Practical Private Range Search Revisited

Ioannis Demertzis ∗ University of Maryland
yannis@umd.edu

Stavros Papadopoulos Intel Labs & MIT
stavrospp@csail.mit.edu

Odysseas Papapetrou ∗ EPFL, Lausanne, Switzerland
odysseas.papapetrou@epfl.ch

Antonios Deligiannakis Technical University of Crete
adeli@softnet.tuc.gr

Minos Garofalakis Technical University of Crete
minos@softnet.tuc.gr

ABSTRACT

We consider a data owner that outsources its dataset to an untrusted server. The owner wishes to enable the server to answer range queries on a single attribute, without compromising the privacy of the data and the queries. There are several schemes on “practical” private range search (mainly in Databases venues) that attempt to strike a trade-off between efficiency and security. Nevertheless, these methods either lack provable security guarantees, or permit unacceptable privacy leakages. In this paper, we take an interdisciplinary approach, which combines the rigor of Security frameworks and proofs with efficient Data Management techniques. We construct a wide set of novel schemes with realistic security/performance trade-offs, adopting the notion of Searchable Symmetric Encryption (SSE) primarily proposed for keyword search. We reduce range search to multi-keyword search using range covering techniques with tree-like indexes. We demonstrate that, given any secure SSE scheme, the challenge boils down to (i) formulating leakages that arise from the index structure, and (ii) minimizing false positives incurred by some schemes under heavy data skew. We analytically detail the superiority of our proposals over prior work and experimentally confirm their practicality.

1. INTRODUCTION

We focus on a setting with two parties; a data owner and a server. The owner outsources its dataset to the server, and enforces the latter with answering range queries on a single attribute. The server is untrusted, and the goal is to protect the privacy of the dataset and the queries. The owner encrypts its data prior to sending them to the server. The challenge lies in enabling the server to process the owner’s queries directly on the encrypted data, while achieving performance costs close to the non-private case. The benefits of data outsourcing and the importance of privacy have been stressed in numerous earlier works (e.g., [9, 31, 34, 38]).

∗Work performed while the author was at the Technical University of Crete.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD’16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882911

Prior work. Privacy-preserving range queries can be solved with optimal security via powerful theoretical cryptographic tools, such as Oblivious Random Access Memory (ORAM) [17, 29] and Fully Homomorphic Encryption (FHE) [12, 13]. Nevertheless, despite their recent advances, both these tools are prohibitively costly for large database applications [37, 35, 36]. Motivated by this, there is a long line of work on private range search (especially in Databases venues) that attempts to strike a more desirable balance between security and practical efficiency [18, 19, 20, 31, 38]. All existing approaches either lack provable security guarantees, or permit unacceptable leakages. For instance, [18, 20, 19] employ deterministic encryption and bucketization techniques that map tuples with the same query attribute value to the same bucket. Deterministic encryption leaks the distribution of the data on the query attribute. Moreover, these schemes do not offer standard security definitions and proofs, which makes it hard to determine any other possible leakage. Another set of works utilizes Order Preserving Encryption (OPE), which has the property that the ciphertexts preserve the ordering of the plaintexts [2, 3, 23, 27, 30]. As such, traditional efficient indexes can be built directly on encrypted data, and queried in the same manner as for plaintexts. Nevertheless, OPE is also deterministic, inheriting the distribution leakage. In addition, it inevitably leaks the ordering of the data.

The work closest to ours is by Li et al. [26], which follows the notion of Searchable Symmetric Encryption (SSE) [5, 6, 8, 9, 22, 34]. SSE has been studied extensively for keyword queries. It relaxes the security of ORAM by leaking the access patterns of each query (i.e., which data it “touched”), as well as the search patterns (i.e., which queries are the same). However, nothing else is leaked (e.g., data distribution). The gain from allowing this leakage is efficiency, since SSE schemes typically build on fast inverted indexes and make use of lightweight cryptography, such as Pseudorandom Function (PRF) and hash evaluations. SSE also provides a rigorous framework for accurately defining leakage for any construction. Unfortunately, the scheme by Li et al. [26] relies on weak, obsolete SSE definitions (explained in detail in Section 2.1). Moreover, it unnecessarily introduces false positives and does not support updates. Most importantly, it does not define leakages that are introduced by the tree structure utilized in query execution. As we will show later, one of our baseline constructions is built on similar ideas to [26], but offers substantially better security and performance, accurately defining leakage, avoiding false positives, and supporting updates.
Our contributions. In this paper, we revisit practical private range search, taking an interdisciplinary approach that combines the state-of-the-art definitional framework of SSE [9], with efficient data management methods. In particular, we reduce range search to multi-keyword search using range covering techniques with tree-like indexes, i.e., converting a range into sub-ranges, each representing an index node and receiving a keyword label. Contrary to off-the-shelf multi-keyword SSE schemes [6] that incur a prohibitive linear search time in the dataset size, we design efficient techniques based on single-keyword SSE protocols. We also show that we can use the (single-keyword) SSE security games to prove the security of a (RSSE) scheme, by carefully defining the extra structural leakage stemming from the use of the tree index to convert a range into a set of keywords. This has the important benefit that any secure SSE scheme can be used as a black box to realize a RSSE scheme, which means that any future advances in the active area of SSE can be readily incorporated into a RSSE construction.

We emphasize that expressing a range with sub-ranges mapped to index nodes gives a lot of flexibility in designing RSSE schemes with variable efficiency and security guarantees. We also point out that the choice of the range covering method affects the security guarantees and could lead to false positives, especially under heavy data skew. To capture the above, we devise a wide set of solutions, which revolve around trading storage overhead and (potentially) false positives for security. Our constructions with their performance and security characteristics are summarized in Table 1 and discussed in detail throughout the paper. We also quantify the costs of Li et al. [26], which is clearly subsumed by our Logarithmic-BRC scheme.

Logarithmic-SRC-i offers the best trade-off between security and efficiency among all our solutions. It employs a novel directed-acyclic graph (DAG) structure that resembles a tree, and entails an extra round of communication between the owner and the server (“i” in its name stands for interactive). This index allows hiding even the order of the results, which means that any future advances in the active area of SSE can be readily incorporated into a RSSE construction.

We tackle updates by adopting techniques from large-scale database systems, such as Logarithmic-SRC and Logarithmic-URC. Our contribution lies in formulating the updates in our setting by effectively using multi-keyword SSE schemes that periodically get consolidated into a single dataset. This is to amortize the average update cost, and substitute numerous random disk accesses with a few linear scan operations. We formulate updates in our setting by effectively using multiple static RSSE instances that periodically get consolidated and re-encrypted.

Our contributions are summarized as follows:

- We are the first to formalize private range search in the context of state-of-the-art SSE, effectively introducing the first concrete Range Searchable Symmetric Encryption (RSSE) framework.
- We devise numerous RSSE schemes, identifying various trade-offs between efficiency and security.
- We analytically detail the superiority of our constructions over prior work and experimentally confirm their practicality.

Our paper is self-contained, in the sense that it does not require any particular knowledge on Security or Data Management. We provide both the rigorous definitions and the intuition behind all the formalism.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Security</th>
<th>Query Size</th>
<th>Search Time</th>
<th>Storage</th>
<th>False Posit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li et al. [26]</td>
<td>0</td>
<td>$O(\log R)$</td>
<td>$\Omega(\log n \log R + r)$</td>
<td>$O(n \log n \log m)$</td>
<td>$O(r)$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>6</td>
<td>$O(1)$</td>
<td>$O(r)$</td>
<td>$O(n \log n)$</td>
<td>-</td>
</tr>
<tr>
<td>Constant-BRC</td>
<td>1</td>
<td>$O(\log R)$</td>
<td>$O(R + r)$</td>
<td>$O(n)$</td>
<td>-</td>
</tr>
<tr>
<td>Constant-URC</td>
<td>2</td>
<td>$O(\log R)$</td>
<td>$O(R + r)$</td>
<td>$O(n)$</td>
<td>-</td>
</tr>
<tr>
<td>Logarithmic-BRC</td>
<td>3</td>
<td>$O(\log R)$</td>
<td>$O(\log R + r)$</td>
<td>$O(n \log m)$</td>
<td>-</td>
</tr>
<tr>
<td>Logarithmic-URC</td>
<td>4</td>
<td>$O(\log R)$</td>
<td>$O(\log R + r)$</td>
<td>$O(n \log m)$</td>
<td>-</td>
</tr>
<tr>
<td>Logarithmic-SRC</td>
<td>6</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n \log m)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Logarithmic-SRC-i</td>
<td>5</td>
<td>$O(1)$</td>
<td>$O(R + r)$</td>
<td>$O(n \log m)$</td>
<td>$O(R + r)$</td>
</tr>
</tbody>
</table>

$n$: dataset size, $r$: result size, $m$: domain size, $R$: query range size

Table 1: Summary of our RSSE schemes and analytical comparison to our closest competitor. Our schemes are named after the storage expansion factor and the range covering technique. A higher value in the Security column means better security guarantees.
from high bandwidth overhead and client storage cost, and require multiple rounds of communication. A wide set of techniques attempts to mitigate the above issues, trading security for efficiency. Below we focus mainly on those targeting range queries.

One class of techniques relies on deterministic encryption (DET) that maps two equal plaintexts to the same ciphertext [18, 19, 20] (its counterpart being probabilistic encryption), enabling equality queries on encrypted data. These schemes perform bucketization (accompanied by indexing) of data by encrypting based on the query attribute, and reduce range search to a set of equality queries that retrieve matching buckets. Although these schemes are quite efficient, they inherit the drawback of DET, which discloses the distribution of the data (since the bucketization essentially reveals a histogram of the data on the query attribute). Moreover, they do not offer rigorous security definitions and proofs. Another class of methods [2, 3, 23, 27, 30] utilizes Order Preserving Encryption (OPE). The latter has the property that the ciphertexes preserve the order of the plaintexts. Therefore, efficient traditional indexes can be built on the ciphertexes, in the same manner as on plaintexts. OPE is deterministic and, thus, inherits the data distribution leakage of DET. In addition, it also leaks the order of the data and, hence, offers even weaker security than DET.

Searchable Symmetric Encryption (SSE) [5, 6, 7, 8, 9, 14, 21, 22, 33, 34, 39, 26] has been initially proposed in the context of keyword search, having as main goals to (i) introduce rigorous security definitions, and (ii) enable the design of schemes that avoid the leakages of DET, while retaining high efficiency. There is currently no off-the-shelf SSE scheme that supports range queries. As we shall see, in our work (as well as in [26] described below) we effectively reduce range search into multi-keyword search. The only SSE scheme that seems applicable to multi-keyword search is that of Cash et al. [6] which targets Boolean queries and can express multi-keyword search as a disjunction of keywords. Unfortunately, disjunctions in [6] are answered in linear time in the number of documents, which conflicts with our performance desiderata.

The work closest to ours is the basic scheme of Li et al. [26]. Note that the authors also introduce another two schemes, which however share similarities with OPE and, hence, inherit its drawbacks explained above. In the sequel, any reference to [26] implies the basic scheme. This scheme assumes a binary tree over the query attribute domain, and computes for every tuple \( d \) in the dataset \( D \) the log \( m \) dyadic ranges covering its attribute value, where \( m \) is the domain size. Let the dyadic ranges of item \( d \) be denoted by \( DR(d) \). Li et al. create a binary tree as follows. The root initially corresponds to all data items, and stores a Bloom filter [1] over \( \{ DR(d) : d \in D \} \). The algorithm works recursively, starting from the root and working top-down. At each node, it randomly permutes and splits the data items in two sets, each corresponding to one child. Then, it stores a Bloom filter over the DR values for the data items corresponding to each node. Eventually, each leaf contains a Bloom filter indexing only \( DR(d) \) for a single \( d \). A range query is answered by splitting the range into its \( O(\log R) \) minimal dyadic ranges, where \( R \) is the range size over the domain, and traversing the tree by checking whether the Bloom filter in each node “contains” some minimum dyadic range.

The costs of [26] are included in Table 1. The scheme fixes the ratio of the false positives (inherent to Bloom filters) at each node. This results in \( O(n \log n \log m) \) storage cost, and \( O(r) \) false positives, where \( n \) is the dataset size and \( r \) the result size. Moreover, the query size is \( O(\log R) \), whereas the search time is \( \Omega(\log n \log R + r) \). Note that it is difficult (and out of our scope) to find a tight upper bound for the actual search time, due to the random perturbation of the items in the tree and the false positives, but [26] points out that this could be \( O(r \log n) \). Performance aside, the most severe drawback of [26] is its security. First, it relies on the SSE definitions from Goh [14] which have been proven weak by [7, 9]; at a very high level, Goh [14] implies that the privacy of the encrypted queries (trapdoors) is not protected, whereas [9] protects both the data and the queries. Second, [26] focuses on non-adaptive adversaries; from a practical point of view, this means that [26] is secure only in applications that allow the users to ask all their queries once in a batch, and then they shut down. Contrary, it is not secure against adversaries that ask some queries first and, based on the responses they get, then adapt their attacks and ask more targeted queries later. This considerably limits the applicability of this solution in realistic settings. Finally, [26] does not support updates. We introduce Constant-BRC/URC (Section 5) and Logarithmic-BRC/URC (Section 6.1) that subsume [26] in all aspects.

There is a long line of work on creating dynamic SSE (DSSE) schemes to handle updates [5, 21, 22, 34]. The challenge in these works is to achieve a property called forward privacy; the server should not learn that a newly added item satisfies a query issued in the past. Most solutions focus on creating a dynamic secure index. In our work, we take an alternative approach that satisfies forward privacy, by utilizing only static SSE schemes and combining them with efficient bulk-loading techniques adopted from large-scale database systems (such as Vertica [25]).

Range queries have also been studied in a different setting where there are multiple parties contributing to the owner’s dataset using her public key, and the owner issues its range queries on this collective dataset with her secret key [4, 32]. This setting is based on asymmetric cryptography and, hence, entails considerably higher computational costs than our schemes.

2.2 Preliminaries

We explain in turn two range covering techniques with binary trees, the PRF and DPRF cryptographic tools, and the required definitions we adopt from the SSE literature.

Range covering techniques. Consider a domain \( A \). We construct a full binary tree over its values bottom-up. Given a range (i.e., a sequence of contiguous values) over \( A \), a range covering technique selects a set of nodes whose subtrees collectively cover the given range entirely. We will describe two techniques, best range cover (BRC) and uniform range cover (URC). BRC essentially selects the minimum number of nodes that cover exactly the range (also called minimum dyadic intervals). For range size \( R \), there are \( O(\log R) \) such
nodes. In Figure 1, for \( A = \{0, \ldots, 7\} \), BRC covers range [2, 7] with nodes \( N_{2,3} \) and \( N_{4,7} \) (shown in gray).

Consider now range [1, 6] which has the same size as [2, 7]. BRC covers [1, 6] with nodes \( N_1, N_{2,3}, N_{4,5} \) and \( N_6 \), i.e., with a different number of nodes at different levels. We shall see later that this leads to extra leakage, since the number of nodes covering a range may imply where an encrypted range query may or may not be. Motivated by this fact, [24] introduces URC in the context of DPRFs (explained below). In particular, [24] points out that there is always a worst-case decomposition of any range of a given size \( R \) into a certain number of nodes at certain levels. Interestingly, this decomposition retains the \( O(\log R) \) complexity, regardless of where this range is placed over the domain. Briefly stated, URC starts with the set of nodes output by BRC, and keeps on "breaking" certain nodes into their two children, until there is at least one node for each level 0, \ldots, \text{max}, where \text{max} is the highest level of nodes in the result. In the example of Figure 1, for range [2, 7], URC initially invokes BRC and retrieves \( N_{2,3} \) and \( N_{4,7} \). These nodes are at levels 1 and 2, respectively, but there is no node at level 0. Subsequently, it breaks \( N_{4,7} \) into \( N_{4,5} \) and \( N_{6,7} \), as well as \( N_{2,3} \) into \( N_2 \) and \( N_3 \). Now [2, 7] is represented by nodes \( N_2, N_3, N_{4,5}, N_{6,7} \) (enclosed in boxes in Figure 1), i.e., by two nodes at level 0 and two nodes at level 1. Observe that [1, 6] is also represented by the same number of nodes at respective levels.

**PRFs and DPRFs.** A Pseudorandom Function (PRF) family \( \mathcal{F} \) is a set of functions \( \{f_k : A \rightarrow B \mid k \in K\} \), where \( A, B, \mathcal{F}, K \) are indexed by a security parameter \( \lambda \) and such that \( f_k(\cdot) \) is efficiently computable. The main property of a PRF is that an adversary that does not know the secret key \( k \) can distinguish \( f_k(\cdot) \) from a truly random function only with negligible probability in \( \lambda \), written as \( \negl(\lambda) \).

A Delegatable PRF (DPRF) [24] is an enhancement of a PRF with an extra property: the party that knows the secret key \( k \) of \( f_k \) can allow another party that does not possess \( k \) to derive DPRF values for a subset of the domain \( A \). The benefit is performance: the secret holder generates and outsources a small set of intermediate values, which can be used by an untrusted third party to produce an exponential number of DPRF values. Similar to a PRF, the intermediate and final DPRF values appear to be random.

In [24] the DPRF values are computed using the seminal GGM pseudorandom generator [16]. This is defined as \( G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda} \), i.e., on a \( \lambda \)-bit input \( x \), \( G(x) \) produces two \( \lambda \)-bit outputs \( G_0(x) \) and \( G_1(x) \) that appear to be random. Let \( a_\ell \ldots a_0 = a \in A \) be some \( \ell \)-bit domain value. Its DPRF value using secret key \( k \) is computed as \( f_k(a_\ell \ldots a_0) = G_{a_0}(\ldots(G_{a_1}(k))) \), i.e., \( k \) serves as the seed to successive computations of \( G \). For example, the binary representation of 6 is \( (110)_2 \). In Figure 1, observe that 6 is reached by a path starting from the root that chooses right, right and left. Assigning 0 to left and 1 to right, this path traversal uniquely identifies the binary expression of value 6. The DPRF of 6 is \( f_k(6) = G_0(G_1(G_1(k))) \). The GGM values are practically organized into binary tree, hereafter called GGM tree.

The purpose of this particular construction is to permit delegation. Let us focus on range [4-7], completely covered by node \( N_{4,7} \) in Figure 1. Observe that, given \( G_1(k) \) and without possessing \( k \), one can derive all DPRF values for 4-7; \( G_1(k) \) is associated with node \( N_{4,7} \), and all values corresponding to its descendants can be derived by applying \( G \) successively using \( G_1(k) \) as the seed and choosing the \( G_0 \) (left) or \( G_1 \) (right) output based on the path.

When the input range is not covered completely with a single node, it is decomposed into a set of multiple nodes covering the range (following BRC or URC), and the appropriate GGM values corresponding to those nodes (paired with the node level) are provided. The receiver of these values can then easily derive all the DPRFs within the range. The GGM values are called tokens and produced by a function \( T \) that implements BRC or URC, whereas the derivation of the actual DPRF values (corresponding to the leaves) is performed by a function \( C \). Both \( T \) and \( C \) are part of the specification of the DPRF function family.

**SSE definitions.** Let \( D \) be a collection of documents, where a document can be any data item, even a tuple. Each document \( d \in D \) has a unique id, which could be the actual document id or an alias that allows easy mapping to \( d \). Every \( d \) is also associated with a set of keywords from a dictionary \( \Delta \). We represent by \( id(w) \) the ids of the documents that contain \( w \). SSE schemes focus on building an encrypted index \( I \) on the document ids. For simplicity, we concentrate only on the ids, since the actual documents are encrypted independently and stored at the server separately from \( I \); once some id is retrieved during search, the server can send the corresponding document to the owner, who decrypts in a final step that is orthogonal to the SSE instantiation.

An SSE protocol involves an owner and a server and consists of the following algorithms:

\[
k \leftarrow \text{Setup}(1^\lambda) \quad \text{: is a probabilistic algorithm run by the owner before commencing the system. It takes as input security parameter \( \lambda \) and outputs a secret key \( k \).}
\]

\[
I \leftarrow \text{BuildIndex}(k, D) \quad \text{: is a probabilistic algorithm run by the owner prior to sending its data to the server. It takes as input the secret key \( k \) and the data collection \( D \), and outputs an encrypted index \( I \) built on the document ids. Index \( I \) is sent to the server, along with the actual encrypted documents.}
\]

\[
t \leftarrow \text{Trprd}(k, w) \quad \text{: is a deterministic algorithm executed by the owner when issuing a query. It takes as input key \( k \) and keyword \( w \), and outputs a token \( t \).}
\]

\[
X \leftarrow \text{Search}(t, I) \quad \text{: is a deterministic algorithm run by the server to retrieve the ids of the documents containing the query keyword. It takes as input a token \( t \) corresponding to the query keyword and the encrypted index \( I \), and outputs a set \( X \) of document ids.}
\]

In state-of-the-art SSE constructions [9, 8, 22, 21, 34, 6, 5], \( I \) is essentially an encrypted inverted index, which allows

![Figure 1: Covering range [2, 7] with BRC and URC](image-url)
Figure 2: SSE ideal-real security game

efficient retrieval of the document id list corresponding to the query keyword. The token t constitutes auxiliary information that allows the server to partially decrypt only the index components that lead to the retrieval of the result ids. However, once these index portions are decrypted, they become permanently known to the server. In other words, SSE inherently introduces certain information leakage.

An ad-hoc way of defining security would be to outline a set of adversarial attacks, and prove that the scheme is robust against these attacks. This is dangerous as we cannot anticipate the types of attacks an adversary is able to launch. A rigorous way to define security is to formulate the leakage, and prove that the adversary learns nothing more than this leakage. Curtmola et al. [9] introduced a framework for achieving this, following the seminal ideal-real paradigm by Goldreich [15]. In particular, after formulating leakage, we define two games. The real is essentially the execution of the actual SSE protocol. The ideal is a simulation of the real, i.e., an attempt to “fake” the real game, knowing only the formulated leakage. Finally, we prove that an adversary can distinguish the output of the first from that of the second with only negligible probability. Intuitively, this means that the adversary indeed does not learn anything more than the leakage, otherwise he would be able to distinguish the real from the ideal execution with non-negligible probability.

We focus on semi-honest, adaptive adversaries. “Semi-honest” means that the adversary is curious to infer information during the execution of the protocol, but does not deviate from the protocol. “Adaptive” means that the adversary attempts to learn information even in between query executions, and may adaptively select the next query based on the previous ones. A non-adaptive adversary submits all queries before starting to learn information. Clearly, adaptive adversaries are more realistic in database applications where the queries are not presented all at once to a system.

For completeness and to facilitate presentation in Section 3, in Figure 2 we present the SSE ideal-real games for (semi-honest) adaptive adversaries, as introduced in [10]. In RealSSE, an adversary A interacts with the actual SSE protocol, choosing the initial document set and (adaptively) the keyword queries. The adversary gets access only to BuildIndex and Trpdr, since it does not know the secret key k. st_A is some state maintained by the adversary. The final view of A is the encrypted index I, and the set of generated tokens t and st_A. Now observe the line correspondence between RealSSE and IdealSSE. In the latter, a simulator S (maintaining state stF) is enforced with “faking” BuildIndex and Trpdr for the same D and query keywords, only using leakage functions L_1 and L_2 (explained below).

Security boils down to returning (I, t, st_A) that is distinguishable with negligible probability from the output by the real game. The challenge lies in properly using leakages L_1 and L_2 to create I and t, such that (i) they “look” like those produced by real, and (ii) the Search algorithm in ideal is consistent, i.e., it functions similarly to that in real.

Although our schemes are independent of the underlying SSE construction, in order to provide context, we describe the leakage functions L_1, L_2 assuming the SSE scheme by [6]. L_1 is associated with what is leaked from the index alone, whereas L_2 accounts for the leakage from the queries.

- L_1(D) = \max_n \max_w (i.e., an upper bound on the size n of D).
- L_2(D, W) = (\alpha(W), \sigma(W))

W is a set of keywords, \alpha(W) = (\text{id}(w))_{w \in W} is the access patterns, i.e., the document ids returned by each keyword query, and \sigma(W) is the search patterns, i.e., for every pair w_i, w_j \in W such that i \neq j, it indicates whether w_i = w_j or w_i \neq w_j.

In the next section, we explain that with minimal changes and by introducing extra leakage, we can use the described SSE security game to formalize the security of our RSSE framework as well.

3. PROBLEM DEFINITION

We define the problem of Range Searchable Symmetric Encryption (RSSE) in a very similar manner to SSE. In fact, the security game of RSSE is identical to that of SSE in Figure 2, where each w_i stands for a range query rather than a keyword. Moreover, similar to SSE, RSSE captures index-based schemes in its game; this is different from OPE that formulates a secure encryption scheme that allows ordered comparisons, without the need of an index. We employ this definitional framework to devise solutions that reduce range query search to multi-keyword search. This further allows us to build secure solutions on top of an existing secure SSE construction.

More specifically, we assume that data owner possesses a dataset D of tuples. We focus on range queries on a single attribute with domain A. We associate a pair (id, a) with each tuple d \in D, where id is a unique identifier for d and a is the value of d on A. We also write d.id and d.a to refer to the elements of this pair. We assume that the owner encrypts each d \in D using a semantically secure encryption scheme, and sends the resulting ciphertext c along with d.id to the server. The goal is to build a secure index I on the d.id values, such that the server can perform range queries that retrieve the set of ids of the tuples satisfying the query. Note that, for each returned result d.id, the owner can retrieve from the server the corresponding ciphertext c of d and decrypt it in a subsequent step, orthogonal to the search process. In a nutshell, our RSSE framework can be summarized in the following main points:

- **Index creation**: Break A into a set of (potentially overlapping) ranges, and attribute a unique keyword to every range. Regard each d \in D as a document, and associate it with the keywords of the ranges that include d.a. We hereafter use terms document and tuple interchangeably. Utilize a static SSE scheme to securely index D using as dictionary D the union of the range keywords of every d \in D.

\footnote{We assume that the values of A are positive integers. Note that we can always convert any real domain to a discrete positive one by proper scaling and translation.}
An RSSE protocol is specified by algorithms Setup, BuildIndex, Trpdr and Search, which are defined identically to those described in Section 2.2 for static single-keyword SSE, where $w$ now stands for the query range. Their instantiation in each of our proposed schemes varies, but builds upon the constructions of traditional SSE. Our contribution revolves around the proper arrangement of keywords to tuples in BuildIndex, the mapping of a range query to keywords/tokens in Trpdr, and potentially the adjustment of Search to function appropriately with the tokens of Trpdr.

Finally, we support updates using only static SSE schemes. Our goal is forward privacy, i.e., the server should not learn that a newly added item satisfies a query issued in the past. Our mechanism is generic capturing all schemes and, thus, is detailed separately in Section 7.

4. QUADRATIC SCHEME

Our Quadratic scheme is a naive baseline whose sole purpose is to help in conveying the basics of our RSSE framework. Let $A$ be the query attribute domain, and $m$ its total size. There are $O(m^2)$ possible range queries that can be applied in this domain. We enumerate all these possible sub ranges of $A$ and assign a unique keyword to each range. Observe that a domain value belongs to $O(m^2)$ sub ranges. We associate each $d \in D$ with the keywords corresponding to the $O(m^2)$ ranges covering $d.a \in A$.

We start by replicating each $d \in D$ into $d_1, \ldots, d_n$, where $n$ is the number of keywords $d$ corresponds to, and include the replicated tuples in a new dataset $D'$. We then use any secure (single-keyword) SSE scheme off-the-shelf (i.e., without any changes), treating each $d' \in D'$ as a separate tuple. The Setup and BuildIndex algorithms are identical as in the SSE scheme, where the tuples in $D'$ are now augmented with the keywords described above. Given a range query, Trpdr simply maps it to the single keyword associated with its range, and the rest of the algorithm is the same as in the SSE scheme. Finally, Search is also used as in SSE without changes, and returns exactly the tuples in $D'$ containing the range keyword. By definition, the returned tuples are exactly those satisfying the range query (without replication).

Table 1 in Section 1 shows the costs of Quadratic. Each tuple is associated with at most $O(m^2)$ keywords and, hence, the index size is $O(nm^2)$, where $n$ is the number of tuples in $D$. The search time is inherited from the SSE scheme, and assuming [6] this is $O(r)$, where $r$ is the number of results. The query size is $O(1)$ as it involves a single keyword/token.

In terms of security, this technique does not introduce any additional leakage to what SSE reveals for $D'$, namely its size. However, this may disclose information about the distribution of the values of $D$ on $A$; two datasets with different distributions (e.g., one where all tuples have the same $d.a$ value, versus one where they all have a different one) will result in different $D'$ sizes. This can be easily tackled by padding (e.g., as in [9, 5]); for any $D'$, the mechanism takes as input the cardinality $n$ of $D$ and the domain size $m$, and always constructs a secure index corresponding to the maximum possible $D'$ size. Hence, only $n, m$ are leaked in Quadratic in $L_1$ along with the $L_2$ leakage of the underlying SSE scheme, which results in the highest security level for our setting. However, Quadratic clearly suffers from a prohibitive storage cost, which motivates our next solutions.

5. CONSTANT SCHEMES

In this section we present our Constant-BRC and Constant-URC schemes, which lie on the other side of the spectrum as far as the storage cost is concerned. Specifically, these techniques introduce a constant asymptotic overhead on the index size with respect to the dataset size $n$. Before embarking on their description, we first explain a naive variant.

We assign to each tuple $d \in D$ a single keyword, which corresponds to its actual value on $A$, namely $d.a$. In other words, the dictionary $\Delta$ is the values in $A$. No replication is involved. We then index $D$ with an SSE scheme, yielding an index $I$ of size $O(n)$. A query range of size $R$ is simply mapped to $R$ keywords, one for each value of $A$ it covers. We trivially use these keywords as search tokens in the Trpdr and Search algorithms of the SSE scheme. The scheme will return the correct $r$ results without false positives in $O(R + r)$ time. Disregarding security for now, the main drawback of this scheme is the potentially unacceptable query size $O(R)$ for very large ranges. This motivates our Constant-BRC and Constant-URC solutions, which take advantage of DPRFs (explained in Section 2.2) to reduce the query size to $O(\log R)$.

The instantiations of the RSSE algorithms for Constant-BRC and Constant-URC are the following (the two algorithms differ only in the range covering technique used for generating the trapdoor – BRC or URC):

$$k \leftarrow \text{Setup}(1^\lambda): \text{Output } (k_1, k_2), \text{ where } k_2 \text{ is a DPRF key and } k_3 \text{ is the key of the underlying SSE scheme.}$$

$$I \leftarrow \text{BuildIndex}(k, D): \text{Associate each } d \in D \text{ with keyword } d.a. \text{Invoke the BuildIndex algorithm of the SSE scheme on } D \text{ and its keywords, but instead of using a PRF to encrypt the ids, use a DPRF instead. Each } d.id \text{ is decrypted only with token } f_k(d.a), \text{ where } f_k \text{ is a DPRF function.}$$

$$t \leftarrow \text{Trpdr}(k, w): \text{Invoke the token generation function } (T) \text{ of the DPRF employing either BRC or URC, and retrieve the corresponding GGM values corresponding to range } w \text{ from the GGM tree over } A \text{ (see Section 2.2). Randomly permute these GGM values and output them as vector } t. \text{ For each GGM value in } t, \text{ provide the level of its respective node in the GGM tree as well.}$$
$X \leftarrow \text{Search}(t, I)$: Derive the (leaf-level) DPRF values from the GGM values in $t$. Use these values as tokens in the Search algorithm of SSE and return the results.

We clarify the algorithm revisiting the example of Figure 1, where $A = \{0, \ldots, 7\}$. The owner first generates a DPRF key $k_1$ in Setup, and computes the DPRF values for the elements on $A$ that appear in $D$, creating a GGM tree. Suppose that $a \in D$ has $d.a = 6$. In BuildIndex, the owner assigns 6 as keyword to $d$. Suppose that $a$, a DPRF key algorithm as SSE (using SSE key BuildIndex where this document can be decrypted by using as token a DPRF value on keyword 6, Constant-BRC/URC use a DPRF value instead. Specifically, they invoke the same BuildIndex algorithm as SSE (using SSE key $k_2$ generated in Setup), but the token to decrypt $d$ (or any other tuple $d$ with $d.a = 6$) is $f_k(6) = G_0(G_1(G_1(k_1)))$. The algorithm proceeds similarly with every other $d \in D$.

Upon a query, $\text{Trpdr}$ outputs as token $t$ the GGM values corresponding to the nodes covering the range with BRC or URC. In our example in Figure 1, if BRC is used, then the output is $t = \langle (G_1(G_0(k_1))), 1), (G_1(k_1), 2) \rangle$, i.e., the GGM values of nodes $N_{2.3}$ and $N_{4.7}$ along with their levels, respectively. If URC is used, then $t$ contains $(G_0(G_1(G_0(k_1))), 0)$ for node $N_{2.3}$, $(G_1(G_0(k_1)), 0)$ for node $N_3$, $(G_0(G_1(k_1)), 1)$ for $N_{4.5}$, and $(G_1(G_1(k_1)), 1)$ for $N_{6.7}$. Note that the elements of $t$ are randomly permuted.

In Search, the server first takes the GGM values of the non-leaf nodes and expands them to compute the DPRF values of GGM nodes. For instance, from $N_{2.3}$’s value $G_1(G_0(k_1))$, it generates DPRFs $G_0(G_1(G_0(k_1)))$ and $G_1(G_0(G_0(k_1)))$. It can do that because (i) $G$ is public, and (ii) it knows the level of $G_1(G_0(k_1))$, i.e., 1. It finally uses as tokens the DPRFs to retrieve the results, invoking SSE’s Search algorithm.

The cost complexities of Constant-BRC and Constant-URC are identical and provided in Table 1. Each tuple is associated with a single keyword and, hence, the storage cost is $O(n)$. Due to the BRC/URC techniques, the query size is $O(\log R)$ for a range size $R$. The search time at the server entails expanding the $O(\log R)$ GGM values into $R$ DPRFs, and retrieving the $r$ results from SSE, yielding a total $O(R + r)$ time. Both solutions do not introduce false positives.

We next turn to security. Our Constant constructions cannot be proven secure against adaptive adversaries that are allowed to issue intersecting range queries. This is an inherent limitation of our underlying DPRFs, as shown in [24]. Briefly stated, this is because the simulator must have an a priori knowledge of the GGM sub-structure shared by the intersecting ranges, in order to produce consistent tokens. However, this is not possible in the adaptive case. In the example of Figure 1, suppose that some query generated a token that includes the GGM value for node $N_{2.3}$. Then, let another query involve producing as token a GGM value for node $N_2$. Without the a priori knowledge that the second range intersects the first at $N_2$, the simulator could not have generated the GGM value for $N_{2.3}$, in a way that it can produce the GGM value for $N_2$.

Consequently, the Constant schemes limit the functionality by not allowing query intersections. Note that this constraint can be enforced at the application level. For instance, the owner’s program may maintain the history of queries and abort when an intersecting query is seen, or may try to answer the query from cached answers of previous queries that collectively encompass the new query range.

To prove security (under the constraint of non-intersecting queries), we define the two leakages as follows:

- $L_1(D, A) = \langle m, n \rangle$
  - $D$ is the dataset, $A$ is the query attribute domain, $n$ is the cardinality of $D$, and $m$ is the size of $A$.
- $L_2(D, A, W) = \langle \alpha(W), \sigma(W), (\mu(N_i), \ell(N_i), id_{map}(N_i))_{N_i \in RC(w) \cap W} \rangle$
  - We will include the security theorems and proofs of all our schemes in the long version of our paper.

**Observation**. This is because our constructions build an index considering the entire span of $A$. We further explain the extra leakage in $L_2(D, A, W)$ incurred by our schemes with an example using Figure 1. Let the first query be $w_1 : [0, 3]$, with results $d_1, d_2$, such that $d_1.a = 0$ and $d_2.a = 3$. Then, $L_2$ over $W = \{w_1\}$ is an alias for node $N_{0.3}$, its level (2), and the information that $d_1$ maps to its left-most leaf, and $d_2$ to its right-most leaf. Now suppose that the second query is $w_2 = [5, 7]$. Then, $L_2$ over $W = \{w_1, w_2\}$ is what explained above, plus aliases for nodes $N_5, N_{0.7}$ (without disclosing their relative order) as well as the mappings of the qualifying tuples in these sub trees. Note that, neither the relative order of $w_1, w_2$ on $A$, nor the relative order of the sub trees of each query are revealed.

We will include the security theorems and proofs of all our schemes in the long version of our paper.

**Qualitative comparison**. The Constant schemes feature considerably better storage than Quadratic (at the expense of slightly increased query size and search time). However, they also introduce significantly higher leakage, since they disclose the exact mapping of the results in a sub tree, which further reveals relative order information. Comparing the BRC and URC variants, URC offers slightly better privacy; the BRC coverage may exclude mapping certain ranges to the query, whereas URC covers all ranges of the same size in an indistinguishable manner.

### 6. LOGARITHMIC SCHEMES

Motivated by the high structural leakage of the Constant schemes, in this section we design solutions that trade off storage for privacy. This is achieved by replicating tuples similar to Quadratic, but with a significantly lower expansion factor. Specifically, we present four constructions that increase the storage complexity only by a logarthmic factor, and differ in the other costs, the privacy level, and the false positives. Section 6.1 describes Logarithmic-BRC and Logarithmic-URC, Section 6.2 explains Logarithmic-SRC, and Section 6.3 presents Logarithmic-SRC-i.

#### 6.1 Logarithmic-BRC/URC

The Logarithmic-BRC and Logarithmic-URC schemes mitigate the structural leakage of Constant-BRC/URC by avoiding the use of DPRFs and associating each tuple with a
logarithmic, instead of constant, number of keywords. The
RSSE protocol for Logarithmic-BRC and Logarithmic-URC
is as follows (the two schemes again differ in the range cov-
ering technique used in Trpdr):

\[ k \leftarrow \text{Setup}(1^k): \text{Same as in SSE.} \]

\[ I \leftarrow \text{BuildIndex}(k, D): \text{Build a binary tree over domain } A \text{ as} \]
described in Section 2.2, and assign a unique keyword at
each node. For each tuple \( d \in D \), find the nodes
on the path from the tree root to \( d.a \), and associate
\( d \) with the node keywords. Replicate every \( d \) for each
keyword it is associated with, and regard each replica
as separate tuple as discussed in Quadratic. Let \( D' \) be
the resulting dataset including all the replicas.Invoke the
\text{BuildIndex} algorithm of the SSE scheme on \( D' \) and
its keywords, after randomly permuting the documents
that are associated with the same keyword.

\[ t \leftarrow \text{Trpdr}(k, w): \text{Let } w \text{ represent the query range. Find} \]
the nodes that cover \( w \) using BRC or URC. Create a token
for each node keyword invoking the \text{Trpdr} algorithm of
the SSE scheme and include it in \( t \). Randomly permute
\( t \) prior to returning it.

\[ X \leftarrow \text{Search}(t, I): \text{Invoke the Search algorithm of SSE for} \]
every element in \( t \), and output the union of the results.

The protocol associates each tuple with the dyadic inte-
vals that cover its attribute value. For instance, if \( d.a = 3 \)
in Figure 1, then \( d \) is associated with keywords \( N_0,7, N_0,7, \)
\( N_2,3 \) and \( N_2,7 \) (where each \( N_i \) is the label of a node). We
then replicate each \( d \) and index the augmented dataset \( D' \)
with traditional SSE. Given a range query, we compute its
cover with BRC or URC, and issue the output node labels
of the range covering technique as keywords for traditional
SSE search, i.e., we invoke the conventional \text{Trpdr} and \text{Search}
algorithms of SSE for every node label and corresponding to-
enken, respectively. For example, for query \([2, 7]\) under BRC,
\text{Trpdr} outputs an SSE token for \( N_2,3 \) and \( N_4,7 \) in random or-
der, and the \text{Search} SSE algorithm is invoked for every such
token separately.

The costs of Logarithmic-BRC and Logarithmic-URC are
summarized in Table 1. The index size is \( O(n \log m) \), where
\( n \) is the number of tuples and \( m \) the domain size, since each
tuple is associated with \( O(\log m) \) keywords. The query size
is \( O(\log R) \), where \( R \) is the size of the query range, since
this is the number of nodes covering the query range in
BRC and URC. The search time is \( O(\log R + r) \), where \( r \)
the result size, because there are \( \log R \) tokens issued to the
underlying SSE scheme, each incurring no additional cost
to the retrieval of its results. Finally, the protocol returns
correct answers without false positives, since BRC and URC
cover the query exactly, and a tuple in the result is certainly
associated with the keyword of a node in the cover.

The only extra information leaked is a partitioning of the
result ids into groups. More formally:

- \( L_1(D, A) = (m, n) \)
  \( D \) is the dataset, \( A \) is the query attribute domain, \( n \)
  is the cardinality of \( D \), and \( m \) is the size of \( A \).

- \( L_2(D, A, W) = \langle \alpha(W), \sigma(W), (\mu(N_i), id(N_i))_{N_i \in RC(w), w \in W} \rangle \)
  \( \alpha(W), \sigma(W) \) are the access and search patterns of the
queries as defined for SSE. For every query range \( w \in \)
\( W \), the leakage contains a tuple that consists of an alias
\( \mu(N_i) \) for every node \( N_i \) returned by the range covering
\( RC(w) \) – where the \( RC \) function is either BRC or URC
– along with the list of tuple ids \( id(N_i) \) associated with
keyword \( N_i \).

**Qualitative comparison.** Logarithmic-BRC/URC feature
increased storage cost compared to Constant-BRC/URC,
but slightly better search time (since the server does not need
to generate DPRFs from the tokens). The main benef-
it of Logarithmic-BRC/URC is in the substantially reduced
leakage, since they hide both the distribution and the total
order of the tuples in each subtree of the query range cover.
The difference between the BRC and URC variants of Loga-
ritic is similar to that in Constant; URC does not disclose
information about the position of the range over the domain.
Nevertheless, what is still leaked in both variants is the par-
titioning of the result tuples into distinct groups (each corre-
spoding to a subtree), which may further disclose ordering
information about the groups. This motivates our solution
in the next section.

### 6.2 Logarithmic-SRC

The extra “result partitioning” leakage of the Logarithmic-
BRC/URC schemes was due to the fact that \text{Trpdr} produces
multiple tokens, one for each subtree with which BRC or
URC cover the query range. Logarithmic-SRC, prevents
this leakage by always covering the query range with a sin-
gle range that is potentially a superset of the query range
(SRC stands for single range cover). In addition to enhanced
privacy, this also leads to constant query size, but may in-
troduce false positives.

We can naively realize Logarithmic-SRC building upon
the same binary tree over \( A \) as in Logarithmic-BRC/URC
as follows. We assign once again to each \( d \in D \) the keywords
corresponding to the nodes whose subtrees cover \( d.a \), and
replicate tuples to create an augmented dataset \( D' \). How-
ever, instead of invoking BRC or URC in \text{Trpdr} to find the
cover of the query range with tree nodes, we could simply
select the node of the smallest subtree fully covering the
query, and use its keyword for searching. In our example
of Figure 1, we would cover query \([2, 7]\) with the tree root
\( N_{0,7} \). Unfortunately, this solution features an unacceptable
worst-case complexity for the number of leaves (i.e., domain
values) contained in the single subtree, which is \( O(m) \) where
\( m \) is the domain size, regardless of the query range size \( R \).
This would further lead to an unacceptable number of false
positives. For example, in Figure 1, query \([3, 4]\) is covered
by the tree root that encompasses the entire domain and,
hence, dataset \( D \). Motivated by the above, in Logarithmic-
SRC we produce keywords for the tuples in \( D \) based on a novel
tree-like Directed Acyclic Graph, henceforth referred to as
TDAG, which ensures that any query range of size \( R \)
is covered by a single subtree with size \( O(R) \).

We explain the TDAG structure using Figure 3. We start by
building a binary tree over domain \( A \), similar to the case
of the previous schemes. We then inject one extra node
between every two nodes at every level of the tree (depicted
in gray), and connect it with the two nodes directly below it
in the next level (i.e., the right child of the node in its left,
and the left child of the node in its right).
The following lemma is useful for our cost analysis of Logarithmic-SRC.

**Lemma 1.** Given a TDAG constructed over a domain $A$ and any range in $A$ of size $R$, there is always a subtree of size $O(R)$ that can completely cover the range.

**Proof.** For any integer $R > 0$, there is an integer $j \geq 0$ such that $2^j \leq R \leq 2^{j+1}$. Any range of size $R$ can be covered by at most 2 dyadic ranges (i.e., subtrees in the binary tree over $A$) of size $2^{j+1}$; either the range is fully contained in such a subtree, or it is split between two consecutive subtrees of size $2^{j+1}$. These two subtrees are either children of the same parent in the binary tree, or cousins. Recall that our TDAG structure horizontally links every two cousins in each level with a new parent. Hence, there is always a node that covers the two subtrees each of size $2^{j+1} \leq 2R$. Thus, for any range of size $R$, there is always a subtree in TDAG with size at most $4R \in O(R)$. □

Given a range of size $R$, the SRC range covering algorithm simply finds the lowest common ancestor of the lower and upper bound of the range, which can be performed in $O(\log R)$ time. In the example of Figure 3, SRC covers range [2, 7] by $N_{0,7}$, and range [3, 5] by $N_{2,5}$.

The RSSE algorithms for Logarithmic-SRC are the same as in Logarithmic-BRC/URC with the following differences: (i) in BuildIndex, each $d \in D$ is associated with the keywords/labels of the nodes of TDAG that cover $d.a$ (instead of the nodes of the binary tree), and (ii) Trpdf generates a single token for the node label output by the SRC covering technique (instead of BRC/URC).

Table 1 summarizes the costs of Logarithmic-SRC. The query size is constant, since the query is represented by a single token. The index has size $O(n \log m)$; for each tuple $d \in D$, there are $O(\log m)$ nodes in the path from the root to $d.a$, and each such node is connected to at most one injected node in the TDAG; the subtrees of all these $O(\log m)$ nodes cover $d.a$ and, thus, each such node is associated with $O(\log m)$ keywords. The false positives depend on the dataset distribution over $A$. If the distribution is uniform, then the false positives are $O(R)$ due to Lemma 1. However, if the dataset is skewed, then the false positives could be up to $O(n)$. For example, if [3, 5] is the query in Figure 3 and there is a single tuple that satisfies the range, but the rest of the dataset has value 2 on $A$, then Logarithmic-SRC will return the entire dataset due to the used keyword $N_{2,5}$. The search time is linear in the result size plus the false positives and, thus, it is $O(n)$ in the worst case where there is heavy skew.

In Logarithmic-SRC, all range queries are reduced to single-keyword queries and, thus, the scheme degenerates to SSE, inheriting its security (assuming the padding technique discussed in Quadratic). However, for technical reasons in our proofs, we must define an extra subtle leakage, namely the fact that two different ranges may map to the same keyword.

This can be modeled by extending the definition of search patterns. Nevertheless, from a practical point of view, this leakage is not observable by an adversary, since the mapping takes place at the owner. In particular, the adversary is unable to distinguish if the same token was produced twice for the same or for different range queries.

**Qualitative comparison.** Contrary to Logarithmic-BRC/URC, in Logarithmic-SRC the adversary is unable to infer ordering information about the results, since each range is mapped to a single keyword and the tuples associated with this keyword are randomly permuted. Logarithmic-SRC also features optimal query size and the highest achievable privacy in the our RSSE framework that builds upon single-keyword SSE. Similar to Logarithmic-BRC/URC, this comes at the cost of extra storage. Logarithmic-SRC is ideal for datasets with uniform distributions over the query attribute domain. However, it may feature an unacceptable number of false positives (and, thus, also search time) under heavy data skew. This is mitigated by our final solution described in the next section.

### 6.3 Logarithmic-SRC-i

The Logarithmic-SRC-i construction aims at reducing the false positives of Logarithmic-SRC from $O(n)$ to $O(R + r)$, where $R$ is the query range size and $r$ is the result size. It achieves this by building a double range $I = I_1 \cup I_2$, where $I_2$ indexes the tuples in $D$ similar to the previous schemes, and $I_1$ is an auxiliary index that guides the search to $I_2$. This construction is interactive (hence the "i" in the name), i.e., it involves an extra round of communication between the owner and the server during the query: the owner first queries $I_1$, it receives the result from the server, and based on the result it queries $I_2$.

We illustrate the construction of $I_1$ and $I_2$ with the example of Figure 4. Suppose that $D = \{d_0, \ldots, d_{15}\}$. Assume also for simplicity that $d_0, \ldots, d_{15}$ are sorted on $A$, so that the subscript $i$ of each $d_i$ implies its position in the total order of the tuples on $A$. Also consider that $d_0.a = \ldots, d_{15}.a = 2$, $d_{10}.a = 4$, $d_{11}.a = d_{12}.a = 5$, $d_{13}.a = d_{14}.a = 6$, and $d_{15}.a = 7$. Consider TDAG1 in the upper part of the figure, which is built on domain $A = \{0, \ldots, 7\}$, and let $[3, 5]$ be the query range. Recall that Logarithmic-SRC answers this query with TDAG1 by using keyword $N_{2,5}$. This returns as false positives $d_0, \ldots, d_9$ corresponding to domain value 2, which comprise more than half of $D$.
Instead of using TDAG to index the tuples, Logarithmic-SRC-i uses it to index the ranges of tuples corresponding to the domain values, where a tuple range accounts for a range of tuple subscripts. Specifically, each leaf is associated with a pair (domain value, tuple range), e.g., \((2, [0, 9])\), where \(N_2\) is associated with (domain value, tuple range) of the leaves in their subtree, e.g., \(N_{2,5}\) corresponds to \((2, [0, 9]), (4, [10, 10]), (5, [11, 12])\). \(I_1\) is constructed using the traditional SSE BuildIndex algorithm, where the documents are the pairs or lists of pairs described above, and the keywords are the TDAG node labels. The construction then builds a second TDAG, denoted by TDAG2, in the lower part of the figure, which is built on the tuples sorted on \(A\). Note that the order of the documents corresponding to the same keyword (i.e., to the same node in TDAG1) does not affect the structure of TDAG2. For instance, TDAG2 would be equivalent if we swapped \(d_{11}\) with \(d_{12}\) in the leaf level (where \(d_{11}\) and \(d_{12}\) have the same keyword). Prior to constructing TDAG2, we randomly shuffle the documents corresponding to the same keyword. Each node corresponds to the tuples covered by its subtree, e.g., \(N_{10,13}\) is associated with \(d_{10}, d_{11}, d_{12}\) and \(d_{13}\). \(I_2\) is constructed using the traditional SSE BuildIndex algorithm, where the documents are \(d_{10}, \ldots, d_{15}\), and the keywords are the TDAG2 node labels.

The scheme performs the query in two stages. In the first, it issues the query range to \(I_1\). For instance, for \([3, 5]\) it creates a token for \(N_{2,5}\) (following always the SRC range covering technique on the TDAG), and receives \((2, [0, 9]), (4, [10, 10]), (5, [11, 12])\). It then selects the pairs that satisfy the query, and creates a new, single query range on the document subscripts, by merging the qualifying ranges. In our example, it merges \([10, 10]\) and \([11, 12]\) to create new query \([10, 12]\) \((0, 9)\) does not satisfy the original query). In the second stage, it issues \([10, 12]\) to \(I_2\), creating the token for node \(N_{10,13}\) in TDAG2, which fully covers the tuple range. The algorithm returns as result \(d_{10}, d_{11}, d_{12}\) and \(d_{13}\). Observe that now there is only a single false positive, \(d_{13}\), whereas Logarithmic-SRC returned 10 false positives for the same query. The RSSE protocol for Logarithmic-SRC-i is as follows:

\[
k \leftarrow \text{Setup}(1^\lambda): \text{Generate and output two SSE keys } (k_1, k_2).
\]

\[
I \leftarrow \text{BuildIndex}(k, D): \text{Build SSE index } I_1 \text{ on the tuple ranges using TDAG1 with key } k_1, \text{ and index } I_2 \text{ on the sorted tuples on } A \text{ using TDAG2 with key } k_2. \text{ Output } (I_1, I_2).
\]

\[
l \leftarrow \text{Trpdr}(k, w): \text{This is an interactive algorithm. Parse } k = (k_1, k_2). \text{ Generate SSE token } t_k \text{ with } k_1 \text{ for the SRC node on TDAG1 that covers range } w, \text{ and send it to the server. Decrypt the answer to retrieve new range } w'. \text{ Generate SSE token } t_k \text{ with } k_2 \text{ for the SRC node on TDAG2 that covers } w', \text{ and output } (t_1, t_2).
\]

\[
X \leftarrow \text{Search}(t, I): \text{This is an interactive algorithm. Parse } t = (t_1, t_2) \text{ and } I = (I_1, I_2). \text{ Retrieve } t_1 \text{ from the owner, invoke the Search algorithm of SSE on } I_1 \text{ and send the result to the owner. Retrieve } t_2 \text{ from the owner, invoke the Search algorithm of SSE on } I_2 \text{ and output the result } X.
\]

Table 1 illustrates the costs of Logarithmic-SRC-i. The number of documents indexed by \(I_1\) is equal to the number of distinct domain values contained in the dataset, since we index the entire range of tuples on a specific domain value by a single (domain value, tuple range) document of constant size. Therefore, since \(I_1\) is constructed using a TDAG similar to Logarithmic-SRC, its size is \(O(n \log m)\). The number of documents indexed by \(I_2\) is \(O(n)\), i.e., the entire \(D\). Contrary to \(I_1\), the TDAG is built on \(n\) leaves and, thus, the size of \(I_2\) is \(O(n \log n)\). Assuming that \(m\) is typically larger than \(n\), the total storage cost becomes \(O(n \log m)\). The query size is \(O(1)\), since there are only two tokens involved. The false positives are \(O(R+r)\), considering also those of \(I_1\). This is because the range size in \(I_1\) is of size \(O(R)\), whereas in \(I_2\) is of size \(O(r)\); due to Lemma 1, the number of false positives in each index is linear in the query range size and, thus, the total false positives are \(O(R+r)\). The search time is dictated by the number of results plus the false positives and, thus, it is also \(O(R+r)\).

Since \(I_1\) and \(I_2\) are built following the construction algorithm of the underlying SSE protocol and using two different keys, the leakage in each index is identical to that of the SSE scheme. Therefore, having the \(L_1\) and \(L_2\) leakages of SSE for both indexes, we can prove the security of Logarithmic-SRC-i.

**Qualitative comparison.** Logarithmic-SRC-i reduces the false positives as compared to Logarithmic-SRC, even in the case of heavy data skew, while retaining the optimal query size and linearithmic storage cost. As a downside, the usage of the auxiliary index leaks slightly more information than its counterpart. For instance, the size of \(I_1\) (derived from \(L_1\) of \(I_1\)) leaks the number of distinct domain values covered by the dataset, whereas the size of a result from a query to \(I_1\) (derived from \(L_2\) of \(I_1\)) reveals the number of distinct domain values covered by the result. Moreover, in non-skewed datasets, Logarithmic-SRC-i inflicts higher search cost than Logarithmic-SRC. This is because the benefits of the extra \(I_1\) index in Logarithmic-SRC-i to reduce the false positives diminish and, thus, the extra search overhead compared to Logarithmic-SRC becomes evident. In that sense, Logarithmic-SRC-i is better under heavy data skew, whereas Logarithmic-SRC is preferable in non-skewed datasets.

### 7. UPDATES

Recall from Section 2.1 that most dynamic SSE (DSSE) schemes [22, 21, 34, 5] create a dynamic index that introduces the least possible leakage and provides forward privacy, i.e., it does not reveal that a new update satisfies a previous query. We follow an alternative methodology adopting a bulk-loading technique from commercial databases (e.g., Vertica [25]), which is simple from a Security point of view, but (i) builds upon static SSE schemes that are faster and easier to implement than their dynamic counterparts, (ii) enables easy formulation of leakage, and (iii) captures forward privacy. It is also generic; it applies to all our solutions and to any future static RSSE scheme.

We assume that updates come in batches, and each batch \(i\) is treated as a separate dataset \(D_i\). The updates can be insertions of new tuples, or modifications/deletions of old tuples. Every update is treated as an insertion in the new dataset; deletions carry a small flag indicating that the tuple must be removed. For each new dataset \(D_i\), the owner creates a new index \(I_i\) with a fresh key \(k_i\) following the BuildIndex of the utilized construction, and sends the en-
crypted data to the server. Suppose that the owner has uploaded \( b \) such indexes. Upon a query, the owner creates \( b \) separate tokens, one for each index with the corresponding key. The server processes them separately on the \( b \) indexes, and returns the results. The final refinement of the results occurs at the owner, who filters out the deleted tuples and appropriately performs the potential modifications.

Clearly, the number of keys, query size, storage cost, search time and result size increase linearly with \( b \) and, thus, the number of indexes should not increase indefinitely. Therefore, we adopt the approach of Vertica, which essentially organizes the datasets/batches into a log-structured merge tree. Specifically, the owner sets a parameter called consolidation step, denoted by \( s \), which determines how frequently the indexes must be merged. After creating \( s \) new indexes, the owner downloads them, merges their tuples into a single index, re-encrypts the index and sends it to the server. This happens hierarchically; after consolidating \( s \) indexes \( s \) times, the \( s \) merged indexes are further consolidated into a new one. Conceptually, this is like organizing the indexes as leaves of a full \( s \)-ary tree created bottom-up, such that when \( s \) nodes are created in a level, they get consolidated creating a parent node at the next higher level. This leads to an amortized logarithmic merge cost in the number of batches [25]. Although this incurs extra periodic cost at the owner for the consolidation and re-encryption, it retains \( O(s \log b) \) indexes at the server (and keys at the owner) at all times, instead of \( b \). Finally, note that \( s \) should be tuned based on the application at hand. For instance, if the application expects frequent deletions, it is beneficial to set \( s \) to a small value, in order to perform merge operations more frequently, thus eliminating the extra cost for storing the deletions as insertions.

The leakage of this methodology is essentially the entire history of the \( L_1, L_2 \) leakages of every index that was once “active” at the server. For instance, from this leakage one could derive the number of deletions that occurred in one of the batches. Also observe that our technique satisfies forward privacy; every index is encrypted with a fresh key and, thus, a token created for one index in the past cannot be used to decrypt index components produced in the future.

8. Experimental Evaluation

In this section we experimentally evaluate the performance of all our proposed schemes, excluding Quadratic that features a prohibitive storage cost. Based on our discussion in Section 2.1 about the three schemes of [26], we also include a comparison to the basic scheme of Li et al. [26], hereafter referred to as PB, recalling though that the latter offers unrealistically weaker security than our schemes. We also include additional experiments in the Appendix.

Setup. We implemented our solutions in Java, and conducted our experiments on a 64-bit machine with an Intel® Core™ i7-2720QM CPU at 2.2GHz and 16GB RAM, running Linux Ubuntu 14.10. We utilized the JavaX.crypto library for the entailed cryptographic operations. Specifically, we implemented PRF and GGM evaluations with HMAC-SHA-512, and hash computations with SHA-1. We also used AES128-CBC for encryption. We chose the construction by Cash et al. [6] as our underlying SSE scheme, setting its parameters to the values recommended in [6] for space-efficiency \( (S = 6000, K = 1.1) \).

We experimented with two real datasets. The first is from the Gowalla geo-social network (snap.stanford.edu/data/loc-gowalla.html), hereafter called Gowalla. This dataset consists of 6,442,890 user location check-ins in a period between February 2009 and October 2010. We used as query attribute the date/time of the check-ins converted to 32-bit integers and translated such that the domain is \( A = \{0, \ldots, 103017913\} \). The second dataset is from the US Postal Service (www.app.com), called USPS, and contains 389,032 employee records. We used as query attribute the annual salary field with domain \( A = \{0, \ldots, 276840\} \). Note that Gowalla is relatively uniform on \( A \) (95% distinct values on \( A \)), whereas USPS is heavily skewed (5% distinct values on \( A \)). All datasets and respective indexes fit in memory.

Index costs. In the first set of our experiments we assess the index size and construction time in Gowalla when varying the dataset size \( n \), and demonstrate the results in Figures 5(a) and 5(b), respectively. The construction time involves also the I/O cost for reading the dataset from the disk into main memory. As in [26], to retrieve the various datasets, we simply partition the initial dataset (sorted on \( A \)) into 10 sets of 500K tuples each, chosen uniformly at random from the entire data set, start with one partition, and gradually add the rest of the partitions. Note that the BRC and URC variants of the same scheme feature identical costs. The index size entails only the replicated tuple ids and their associated keywords. The curves in both figures have the same trends, since the index size dictates the construction time. Moreover, both the index size and construction time scale linearly with \( n \), since even the logarithmic factors essentially add a constant factor to the overall size.

As expected, the Constant schemes achieve the smallest index size \( (12.63-131 \text{ MB}) \), as well as the lowest construction time \( (288-332 \text{ s}) \). The costs in Logarithmic-BRC/URC increase faster due to the logarithmic factor in the index size. Logarithmic-SRC incurs about twice the size and time compared to Logarithmic-BRC/URC, due to the nodes injected to form the TDAG. Logarithmic-SRC-i requires double the size and time compared to Logarithmic-SRC. Recall that Logarithmic-SRC-i entails building a second index on top of the one in Logarithmic-SRC, whose size depends on the distinct values on \( A \) covered by the dataset. In Gowalla, the tuples cover 95% of the query domain. Therefore, the size of the extra index is almost as large as the basic one, doubling the overall size. Finally, observe that Constant-BRC/URC and Logarithmic-BRC/URC outperform PB [26] in terms of index size, whereas the construction cost of all our schemes is significantly lower than PB.

![Figure 5: Index costs (Gowalla)](image-url)
Table 2 includes the index size and construction time for the USPS dataset. In this experiment we do not vary the dataset size because it is small. Similar to the case of Gowalla, the Constant schemes feature the smallest overheads, Logarithmic-BRC/URC add a constant factor to the costs of Constant, and Logarithmic-SRC incurs almost twice as high costs as Logarithmic-BRC/URC. However, the most interesting observation here is that, contrary to the case of Gowalla, Logarithmic-SRC-i adds minimal overheads to those of Logarithmic-SRC. This is because in USPS there are only 5% distinct values on $A$, which makes the extra index in Logarithmic-SRC-i very compact. Once again, Constant-BRC/URC and Logarithmic-BRC/URC feature a smaller index size than PB, whereas all our schemes have orders of magnitude faster construction time than PB.

Table 2: Index costs (USPS)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Index size (MB)</th>
<th>Constr. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant-BRC/URC</td>
<td>10.30</td>
<td>2.831</td>
</tr>
<tr>
<td>Logarithmic-BRC/URC</td>
<td>195.7</td>
<td>54.967</td>
</tr>
<tr>
<td>Logarithmic-SRC</td>
<td>391.4</td>
<td>106.970</td>
</tr>
<tr>
<td>Logarithmic-SRC-i</td>
<td>419.14</td>
<td>119.662</td>
</tr>
<tr>
<td>PB (Li et al. [26])</td>
<td>299.06</td>
<td>2574</td>
</tr>
</tbody>
</table>

False positives. Figures 6(a) and 6(b) plot the average false positive rate (i.e., the average ratio of false positives over the total result size) as a function of the query range size, computed over the results of 200K random queries on each domain. We evaluate only Logarithmic-SRC-i versus Logarithmic-SRC, since these are the only schemes introducing false positives. The rate decreases almost linearly with the range size, since more tuples previously marked as false positives are contained in the range. In both datasets, Logarithmic-SRC-i incurs fewer false positives, outperforming Logarithmic-SRC by up to 27% in Gowalla and 38% in USPS. In USPS the rate drops more steeply in Logarithmic-SRC-i than PB, whereas all our schemes have orders of magnitude faster construction time than PB.

Figures 7(a) and 7(b) report the average CPU cost in Gowalla and USPS, respectively, for the same 200K queries of Figure 6. We compare all schemes, and provide also the inevitable cost of retrieving the actual results with the underlying SSE scheme of [6]. The search time is dominated by the PRF/DPRF evaluations entailed in [6] for each retrieved tuple. Consequently, Logarithmic-BRC (resp. Constant-BRC) and Logarithmic-URC (resp. Constant-URC) have negligible performance difference and we, thus, group them into a single curve.

Logarithmic-BRC/URC are the fastest and their performance coincides with that of pure SSE. This is expected since their search complexity is $O(\log R + r)$ and $\log R$ adds negligible overhead. The Constant schemes are slightly more expensive, due to the extra expansion of the GGM tokens into $O(R)$ DPRFs. Observe that this additional cost is more pronounced in Gowalla, due to its significantly larger domain (and, thus, query range sizes) than USPS. The SRC-based schemes are more costly, due to the false positives they introduce. In Gowalla, Logarithmic-SRC-i is more expensive than Logarithmic-SRC, due to the extra searches in its auxiliary index. Nevertheless, in USPS, Logarithmic-SRC-i outperforms Logarithmic-SRC, since its savings in false positives outweigh the extra index cost. PB features comparable search cost with that of Constant-BRC/URC and Logarithmic-BRC/URC in the Gowalla dataset, but higher search cost in the case of USPS. In overall, the Constant-BRC/URC and Logarithmic-BRC/URC schemes subsume PB also in terms of performance.

9. CONCLUSION

In this paper we revisited the problem of range search over data outsourced to an untrusted server. Prior techniques are either secure but exhibit prohibitive performance cost, or efficient but with unacceptable privacy leakages. We presented the first concrete framework for practical private range search building upon the definitional framework of Searchable Symmetric Encryption (SSE). We introduced a variety of schemes with realistic security/efficiency trade-offs. Our constructions utilize any secure (existing or future) SSE scheme as a black box, and appropriately convert range search to multi-keyword search. We formally defined the security of all proposed algorithms formulating the leakages, and experimentally demonstrated their practicality. In our future work, we plan to focus on the considerably harder setting of multi-dimensional (i.e., multi-attribute) range queries.

Acknowledgements

This work was partially supported by the European Commission under ICT-FP7-LEADS-318809 (Large-Scale Elastic Architecture for Data-as-a-Service) and ICT-FP7-QualiMaster-619525 (A Configurable Real-time Data Processing Infrastructure Mastering Autonomous Quality Adaptation).
10. REFERENCES


APPENDIX

A. ADDITIONAL EXPERIMENTS

Query costs at owner. To verify the practicality of the proposed approaches, we also evaluated experimentally the costs incurred at the owner during query execution. In particular, we generated random queries for ranges of size 1 to 100 over the domain $A = \{0, \ldots, 2^{20}\}$, and measured the size of each query (in bytes) and the time required for generating it. Figures 8(a) and 8(b) present the average results over 1000 executions for each query range size. For clarity, the curves corresponding to Constant and Logarithmic for the same range covering technique (i.e., BRC or URC) are grouped together in both figures, since (i) the same range covering technique leads to the same query size in both Constant and Logarithmic, and (ii) the difference in query generation time between Constant and Logarithmic is negligible, as token generation times of Constant are only slightly higher than those of Logarithmic (due to the GGM value expansion starting from the root).

As expected, Logarithmic-SRC and Logarithmic-SRC-i feature the smallest query sizes: the former always entails a single token per query, and the latter two tokens, where each token takes 24 bytes. On the other hand, the query size in both BRC- and URC-based schemes scales logarithmically with the range size. The saw-like trend of URC is due to the worst-case decomposition, whose size oscillates with the range size regardless of the randomly-selected query position. In contrast, in BRC, different query positions for the same range size lead to different numbers of tokens, which are smoothed by the averaging.

Figure 8: Query costs at owner

Figure 8(b) depicts the wall-clock time required in Trpdr to compute the tokens for all nodes that are produced by the range covering technique. We see that the curves follow a similar trend to those of Figure 8(a), which is expected since the total time is dominated by the PRF evaluations (one for each covering node/keyword). Observe that these operations are lightweight, and are typically carried out in less than 0.01 milliseconds in all our schemes. PB, on the other hand, incurs larger query sizes and higher query generation times than all our schemes, mainly due to the excessive number of cryptographic hash functions involved in its Bloom filter approach.

It is important to note that the reported costs of our methods do not depend on the datasets, or on the domain; they only depend on the position of the range in the binary tree over the domain. As such, the presented results are representative of all possible domains and datasets.