q$ and $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: **return** $v = (\mathcal{I}, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$

* Let M_k be the messages from client to server in the Search/SimSearch protocols.

Figure 14: SE/OSE real-ideal security experiments.

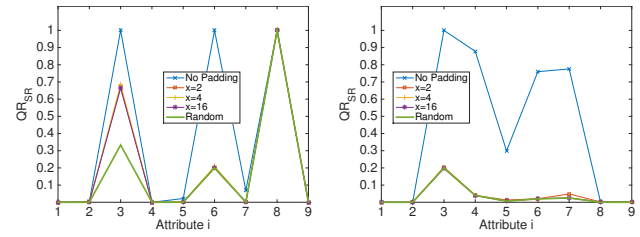


(a) $x = \perp$ (b) $x = 2$



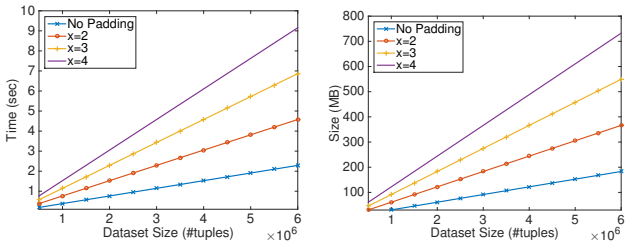
(c) $x = 3$ (d) $x = 4$

Figure 18: Search costs - Crime Dataset (Attribute 8)



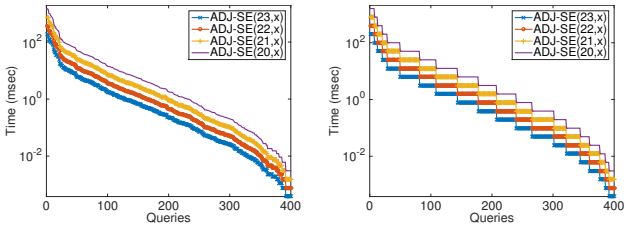
(a) ORDERS (TPC-H) (b) PART (TPC-H)

Figure 15: Query Recovery Attack against POINT-ADJ-SE for various x.

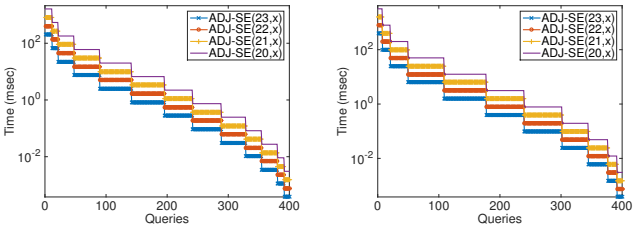


(a) Construction Time (b) Index Size

Figure 16: Index Costs - Crime Dataset



(a) $x = \perp$ (b) $x = 2$



(c) $x = 3$ (d) $x = 4$

Figure 17: Search costs - Crime Dataset (Attribute 5)