SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage

Ioannis Demertzis, *University of Maryland;* Dimitrios Papadopoulos, *Hong Kong University of Science and Technology;* Charalampos Papamanthou, *University of Maryland;* Saurabh Shintre, *NortonLifeLock Research Group*

https://www.usenix.org/conference/usenixsecurity20/presentation/demertzis

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage

Ioannis Demertzis[*]
University of Maryland

Dimitrios Papadopoulos
Hong Kong University of Science & Technology

Charalampos Papamanthou
University of Maryland

Saurabh Shintre
NortonLifeLock Research Group

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*, *group-by* 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). Thus, SEAL could comprise a promising approach to build 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 on the underlying encrypted data. Initially implemented with weak primitives like order-preserving (OPE) and deterministic (DET) encryption (e.g., [5, 6, 43, 48])[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 [17, 18], range queries [15, 16, 20], and SQL queries [30] (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. [28] and Cash et al. [11] were the first to exploit access pattern leakage and prior knowledge about the dataset to recover the queried keywords. Zhang et al. [51] propose file injection attacks for encrypted email applications to improve the recovery rate of queried keywords. Blackstone et al. [7] revisit various assumptions of existing leakage-abuse attacks. For private range search, effective access pattern and volumetric attacks through which the attacker learns the plaintext order and value of encrypted records, without any prior knowledge, have been proposed [13, 24, 25, 27, 32, 34–36, 39]. This growing body of leakage-abuse attacks has already alerted the community about using SE for implementing encrypted databases [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) [46] (introducing a polylogarithmic search overhead) and (ii) perform worst-case padding

[*]Work partially done as a research intern at Symantec Research Labs (now NortonLifeLock Research Group) and HKUST.

[1]Note that such implementations have been shown to be susceptible to inference attacks [41] since they leak statistical and order information allowing an attacker to decrypt the actual encrypted records.

(resulting in worst-case linear search time [29] 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. To better motivate SEAL, we first present new attacks on existing SE-based encrypted databases. In particular, we show that the same inference attacks on DET systems [41] can be used by a persistent adversary to recover the database values in SE-based systems, such as those implementing point queries (e.g., [15, 17]), and group-by and join queries (e.g., [30]). 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 [15, 20], 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 construction 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 constructions [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 plaintext values (for the queried attribute) to the tuples of the encrypted database. Note that since SEAL hides some bits of access pattern via ADJ-ORAM, database recovery can be quite challenging (see Section 5).

5. We observe that for all 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 powerful attackers only at the expense of an acceptable slowdown from plain SE—around $32\times$.[3]

Prior work. Wagh et al. [49] 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 [46]. 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. [11], and Bost and Fouque [9] 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. Similar padding approaches have been also proposed in other areas, e.g., [37] proposes padding approaches for preventing snapshot attacks on deterministically encrypted data and [38] proposes padding for traffic analysis attacks. Bost and Fouque [9] also propose new security definitions for SE aiming at capturing existing leakage abuse attacks.

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

[3]In Section 5, we report for certain parameters of α and x the performance of SEAL compared with the most secure solution (sequential scan) and the one that leaks access and search patterns (SE scheme). We highlight that both sequential scan and SE are **not** competitors of SEAL since they provide different security, but we used those two schemes only as reference points.

These theoretical definitions could potentially provide some intuition on how we can modify existing schemes in order to make them robust against such attacks.

Recently, Kamara et al. [31] 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 [29] 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.

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 $\nu : \mathbb{N} \rightarrow \mathbb{R}$ 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 in [22], 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 [50] for a detailed description):

- $(T, \sigma) \leftarrow \textsc{ODictSetup}(1^\lambda, N)$: 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.

- $((\texttt{value}, \sigma'), T') \leftrightarrow \textsc{ODictSearch}((\texttt{key}, \sigma), T)$: Given the search key `key` and σ, returns the corresponding value `value`, the updated T' and σ'.

- $(\sigma', T') \leftrightarrow \textsc{ODictInsert}((\texttt{key}, \texttt{value}, \sigma), T)$: 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* 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]:

- $(st_C, I) \leftarrow \textsc{Setup}(1^\lambda, \mathcal{D})$: 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 I which is sent to the server. st_C is sent to the client and it contains the secret key k.

- $((\mathcal{X}, st_C), I) \leftrightarrow \textsc{Search}((st_C, w), I)$: 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 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 index I.

The security of the above SE scheme is captured by the following definition, using the standard SE's real/ideal security game [12] (see Figure 1).

Definition 1 *Suppose* $(\textsc{KeyGen}, \textsc{Setup}, \textsc{Search})$ *is a SE scheme based on the above definition, let* $\lambda \in \mathbb{N}$ *be the security parameter and consider experiments* $\boldsymbol{Real}(\lambda)$ *and* $\boldsymbol{Ideal}_{\mathcal{L}_1,\mathcal{L}_2}(\lambda)$ *presented in Figure 1, 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*:*

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

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 $\mathcal{L}_1, \mathcal{L}_2$ [12]. 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

```
**Real**$(\lambda)$
1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^{\lambda})$
2: $(st_{C}, I_0) \leftarrow \mathsf{Setup}(1^{\lambda}, \mathcal{D})$
3: for $1 \leq i \leq q$ do
4:     $(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, I_{i-1}, M_1, \ldots, M_{i-1})^{*}$
5:     $(\mathcal{X}_i, st_{C}, I_i) \leftrightarrow \mathsf{Search}(st_{C}, w_i, I_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $I = I_0 \ldots I_q$, $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: return $v = (I, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$
**Ideal**$_{\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{QUERY}}}(\lambda)$
1: $(\mathcal{D}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^{\lambda})$
2: $(st_{\mathcal{S}}, I_0) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$
3: for $1 \leq i \leq q$ do
4:     $(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(st_{\mathcal{A}}, I_{i-1}, M_1, \ldots, M_{i-1})^{*}$
5:     $(\mathcal{X}_i, st_{\mathcal{S}}, I_i) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}},$
                                   $\mathcal{L}_{\mathsf{QUERY}}(\mathcal{D}, w_i), I_{i-1})$
6: Let $\mathbf{M} = M_1 \ldots M_q$, $I = I_0 \ldots I_q$ and $\mathcal{X} = \mathcal{X}_0 \ldots \mathcal{X}_q$
7: return $v = (I, \mathbf{M}, \mathcal{X})$, $st_{\mathcal{A}}$
* Let $M_k$ be the messages from client to server in the Search/SimSearch protocols.
```

Figure 1: SE/OSE real-ideal security experiments.

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*.

For certain database query types, such as point queries, $\mathcal{L}_2$ leakage contains only the *search* and *volume pattern leakage*. The reason is that there is a structural difference between the keyword search problem and database point queries. In keyword search, one document identifier can be included in multiple keywords, while in database search one tuple-id or an encrypted tuple can have exactly one searchable value per attribute. For example, a patient cannot have more than one date of birth. Using this observation, we can store in the encrypted index directly the encrypted tuples instead of the tuple-ids without increasing asymptotically the storage.

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. [41] proposed the *frequency analysis* and $\ell_{\mathbf{p}}$*-optimization* attacks that apply to databases encrypted with the use of deterministic schemes such as CryptDB [43].

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 new) on these constructions. Our findings systematically re-establish that using SE to implement encrypted databases [15, 20, 30] 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 [26, 41].

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. [15], and Kamara and Moataz [30]) 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 [30]), where a 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

[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.

therefore compute the size of each group. By running a frequency analysis attack or an $\ell_{\mathbf{p}}$-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.

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 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 [30] 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, JOIN-SE-PRECOMPUTE. It is easy to see that JOIN-SE and JOIN-SE-PRECOMPUTE leak the *encrypted join graph*. That is, for each encrypted tuple t of $\mathcal{T}$, the respective encrypted tuples t′ of $\mathcal{R}$ that have the same value on x with 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 [43]). 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, 25, 32, 35, 36, 39]. 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. [20] and Demertzis et al. [15, 16] 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 [15], builds a binary-tree data structure with some extra "internal" nodes (see Figure 2) 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.

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 2 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 [15], 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 $[a',b']$

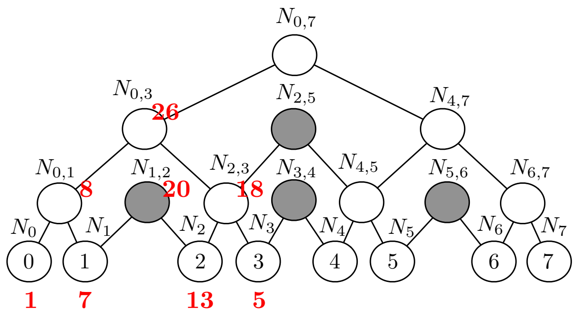


Figure 2: LOGARITHMIC-SRC [15, 16] consists of a full binary tree over the domain with an extra internal node between every two cousins. Red values denote the number of tuples each node contains (used for the proposed attack).

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 to querying T_1) and r is the size of the result (and is due to querying T_2).

Do existing attacks apply? It seems that existing (volumetric) attacks on RANGE-SE [13, 24, 25, 27, 32, 34–36, 39] do not apply to the above, response-hiding, schemes. However we must note that LOGARITHMIC-SRC and LOGARITHMIC-SRC-I leak the volume pattern of a restricted set of queries and may be vulnerable to new volumetric attacks. In particular, the very recent and concurrent work of Gui et al. [27] proposed new volumetric attacks that can handle cases of missing/spurious queries, and cases that return noisy results. These attacks for missing and noisy queries could potentially be used against LOGARITHMIC-SRC by setting a small window size and treating all volumes from large windows as noise. However, it is not clear how this noise would affect the attack output since the missing queries are not chosen at random as is assumed in [27]. Below, we describe our new attacks tailored to 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 LOGARITHMIC-SRC tree with $\mathsf{Dom} = \{0,1,2,3\}$ (and therefore 8 nodes, including the one "extra" internal node—see Figure 2). 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 2, 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 since for each range query one node of T_1 and one for T_2 will be accessed together.

The above attacks are not analyzed in full detail since we want to use them mainly as a way to manifest the weaknesses of the Logarithmic-SRC and Logarithmic-SRC-i schemes [15]. We also use them as a motivation to introduce our new RANGE-SRC-SE-(a,x) scheme (see Section 4.5). Exploring these attacks against Logarithmic-SRC and Logarithmic-SRC-i 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 section 3 exploit the leakage of SE such as the *search*, *overlapping* and *volume pattern*. In this section we propose SEAL, a family of new SE schemes with adjustable 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.

```
$bit \leftarrow \mathbf{Real}^{\text{ADJ-ORAM-}\alpha}(\lambda)$:
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        ▷ q: polynomial #queries
4:     $i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5:     $((v_{i_k}, \sigma_k), \mathsf{EM}_k) \leftrightarrow$ ADJ-ORAMACCESS$((\mathtt{op}, i_k, v_{i_k}, \sigma_{k-1}), \mathsf{EM}_{k-1})$.
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:     $i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5:     $(st_\mathcal{S}, \mathsf{EM}_k) \leftrightarrow$ 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 3: ADJ-ORAM-α real-ideal security experiments. With $m_0, m_1, \ldots$, we denote the messages exchanged at Line 5 of both experiments.

4.1 Adjustable Oblivious RAM

An adjustable ORAM (ADJ-ORAM-α) is parameterized by a parameter α that defines the number of leaked bits of the accessed memory location ($\alpha = 0$ for a traditional ORAM). We 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$((\mathtt{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$ 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 $\mathtt{op} = \mathtt{read}$ 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 $\mathtt{op} = \mathtt{read}$ (for $\mathtt{op} = \mathtt{write}$ 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 3 parametrized by leakage functions $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$.

Definition 2 *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):*

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

is at most neg(λ), where the above experiments are defined in Figure 3 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.

```
$(\sigma, \mathsf{EM}) \leftrightarrow$ 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 $\pi_k$ 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:     Let $\ell$ be the integer representation of the $\alpha$ most significant bits of $\pi_k[i]$ and $\phi$ be the integer representation of the remaining bits of $\pi_k[i]$.
7:     $S_{\ell+1}[\phi + 1] = v_i$.
8: for $i = 1, \ldots, \mu$ do
9:     $(\sigma_i, \mathsf{EM}_i) \leftrightarrow$ ORAMINITIALIZE$((1^\lambda, S_i), \perp)$.
10: Let EM to be $\mathsf{EM}_1, \ldots, \mathsf{EM}_\mu$ and $\sigma$ to $(\sigma_1, \ldots, \sigma_\mu)$.
11: return $(\sigma, \mathsf{EM})$.
$((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-ORAMACCESS$((\mathtt{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$
1: Parse $\sigma$ as $(\sigma_1, \ldots, \sigma_\mu)$ and EM as $(\mathsf{EM}_1, \ldots, \mathsf{EM}_\mu)$ where $\mu = 2^\alpha$.
2: Let $\ell$ be the integer representation of the $\alpha$ most significant bits of $\pi_k[i]$ and $\phi$ 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$((\mathtt{op}, \phi, v_i, \sigma_\ell), \mathsf{EM}_\ell)$.
5: return $(v_i, \sigma, \mathsf{EM})$.
```

Figure 4: ADJ-ORAM-α using any ORAM as a black box.

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.

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 [23, 40, 45]), 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 4.

Theorem 1 *Assuming* $(\textsc{OramInitialize}, \textsc{OramAccess})$ *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 Def. 2.*

The ORAM scheme used is secure and therefore we use its algorithms $\textsc{SimOramInitialize}$ and $\textsc{SimOramAccess}$. In particular, the ADJ-$\textsc{SimOramInitialize}$ takes as an input $\mathcal{L}_1^\alpha = (n, \alpha)$ and the security parameter λ, and it creates $\text{EM}_1, \ldots \text{EM}_\mu$ and $\sigma_1, \ldots, \sigma_\mu$ using $\textsc{SimOramInitialize}(1^\lambda, \frac{n}{\mu})$ for $\mu = \frac{n}{2^\alpha}$. The ADJ-$\textsc{SimOramAccess}$ 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 $\textsc{SimOramAccess}(\sigma_i, \text{EM}_i)$. Then, the simulator properly updates EM_i and σ_i.$\square$

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 [42]), 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 [46] and oblivious data structures [50] 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—leaking $\log\log N + 1$ bits, at most doubling the search and storage overhead.

```
$\mathcal{D} \leftarrow$ 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 5: ADJ-Padding-x leading to $\log_x N$ different sizes.

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 5). 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. [11] and Bost and Fouque [9], as well as Lacharite et al. [37] and Liberatore et al. [38]). 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 $\text{SEAL}(\alpha, x)$, our adjustable SE construction that uses ADJ-ORAM-α, ADJ-PADDING-x and an oblivious dictionary 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). $\text{SEAL}(\alpha, x)$ is defined similarly with SE (see Section 2) and has algorithms/protocols Setup and Search. Our construction is described in Figure 6.

```
$(st_C, I) \leftarrow \textsc{Setup}(1^\lambda, \mathcal{D})$
 1: Let $\mathcal{D}$ be the input dataset and let $\mathbf{W}$ be 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: $(T, \sigma_{\text{odict}}) \leftarrow \textsc{ODictSetup}(1^\lambda, N)$.
 5: for all $w \in \mathbf{W}$ do
 6:     Let $cnt_w = |\mathcal{D}(w)|$.
 7:     $(\sigma_{\text{odict}}, T) \leftrightarrow \textsc{ODictInsert}((w, i_w||cnt_w, \sigma_{\text{odict}}), T)$.
 8: $(\sigma_{\text{oram}}, \text{EM}) \leftarrow$ ADJ-$\textsc{OramInitialize}(1^\lambda, \text{M}, \alpha)$.          ▷ Parameter $\alpha$ is public.
 9: $st_C = (\sigma_{\text{oram}}, \sigma_{\text{odict}})$ and $I = (\text{EM}, T)$.
10: return $(st_C, I)$.
$((\mathcal{X}, st_C), I) \leftrightarrow \textsc{Search}((st_C, w), I)$
 1: Parse $I$ as (EM, $T$) and $st_C$ as ($\sigma_{\text{odict}}$, $\sigma_{\text{oram}}$) and let $\mathcal{X}$ be empty.
 2: $((\texttt{value}, \sigma_{\text{odict}}), T) \leftrightarrow \textsc{ODictSearch}((w, \sigma_{\text{odict}}), T)$.
 3: Parse value as $(i_w||cnt_w)$.
 4: for $i = i_w, \ldots, i_w + cnt_w$ do
 5:     $((v_i, \sigma_{\text{oram}}), \text{EM}) \leftrightarrow$ ADJ-$\textsc{OramAccess}((read, i, \perp, \sigma_{\text{oram}}, \alpha), \text{EM})$.     ▷ Parameter $\alpha$ is public.
 6:     $\mathcal{X} \leftarrow \mathcal{X} \cup v_i$.
 7: return $(\mathcal{X}, st_C, I)$.
```

Figure 6: Our SEAL(α,x) scheme using ADJ-ORAM-α, ADJ-PADDING-x, and an oblivious dictionary as black boxes.

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 I, which contains the small oblivious memories $\text{EM}_1, \ldots$ 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 simply denote these leakages as $\mathcal{L}_1$ and $\mathcal{L}_2$.

Theorem 2 *Assuming that* ODICT *is a secure oblivious data structure according to [50] (Def. 1) and* ADJ-ORAM-α *is secure according to Def. 2, then* SEAL(α,x) *is $(\mathcal{L}_1, \mathcal{L}_2)$-secure according to Def. 1.*

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 $(T, \sigma_{\text{odict}}) \leftarrow \textsc{ODictSetup}(1^\lambda, N)$ 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-$\textsc{SimOramInitialize}(1^\lambda, N', \alpha)$ 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 $|\mathcal{D}_\alpha^x(w)|$ times the ADJ-SIMORAMACCESS with input the α-bit identifiers in $\mathcal{D}_\alpha^x(w)$ ($\mathcal{D}_\alpha^x(w)$ 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 is

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

and the client space is just $c(N)$. Whether one chooses to store the local states locally or outsource them depends on the parameter α. 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 [46]. 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 [50] (this requires $b \log^2 N$ additional additive cost where b is the bitsize 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 [8, 10, 21, 47]. 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, and 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 of LOGARITHMIC-SRC-I [15] so that the potential attack presented in Section 3.3 is mitigated. Recall the attack works by exploiting volumes exposed by tree T_1 (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 is $O(x \cdot r)$ and the space is $O(N \log_x N)$.

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 [11, 13, 25, 28, 32, 36]. However, these attacks are very sensitive to the *exact* overlapping or volume pattern (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 *quite powerful*, having plaintext access to the input dataset. We stress that this is a "heavy" benchmark that already covers known attacks [11, 13, 25, 28, 32, 36]. 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 model considers a single-client setting (we do not support a multi-client scenario with multiple parties accessing the data). We assume that our adversary: (i) is the system provider that hosts the encrypted database (including the encrypted index) and performs the encrypted query execution; (ii) is honest-but-curious (i.e., tries to infer information during the execution of the protocol, but does not deviate from the protocol, e.g., to give a "tampered" answer); (iii) has full visibility of the server-side execution and memory; (iv) acquires all the possible leaked information from query execution—observing all possible queries at least once; (v) has access to 100% of the plaintext database. 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*, that requires to map plaintext values (for the queried attribute) to the tuples of the encrypted database.

We stress that this a strong attacker model, one that we believe is beyond most real-world adversaries' capabilities. This was a deliberate design decision as our main goal is to evaluate our proposed mitigation techniques against a strong adversary. On the other hand, our analysis does not capture cases where the attacker has information about the query distribution.

Note here that a database recovery attack in the case of SE ($\alpha = \log N$) is trivial, since the identifiers of the encrypted records reveal the desired mapping to the plaintext records directly. This task becomes more challenging for smaller values of α where this information is not given in its entirety.

```
$QR_{SR} \leftarrow$ QueryRecoveryAttack($\mathcal{T}, \{t_q, |q|\}_{q \in Q}$)
Input: Plaintext tuples $\mathcal{T}$ and tokens $t_q$ 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 decryption for $t_q$ then
7:         CORRECT++.
8: return CORRECT$/|Q|$.
```

Figure 7: Query Recovery Attack for Point Queries.

In addition, note that the database recovery attack becomes also trivial if SEAL does not re-randomize or assign new tuple ids to encrypted tuples; which is not the case in SEAL (see Line 6 of the used ADJ-ORAM-α).

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 encrypted tuples that have been correctly mapped to the plaintext 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-SRC-SE-(α,x), we used the TPC-H benchmark [4] with scaling factor 0.1 which is widely used by the database community[5]. TPC-H consists of eight separate tables (PART, SUPPLIER, PARTSUPP, CUSTOMER, NATION, LINEITEM, REGION, ORDERS). Our attacks take as input the leakage of all possible queries (worst-case leakage). The same attacks can be run with less queries, leading to lower success rate. When evaluating the performance of

[5]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).

```
$DR_{SR} \leftarrow$ DatabaseRecoveryAttack($\mathcal{T}, enc(\mathcal{T}), \{t_q, S_q\}_{q \in Q}$)
Input: Plaintext tuples $\mathcal{T}$, encrypted tuples $enc(\mathcal{T})$ and tokens $t_q$ along with respective set $S_q$ of encrypted tuples (and their $\alpha$-bit identifiers).
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 $(t_q, S_q)$ do
4:     Choose $q'$ at random from the set $\{q' : |\mathcal{T}(q')| = |S_q|\}$.
5:     for each encrypted tuple $e \in S_q$ do
6:         Let $id$ be the $\alpha$-bit identifier of $e$.
7:         Choose at random a tuple $t$ from $enc(\mathcal{T})$ that has $id$ as the first $\alpha$ bits of its identifier.
8:         Remove $t$ from $enc(\mathcal{T})$.
9:         if encrypted tuple $t$ has value $q'$ at the queried attribute then
10:            CORRECT++.
11:    Remove $q'$ from $\mathcal{T}$.
12: return CORRECT$/\sum |S_q|$.
```

Figure 8: Database Recovery Attack for Point Queries.

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. In both attacks we consider one attribute of one table at a time.

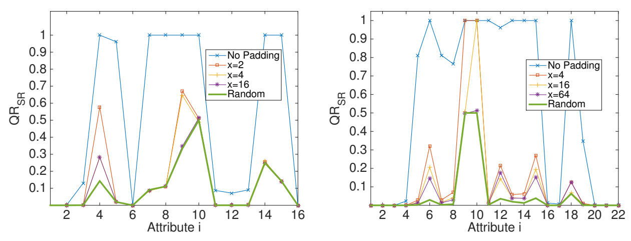
Our *query recovery* attack (see Figure 7) is very simple and uses only volume pattern leakage. Having access to the plaintext table $\mathcal{T}$, the adversary computes the new padded table for the queried attribute (Line 1 in Figure 7) using the padding parameter x. Now, for a given encrypted query 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 7). 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 8) works as follows. First the adversary decrypts which keyword we are querying, as before—say this keyword is q'. Now, the goal is to map the value q' to the correct encrypted tuples in $enc(\mathcal{T})$, where $enc(\mathcal{T})$ is the encrypted database produced by the SETUP algorithm of SEAL. The adversary knowing from $\mathcal{L}_2$ leakage the α-bits of each returned encrypted tuple, chooses at random for each of them one tuple from $enc(\mathcal{T})$ with same α bits as prefix and maps q' to this tuple. Finally, the adversary removes the chosen tuples t from $enc(\mathcal{T})$. The adversary is successful if after this process the encrypted tuple t has value q' at the queried attribute. Clearly, the smaller α is, the more bits the adversary will have to guess (the larger the set of tuples with same α bits as prefix is) and therefore the less successful the attack is going to be.

Query recovery attack evaluation. Figures 9(a), and 9(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 9(a) demonstrates the evaluation for the LINEITEM table (TPC-H), while Figure 9(b) presents the results for the Crime dataset. In all figures, we 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.

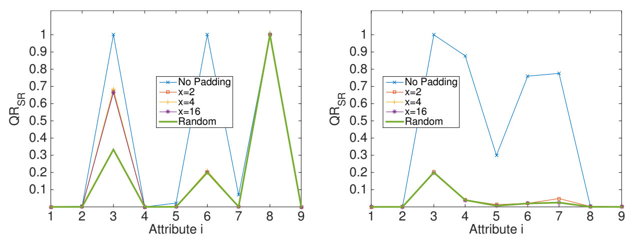
In Figure 9(a), for $x = 2$ (only a $2\times$ overhead in search time and storage), we see that our scheme forces the attacker to perform very close to "Random" for 14 out of 16 attributes. We observe that QR_{SR} for attribute 8 is close to Random for $x = 16$, while for attribute 4 greater values of 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 them. We observe similar patterns for the tables of TPC-H and we report the results for tables ORDERS and PART in Figure 10.

In Figure 9(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 $4\times$ 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 $x = 64$.



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

Figure 9: Query Recovery Attack against POINT-ADJ-SE for various x.



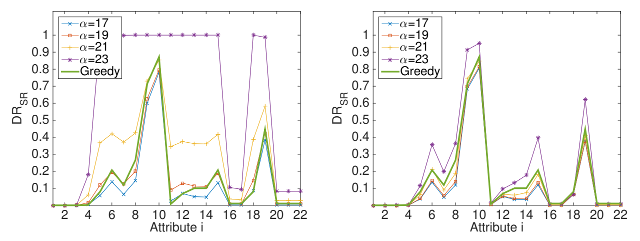
(a) ORDERS (TPC-H) (b) PART (TPC-H)

Figure 10: Query Recovery Attack against POINT-ADJ-SE for various x.

Database recovery attack evaluation. 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 11 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 tuples 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 11 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 12, we provide a more detailed evaluation for 4 specific attributes of the crime dataset for $\alpha \in [0, \log N]$ and $x = \perp, 2, 3, 4$.

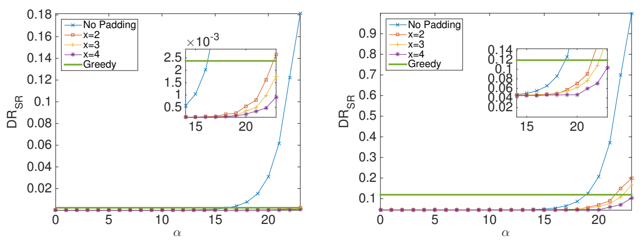
5.4 Attacking JOIN-ADJ-SE

We evaluate the effectiveness of JOIN-ADJ-SE-(α, x) using the database recovery attack proposed for point queries (see Figures 8). Since the database schema and the size of each table are usually not considered private information, we do not consider join query recovery attacks.

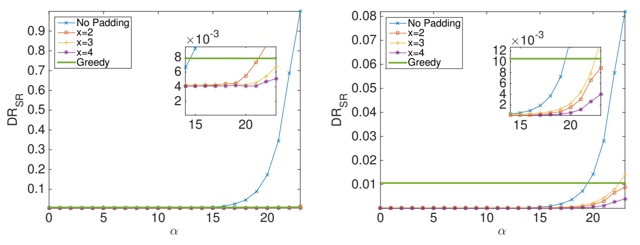


(a) $x = \perp$ (b) $x = 2$

Figure 11: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. We show all attributes.



(a) Attribute 4 (b) Attribute 7



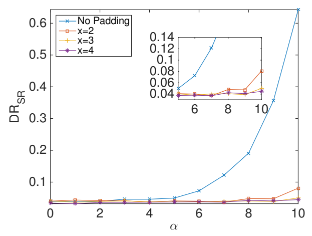
(c) Attribute 11 (d) Attribute 20

Figure 12: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. Attributes 4,7,11,20.

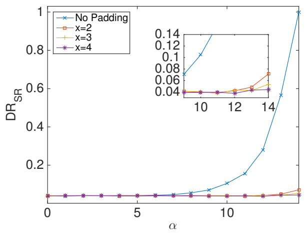
Attack evaluation. Figure 13 demonstrates the database recovery attack for foreign-key join queries. We consider foreign-key joins between tables (i) SUPPLIER and NATION—Figure 13(a), and (ii) CUSTOMER and NATION; the TPC-H benchmark contains only foreign-key joins. We observe in Figure 13(b) the DR_{SR} for $\alpha = [0, \log N]$, and $x = \perp, 2, 3, 4$. For $\alpha = 0$ and $x = \perp$, DR_{SR} is 65% in Figure 13(a) and 97% in Figure13(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 7 and 8). In particular in Line 2 of both Figure 7 and 8, 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.



(a) SUPPLIER ⋈NATION



(b) CUSTOMER ⋈NATION

Figure 13: Database Recovery Attack for Foreign-key Join Queries for the TPC-H Benchmark.

Attribute	#Queries	# Correctly Decrypted Queries			
		Baseline	RANGE-SRC-SE		
			$x=2$	$x=4$	$x=8$
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

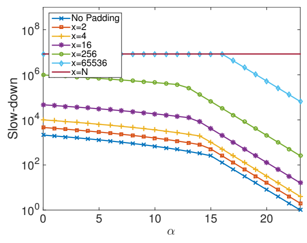
Table 1: Query Recovery Attack for Range Queries (QR_{SR} = # Correctly Decrypted Queries /#Queries)

Attack evaluation. We focus on numeric attributes PS_SupplyCost from table PARTSUPP; P_Size and P_RetailPrice from PART; L_TAX, L_QUANTITY, L_DISCOUNT from LINEITEM. Table 1 presents for each attribute the number of all possible 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 $x = 8$ (Columns 4, 5, 6 of Table 1). We observe that $x = 8$ 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.

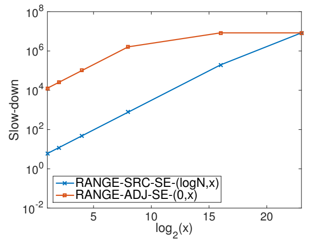
5.6 Efficiency of Adjustable Constructions

In Figure 14(a), we fix a database with size 2^{22} records, and we show the largest slowdown (across all the possible result sizes—$1,2,3\ldots N$) of SEAL(α,x) compared to a SE scheme which has the maximum leakage. Similarly, in Figure 15(a), we show the smallest speedup achieved by our construction SEAL(α,x) (for various values of α and x) compared to an approach that performs sequential scan and has no leakage. Because, we consider the worst-case speedup from the most secure solution ($\alpha = 0$ and $x = N$), sequential scan provides a more efficient approach than the use of worst-case padding with ORAM which is also achieves the same security. We do an analysis of these plots in the next section.

We highlight again that **neither** SE **nor** sequential scan are competitors of SEAL, since (i) SEAL encapsulates those schemes (e.g., for $\alpha = 0$ and $x = N$ becomes sequential scan and for $\alpha = \log N$ and $x = \perp$ becomes SE scheme), and (ii)

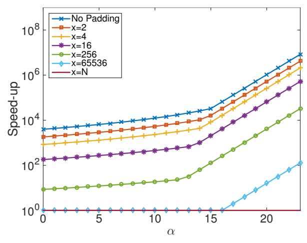


(a) SEAL(α,x)

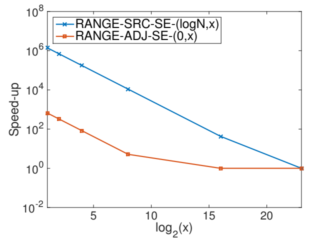


(b) Range Schemes

Figure 14: Slowdown from SE.



(a) SEAL(α,x)



(b) Range Schemes

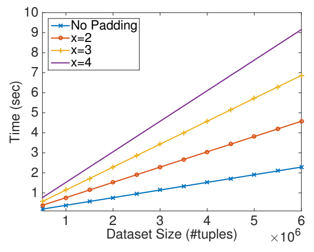
Figure 15: Speedup from sequential scan.

for non-trivial α and x they provide different security level. We provide those experiments only as reference points of SEAL's performance compared with the most and least secure solutions. In addition, Figures 14(a),15(a) can be used in combination with Figures 9-13 and Table 1: For a given query type and attack we can specify good values for α, x (for mitigating the attack) from Figures 9-13 and Table 1,and for those values we can see the relative performance of SEAL compared with SE and sequential scan in Figures 14(a),15(a).

Figure 14(b) and 15(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.[6]

We provide additional experiments regarding the performance of our SEAL scheme for the crime dataset. 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 $x = \perp, 2, 3, 4$. Finally, in Figures 17 and 18 we evaluate the end-to-end search time of our SEAL scheme for two attributes of the crime dataset for $\alpha = 20, 21, 22, 23$ and $x = \perp, 2, 3, 4$.

[6] Although the search pattern (combined with the access pattern) has been used in recent work by Kornaropoulos et al. [35] to attack RANGE-SE, it is not clear how it can be used for RANGE-SRC-SE-(α,x).


(a) Construction Time

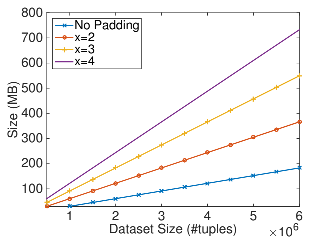
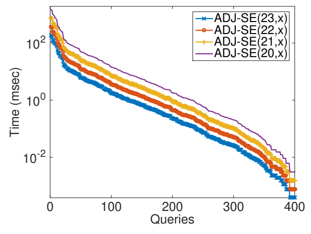
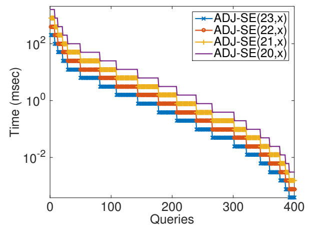
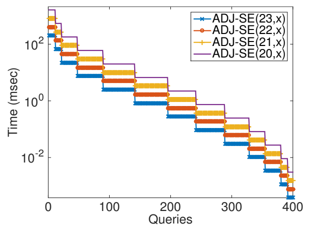

(b) Index Size

Figure 16: Index Costs - Crime Dataset


(a) $x = \perp$


(b) $x = 2$


(c) $x = 3$

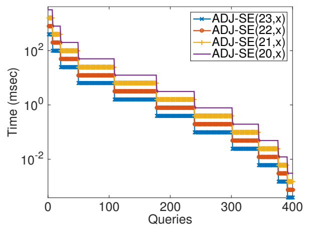
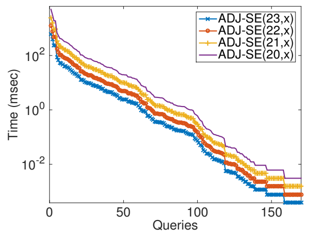
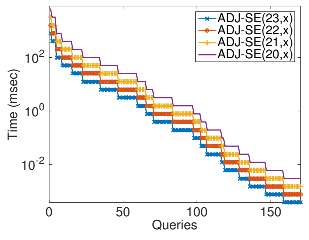
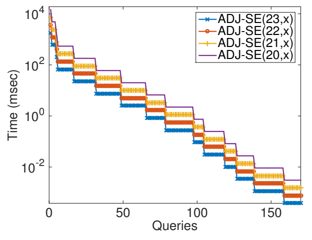

(d) $x = 4$

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


(a) $x = \perp$


(b) $x = 2$


(c) $x = 3$

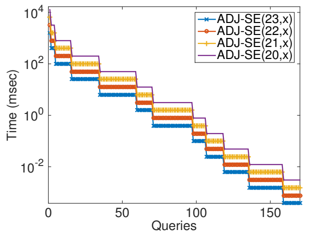

(d) $x = 4$

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

5.7 Setting Parameters α and x in Practice

From the above findings, it should be evident that finding appropriate parameter values is heavily data-dependent. In particular, it depends on the size of the database, number of distinct values, and the distribution of a given searchable attribute. One way for users to tune these parameters is to use our attacks as an estimator, e.g., provide their databases as input and try different values of α and x in order to set their desirable success rate thresholds against our attacks (before outsourcing the database). Below, we provide more general guidelines on how one can set these parameters based on our evaluation.

Setting parameter x. Parameter x solely controls the success rate of the query recovery attack for point, range (RANGE-SRC scheme) and group-by queries. The query recovery attack tries to map the encrypted queries to plaintext ones based on the volume leakage. For instance, if a database contains only two values a and b and the volume of the former value is greater than the latter, i.e., $|q(a)| > |q(b)|$, the adversary can correctly map with certainty the encrypted query with the greater volume to a and the other one to b. Now, assuming that both values have the same volume, the adversary cannot distinguish the encrypted queries and is forced to guess. Increasing the parameter x, we try to have more queries with the same size in order to increase the adversary's uncertainty, but finding a good value of x also depends on the distribution of the searchable value. For instance, attribute 9 of the crime dataset is a binary attribute (it has 2 distinct values), in which $|q(a)| = 4374175$ and $|q(b)| = 1749100$. We observe that for $x = 2$ these queries still will have different volumes, but for $x = 3$ they obtain the same volume (i.e., $|q'(a)| = |q'(b)| = 4782969$) and they will be indistinguishable. Attribute 10 of the crime dataset, which is also a binary attribute, has $|q(a)| = 5337429$ and $|q(b)| = 785846$ and in order to make these sizes indistinguishable higher values of x are needed, i.e., $x = 14$. Again, this kind of analysis can be performed locally, prior to outsourcing the dataset.

Setting parameter α. Parameter α affects the success rate of the database recovery attacks for point, range (RANGE-SRC scheme), join and group-by queries. The success of this attack firstly depends on the outcome of the query recovery attack. Thus, tuning the parameter x in order to increase the uncertainty of the adversary is very important. Nevertheless, parameter α controls how many tuples are indistinguishable from each other. For example, setting $\alpha = \log N - 1$ our scheme creates $N/2$ ORAMs of size 2—thus every tuple is indistinguishable from another one (all the tuples that are in the same ORAM are indistinguishable from each other). Therefore, even if the query recovery attack has 100% success rate and we are trying to find the correct mapping of plaintext tuples to encrypted ones, the success rate of this attack will be at most 50% for $\alpha = \log N - 1$. However, in our proposed database recovery attack, we treat the case when encrypted and plaintext tuples have the same searchable value but differ in the rest of the attributes as a success. Due to this, the distribution of an attribute will also affect the success of the database recovery attacks. For instance, for point queries attribute 9 of the crime dataset (which has 2 values— $|q(a)| = 4374175$ and $|q(b)| = 1749100$) for $x = \perp$ and $\alpha = \log N - 1 = 22$, our attack has success rate around 87%, because the success rate of the query recovery attack is 100% and the adversary has uncertainty only when the same ORAM contains both tuples with value a and b.

Finally, we provide some general conclusions from the analysis that we performed on our chosen datasets. We observe that for point and join queries setting $\alpha = \log N - 3$ and $x = 4$ significantly reduces both QR_{SR} and DR_{SR} (e.g., attributes 4,5 of LINEITEM and attributes 13,14 of crime dataset for point queries; SUPPLIER⋈NATION and CUSTOMER⋈NATION for join queries), 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 are rare cases that attributes with skewed distribution and small number of distinct values, e.g., binary attributes, require higher values of x, such as $x = 16$ or $x = 64$ (e.g., attribute 9 of LINEITEM and attributes 9,10 of the crime dataset for point queries). In the cases of range queries, we observe that our RANGE-SRC-SE-$(\log N, x)$ for $x = 8$ significantly mitigates our all-powerful query recovery attack (e.g., L_Tax and L_Discount attribute—the success rate of the attack drops from 100% below 7% and 2% respectively) and achieves a maximum $48\times$ slowdown from plain RANGE-SE.

6 Challenges for Dynamic Databases

Our work only focuses on static databases. We believe that a very interesting problem for future work is to extend this work for dynamic databases, an approach that introduces more leakage and makes the problem more challenging. Towards this goal, we know from the literature of SE how we can support dynamic point queries (there is an extensive literature on dynamic schemes that achieve forward/backward privacy [10, 14, 19, 21, 33, 44]—the state-of-the-art security definitions for dynamic SE. A first challenge towards dynamic databases is to study if these security definitions for point queries are suitable for other query types (such as range, joins and group-by queries), as well as to find schemes that achieve those definitions. A second challenge is that prior ORAM and our ADJ-ORAM schemes require initializing at setup the worst-case memory size—modifying them for the dynamic case (without having to set a-priori a large upper bound) is a non-trivial problem. A third challenge is how we could efficiently use our ADJ-Padding technique, since new updates will continuously change the distribution of the searchable attribute. Predicting the required padding size (without extra costly bookkeeping) for a certain keyword without knowing future updates would be very challenging.

One approach for handling dynamic point queries would be to explore whether our ADJ-ORAM can be used as a drop-in replacement in existing dynamic ORAM-based SE schemes (e.g., ORION from [21]), obtaining a good efficiency/security trade-off. However, this would require addressing the aforementioned second and third challenges. An alternative direction that avoids these challenges is to use existing techniques that transform static SE to dynamic ones (e.g., SD_a from [14]). At a high level, this requires storing the result of N updates in a sequence of $\log N + 1$ separate indexes (with size $2^0, \ldots, 2^{\log N}$), where each update is first stored in the smallest index and whenever two indexes of the same size exist they are downloaded and merged to a larger new index by the client. Search queries are executed at all encrypted indexes independently. Such techniques that periodically rebuild the encrypted indexes do not require defining a maximum capacity during setup. Moreover, they allow the client to update the parameters α and x depending on how the database has evolved. However, the main drawback of this approach is updates, since it has a (amortized) $O(\log N)$ update cost. While de-amortization is possible, it is not trivial, especially in our adjustable setting, and we believe that it is a very interesting problem for future work.

7 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). In our evaluation we show that for our tested datasets SEAL is robust against all-powerful attacks with a reasonable performance overhead. Finally, we believe SEAL can serve as a benchmark for measuring the effectiveness of existing and future leakage-abuse attacks.

Acknowledgements

This work was supported in part by NSF awards #1514261 and #1652259, the National Institute of Standards and Technology, Hong Kong RGC grant ECS-26208318, and by a Symantec Research Lab Graduate Fellowship. We thank NortonLifeLock Inc. for its support, Niels Provos for shepherding the paper and the anonymous reviewers for their valuable suggestions and comments.

References

[1] Attack of the week: searchable encryption and the ever-expanding leakage function. `https://blog.cryptographyengineering.com/`. Accessed: 2019-06-06.

[2] Bouncy castle. http://www.bouncycastle.org.

[3] Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/ public-safety/crimes-2001-to-present/ijzp-q8t2.

[4] Tpc-h benchmark. http://www.tpc.org/tpch.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order Preserving Encryption for Numeric Data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004.

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

[7] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Proc. of NDSS*, 2020.

[8] R. Bost. Sofos: Forward Secure Searchable Encryption. In *CCS*, 2016.

[9] R. Bost and P.-A. Fouque. Thwarting leakage abuse attacks against searchable encryption–a formal approach and applications to database padding. Technical report, Cryptology ePrint Archive, Report 2017/1060.

[10] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[11] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[12] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011.*

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

[14] I. Demertzis, J. Ghareh Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[15] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical Private Range Search Revisited. In *SIGMOD*, 2016.

[16] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou. Practical private range search in depth. *TODS*, 2018.

[17] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[18] I. Demertzis, R. Talapatra, and C. Papamanthou. Efficient searchable encryption through compression. *PVLDB*, 2018.

[19] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *PETS*, 2018.

[20] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[21] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[22] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996.*

[23] L. Granboulan and T. Pornin. Perfect block ciphers with small blocks. In *International Workshop on FSE*, 2007.

[24] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. 2019.

[25] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range series. In *CCS*, 2018.

[26] P. Grubbs, T. Ristenpart, and V. Shmatikov. 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*, pages 162–168, 2017.

[27] Z. Gui, O. Johnson, and B. Warinschi. Encrypted databases: New volume attacks against range queries. In *CCS*, 2019.

[28] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246. ACM, 2014.

[29] S. Kamara and T. Moataz. Encrypted multi-maps with computationally-secure leakage. 2019.

[30] S. Kamara and T. Moataz. Sql on structurally-encrypted databases. *ASIACRYPT*, 2019.

[31] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[32] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[33] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, 2017.

[34] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1033–1050. IEEE, 2019.

[35] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. IEEE SSP 2020, 2020.

[36] M.-S. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[37] M.-S. Lacharité and K. G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

[38] M. Liberatore and B. N. Levine. Inferring the source of encrypted http connections. In *CCS*, 2006.

[39] E. A. Markatou and R. Tamassia. Full database reconstruction with access and search pattern leakage. ISC 2019, 2019.

[40] B. Morris and P. Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *EUROCRYPT*, 2014.

[41] M. Naveed, S. Kamara, and C. V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*, 2015.

[42] S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.

[43] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[44] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[45] E. Stefanov and E. Shi. Fastprp: Fast pseudo-random permutations for small domains. *IACR*, 2012.

[46] 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.

[47] S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *CCS*, 2018.

[48] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[49] S. Wagh, P. Cuff, and P. Mittal. Differentially private oblivious ram. *Proceedings on Privacy Enhancing Technologies*, 2018.

[50] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, 2014.

[51] 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*.