Rich Queries on Encrypted Data: Beyond Exact Matches

Sky Faber, Stanislaw Jarecki, Hugo Krawczyk(✉), Quan Nguyen, Marcel Rosu, and Michael Steiner

Yorktown, USA
hugo@ee.technion.ac.il

Abstract. We extend the searchable symmetric encryption (SSE) protocol of [Cash et al., Crypto'13] adding support for range, substring, wildcard, and phrase queries, in addition to the Boolean queries supported in the original protocol. Our techniques apply to the basic single-client scenario underlying the common SSE setting as well as to the more complex Multi-Client and Outsourced Symmetric PIR extensions of [Jarecki et al., CCS'13]. We provide performance information based on our prototype implementation, showing the practicality and scalability of our techniques to very large databases, thus extending the performance results of [Cash et al., NDSS'14] to these rich and comprehensive query types.

1 Introduction

Searchable symmetric encryption (SSE) addresses a setting where a client outsources an encrypted database (or document/file collection) to a remote server $\mathcal{E}$ such that the client, which only stores a cryptographic key, can later search the collection at $\mathcal{E}$ while hiding information about the database and queries from $\mathcal{E}$. Leakage to $\mathcal{E}$ is to be confined to well-defined forms of data-access and query patterns while preventing disclosure of explicit data and query plaintext values. SSE has been extensively studied [4–7,9,11–14,16–19,24], particularly in last years due to the popularity of clouds and data outsourcing, focusing almost exclusively on single-keyword search.

Recently, Cash et al. [5] and Pappas et al. [19] presented the first SSE solutions that go well beyond single-keyword search by supporting Boolean queries on multiple keywords in sublinear time. In particular, [4,5] build a very scalable system with demonstrated practical performance with databases containing indexes in the order of tens of billions document-keyword pairs. In this work we extend the search capabilities of the system from [5] (*referred to as the OXT protocol*) by supporting range queries (e.g., return all records of people born between two given dates), substring queries (e.g., return records with textual information containing a given pattern, say 'crypt'), wildcard queries (combining substrings with one or more single-character wildcards), and phrase queries (return records that contain the phrase "searchable encryption"). Moreover, by preserving the overall system

G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 123–145, 2015.
DOI: 10.1007/978-3-319-24177-7_7

design and optimized data structures of [4], we can run any of these new queries in combination with Boolean-search capabilities (e.g., combining a range and/or substring query with a conjunction of additional keywords/ranges/substrings) and we can do so while preserving the scalability of the system and additional properties such as support for *dynamic data*.

We also show how to extend our techniques to the more involved multi-client SSE scenarios studied by Jarecki et al. [12]. In the first scenario, denoted MC-SSE, the owner of the data, $\mathcal{D}$, outsources its data to a remote server $\mathcal{E}$ in encrypted form and later allows multiple clients to access the data via search queries and according to an authorization policy managed by $\mathcal{D}$. The system is intended to limit the information learned by clients beyond the result sets returned by authorized queries while also limiting information leakage to server $\mathcal{E}$. A second scenario, OSPIR-SSE or just OSPIR (for Outsourced Symmetric PIR), addresses the multi-client setting but adds a requirement that $\mathcal{D}$ can authorize queries to clients following a given policy, but without $\mathcal{D}$ learning the specific values being queried. That is, $\mathcal{D}$ learns minimal information needed to enforce policy, e.g., the query type or the field to which the keyword belongs, say `last name`, but not the actual last name being searched.

We present our solution for range queries in Sect. 3, showing how to reduce any such query to a *disjunction of exact keywords*, hence leveraging the Boolean query capabilities of the OXT protocol and its remarkable performance. In the OSPIR setting, we show how $\mathcal{D}$ can authorize range queries based on the total size of the queried range without learning the actual endpoints of the range. This is useful for authorization policies that limit the size of a range as a way of preventing a client from obtaining a large fraction of the database. Thus, $\mathcal{D}$ may learn that a query on a data field spans 7 days but not *which* 7 days the query is about. Achieving privacy from both $\mathcal{D}$ and $\mathcal{E}$ while ensuring that the authorized search interval does not exceed a size limit enforced by $\mathcal{D}$, is challenging. We propose solutions based on the notion of *universal tree covers* for which we present different instantiations trading performance and security depending on the SSE model that is being addressed.

The other queries we support, i.e. substrings, wildcards and phrases, are all derived from a novel technique that allows us to search on the basis of positioning information (where the data and the position information are encrypted). This technique can be used to implement any query type that can be reduced to Boolean formulas on queries of the form "are two data elements at distance Δ?". For example, in the case of substring queries, the substring is tokenized (i.e., subdivided) into a sequence of possibly-overlapping k-grams (strings of k characters) and the search is performed as a conjunction of such k-grams. However, to avoid false positives, i.e., returning documents where the k-grams appear but not at the right distances from each other, we use the relative positions of the tokens to ensure that the combined k-grams represent the searched substring. Wildcard queries are processed similarly, because t consecutive wildcard positions (i.e., positions that can be occupied by any character) can be implemented by setting the distance between the two k-grams that bracket the string of t wildcards to

$k+t$. Phrase queries are handled similarly, by storing whole words together with their encrypted positions in the text.

The crux of this technique is a homomorphic computation on encrypted position information that gives rise to a very efficient SSE protocol between client $\mathcal{C}$ and server $\mathcal{E}$ for computing relative distances between data elements while concealing this information from $\mathcal{E}$. This protocol meshes naturally with the homomorphic properties of OXT but in its general form it requires an additional round of interaction between client and server. In the SSE setting, the resulting protocol preserves most of the excellent performance of the OXT protocol (with the extra round incurring a moderate increase in query processing latency). Similar performance is achieved in the MC setting while for the OSPIR setting we resort to bilinear groups for some homomorphic operations, hence impacting performance in a more noticeable way which we are currently investigating.

We prove the security of our protocols in the SSE model of [5,7,9], and the extensions to the MC-SSE and OSPIR settings of [12], where security is defined in the real-vs-ideal model and is parametrized by a specified leakage function $\mathcal{L}(\mathsf{DB}, \mathbf{q})$. A protocol is said to be secure with leakage profile $\mathcal{L}(\mathsf{DB}, \mathbf{q})$ against adversary $\mathcal{A}$ if the actions of $\mathcal{A}$ on adversarially-chosen input DB and query set $\mathbf{q}$ can be simulated with access to the leakage information $\mathcal{L}(\mathsf{DB}, \mathbf{q})$ only (and not to DB or $\mathbf{q}$). This allows modeling and bounding the partial leakage incurred by SSE protocols. It means that even an adversary that has full information about the database and queries, or even chooses them at will, does not learn anything from the protocol execution other than what can be derived solely from the defined leakage profile. We achieve provable *adaptive* security against adversarial servers $\mathcal{E}$ and $\mathcal{D}$, and against malicious clients. Servers $\mathcal{E}$ and $\mathcal{D}$ are assumed to return correct results (e.g., server $\mathcal{E}$ returns all documents specified by the protocol) but can otherwise behave maliciously. However, in the OSPIR setting, query privacy from $\mathcal{D}$ is achieved as long as $\mathcal{D}$ does not collude with $\mathcal{E}$.

Practicality of our techniques was validated by a comprehensive implementation of: (i) the SSE protocols for range, substring and wildcard queries, and their combination with Boolean functions on exact keywords, and (ii) the OSPIR-SSE protocol for range queries. These implementations (extending those of [4,5,12]) were tested by an independent evaluator on DB's of varying size, up to 10 Terabytes with 100 million records and 25.6 billion record-keyword pairs. Performance was compared to MariaDB's (an open-source fork of MySQL) performance on the same databases running on *plaintext data and plaintext queries*. Due to the highly optimized protocols and careful I/O management, the performance of our protocols matched and often exceeded the performance of the plaintext system. These results are presented in Appendix A.

Related Work. The only work we are aware of that addresses substring search on symmetrically encrypted data is the work of Chase and Shen [8]. Their method, based on suffix trees, is very different than ours and the leakage profiles seem incomparable. This is a promising direction, although the applicability to (sublinear) search on large databases, and the integration with other query types, needs to be investigated. Its potential generalization to the multi-client or

OSPIR settings is another interesting open question. Range and Boolean queries are supported, also for the OSPIR setting, by Pappas et al. [19] (building on the work of Raykova et al [21]). Their design is similar to ours in reducing range queries to disjunctions (with similar data expansion cost) but their techniques are very different offering an alternative (and incomparable) leakage profile for the parties. The main advantages of our system are the support of the additional query types presented here and its scalability. The scalability of [19] is limited by their crucial reliance on Bloom filters that requires database sizes whose resultant Bloom filters can fit in RAM. A technique that has been suggested for resolving range queries in the SSE setting is *order-preserving encryption* (e.g., it is used in the CryptDB system [20]). However, it carries a significant intrinsic loss of privacy as the ordering of ciphertexts is visible to the holding server (and the encryption is deterministic). Range queries are supported in the multi-writer public key setting by Boneh-Waters [3] and Shi et al. [23] but at a significantly higher computational cost.

2 Preliminaries

Our work concerns itself with databases in a very general sense, including relational databases (with data arranged in "rows" and "columns"), document collections, textual data, etc. We use interchangeably the word 'document' and 'record'. We think of keywords as (attribute,value) pairs. The attribute can be structured data, such as name, age, SSN, etc., or it can refer to a textual field. We sometimes refer explicitly to the keyword's attribute but most of the time it remains implicit. We denote by m the number of distinct attributes and use $I(w)$ to denote the attribute of keyword w.

SSE Protocols and Formal Setting (following [5]). Let τ be a security parameter. A database $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^{d}$ is a list of identifier and keyword-set pairs, where $\mathsf{ind}_i \in \{0,1\}^\tau$ is a document identifier and $\mathsf{W}_i = \mathsf{DB}[\mathsf{ind}_i]$ is a list of its keywords. Let $\mathsf{W} = \bigcup_{i=1}^{d} \mathsf{W}_i$. A *query* ψ is a predicate on W_i where $\mathsf{DB}(\psi)$ is the set of identifiers of document that satisfy ψ. E.g. for a single-keyword query we have $\mathsf{DB}(w) = \{\mathsf{ind} \text{ s.t. } w \in \mathsf{DB}[\mathsf{ind}]\}$.

A *searchable symmetric encryption (SSE) scheme* Π consists of an algorithm Setup and a protocol Search fitting the following syntax. Setup takes as input a database DB and a list of document (or record) decryption keys RDK, and outputs a secret key K along with an encrypted database EDB. The search protocol Search proceeds between a *client* $\mathcal{C}$ and *server* $\mathcal{E}$, where $\mathcal{C}$ takes as input the secret key K and a query ψ and $\mathcal{E}$ takes as input EDB. At the end of the protocol, $\mathcal{C}$ outputs a set of $(\mathsf{ind}, \mathsf{rdk})$ pairs while $\mathcal{E}$ has no output. We say that an SSE scheme is *correct* for a family of queries Ψ if for all $\mathsf{DB}, \mathsf{RDK}$ and all queries $\psi \in \Psi$, for $(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(\mathsf{DB}, \mathsf{RDK})$, after running Search with client input (K, ψ) and server input EDB, the client outputs $\mathsf{DB}(\psi)$ and $\mathsf{RDK}[\mathsf{DB}(\psi)]$ where $\mathsf{RDK}[S]$ denotes $\{\mathsf{RDK}[\mathsf{ind}] \,|\, \mathsf{ind} \in S\}$. Correctness can be statistical (allowing a negligible probability of error) or computational (ensured only against computationally bounded attackers - see [5]).

Note (retrieval of matching encrypted records). Above we define the output of the SSE protocol as the set of identifiers ind pointing to the encrypted documents matching the query (together with the set of associated record decryption keys rdk). The retrieval of the document payloads, which can be done in a variety of ways, is thus decoupled from the storage and processing of the metadata which is the focus of the SSE protocols.

Multi-Client SSE Setting [12]**.** The MC-SSE formalism extends the SSE syntax by an algorithm $\mathsf{GenToken}$, which generates a search-enabling value token from the secret key K generated by the data owner $\mathcal{D}$ in Setup, and query ψ submitted by client $\mathcal{C}$. Protocol Search is then executed between server $\mathcal{E}$ and client $\mathcal{C}$ on resp. inputs EDB and token, and the protocol must assure that $\mathcal{C}$ outputs sets $\mathsf{DB}(\psi)$ and $\mathsf{RDK}[\mathsf{DB}(\psi)]$.

OSPIR SSE Setting [12]**.** An OSPIR-SSE scheme replaces the $\mathsf{GenToken}$ procedure, which in MC-SSE is executed by the data owner $\mathcal{D}$ on the cleartext client's query q, with a two-party protocol between $\mathcal{C}$ and $\mathcal{D}$ that allows $\mathcal{C}$ to compute the search-enabling token without $\mathcal{D}$ learning ψ. However, $\mathcal{D}$ should be able to enforce a query-authorization policy on $\mathcal{C}$'s query. We consider attribute-based policies, where queries are authorized based on the attributes associated to keywords in the query (e.g., a client may be authorized to run a range query on attribute 'age' but not on 'income', or perform a substring query on the 'address' field but not on the 'name' field, etc.). Later, we will consider extensions where the policy can define further constraints, e.g., the total size of an allowed interval in a range query, or the minimal size of a pattern in a substring query. An attribute-based policy for any query type is represented by a set of attribute-sequences P s.t. a query ψ involving keywords (or substrings, ranges, etc.) $(w_1, ..., w_n)$ is *allowed by policy* P if and only if the sequence of attributes $\mathsf{av}(\psi) = (I(w_1), ..., I(w_n)) \in \mathsf{P}$. Using this notation, the goal of the $\mathsf{GenToken}$ protocol is to let $\mathcal{C}$ compute token corresponding to its query on ψ only if $\mathsf{av}(\bar{w}) \in \mathsf{P}$. Note that different query types will have different entries in P. Reflecting these goals, an OSPIR-SSE scheme is a tuple $\Sigma = (\mathsf{Setup}, \mathsf{GenToken}, \mathsf{Search})$ where Setup and Search are as in MC-SSE, but $\mathsf{GenToken}$ is a protocol run by $\mathcal{C}$ on input ψ and by $\mathcal{D}$ on input (P, K), with $\mathcal{C}$ outputting token if $\mathsf{av}(\psi) \in \mathsf{P}$, or $\perp$ otherwise, and $\mathcal{D}$ outputting $\mathsf{av}(\psi)$.

3 Range Queries

Our solution for performing range queries on encrypted data reduces these queries to a disjunction of exact keywords and therefore can be integrated with SSE solutions that support such disjunctions. In particular, we use this solution to add range query support to the OXT protocol from [5,12] while keeping all the other properties of OXT intact. This includes OXT's remarkable scalability, its support for different models (SSE, MC, OSPIR), and its boolean search capability. Thus, we obtain a protocol where range queries can be run in isolation or in combination with boolean expressions on other terms, including conjunctive ranges such as $30 \leq \text{AGE} \leq 39$ and $50{,}000 \leq \text{INCOME} \leq 99{,}999$.

Range queries can be applied to any ordered set of elements; our description focuses on integer ranges for simplicity. We denote range queries with input an interval $[a, b]$, for integers $a \leq b$, by $\mathsf{RQ}(a, b)$. We refer to a and b as the *endpoints* and to the number $b - a + 1$ as the *size of the range*. Inequality queries of the form $x \geq a$ are represented by the range $[a, b]$ where b is an upper bound on all applicable values for the searched attribute; queries of the form $x \leq b$ are handled similarly.

We now describe the extensions to the OXT protocol (and its OSPIR version) for supporting range queries. Thanks to our generic reduction of range queries to disjunctions of exact keywords, our range-query presentation does not require a detailed knowledge of the OXT protocol and basic familiarity with OXT suffices (the interested reader can find more details on OXT in the above papers and also in Sect. 4.1).

Pre-Processing (Setup). For concreteness, consider a database table with an attribute (or column) A over which range queries are enabled. The values in the column are mapped to integer values between 0 and $2^t - 1$ for some number t. To support range queries on attribute A we augment the given cleartext database DB with t *virtual* columns which are populated at Setup as follows. Consider a full binary tree with $t + 1$ levels and 2^t leaves. Each node in the tree is labeled with a binary string describing the path from the root to the node: The root is labeled with the empty string, its children with strings 0 and 1, its grandchildren with 00, 01, 10, 11, and so on. A node at depth d is labeled with a string of length d, and the leaves are labeled with t-long strings that correspond to the binary representation of the integer value in that leaf, i.e. a t-bit binary representation padded with leading zeros.

Each of the t added columns correspond to a level in the tree, denoted $A'(1), A'(2), \ldots, A'(t)$ (A' indicates that this is a "virtual attribute" derived from attribute A). A record (or row) whose value for attribute A has binary representation $v_{t-1}, \ldots, v_1, v_0$ will have the string $(v_{t-1}, \ldots, v_1, v_0)$ in column $A'(t)$, the string $(v_{t-1}, \ldots v_1)$ in column $A'(t-1)$, and so on till column $A'(1)$ which will have the string v_{t-1}. Once the above plaintext columns $A'(1), \ldots, A'(t-1)$ are added to DB (note that $A'(t)$ is identical to the original attribute A), they are processed by the regular OXT pre-processing as any other original DB column, but they will be used exclusively for processing range queries.

Client Processing. To query for a range $\mathsf{RQ}(a, b)$, the client selects a set of nodes in the tree that form a *cover* of the required range, namely, a set of tree nodes for which the set of descendant leaves corresponds exactly to all elements in the range $[a, b]$ (e.g. a cover for range 3 to 9 in a tree of depth 4 will contain cover nodes 0011, 01, 100). Let $c_1, \ldots, c_\ell$ be the string representation of the nodes in the cover and assume these nodes are at depths $d_1, \ldots, d_\ell$, respectively (not all depths have to be different). The query then is formed as a *disjunction of the ℓ exact-match queries "column $A'(d_i)$ has value c_i"*, for $i = 1, \ldots, \ell$. Note that we assume that the client knows how nodes in the tree are represented; in particular it needs to know the total depth of the tree. We stress that *this reduction to a disjunctive query works with any strategy for selecting*

the cover set. This is important since different covers present different trade-offs between performance and leakage. Moreover, since the pre-processing of data is independent of the choice of cover, one can allow multiple cover strategies to co-exist to suit different leakage-performance trade-offs. Later, we will describe specific strategies for cover selection.

Interaction of Client $\mathcal{C}$ with Server $\mathcal{E}$. The search at $\mathcal{E}$ is carried exactly as in the Search phase of OXT as with any other disjunction. In particular, $\mathcal{E}$ does not need to know whether this disjunction comes from a range query.

Server $\mathcal{D}'s$ Token Generation and Authorization. For the case of single-client and multi-client) SSE, token generation and authorization work as with any disjunction in the original OXT protocol. However, in the OSPIR setting, $\mathcal{D}$ needs to authorize the query without learning the queried values. Specifically, in the scenario addressed by our implementation, authorization of range queries is based on the searched attribute (e.g., age) and the total size of the range (i.e., policy attaches to each client an upper bound on the size of a range the client is allowed to query for the given attribute). To enforce this policy, we allow $\mathcal{D}$ to learn the searched attribute and the total size of the range, i.e., $b - a + 1$, but not the actual end-point values a, b. This is accomplished as follows.

Client $\mathcal{C}$ computes a cover corresponding to his range query and maps each node in the cover to a keyword (d, c), where d is the depth of the node in the tree and c the corresponding string. It then generates a disjunction of the resultant keywords $(d_i, c_i), i = 1, \ldots, \ell$, where ℓ is the size of the cover, d_i acts as the keyword's attribute and c_i as its value. $\mathcal{C}$ provides $\mathcal{D}$ with the attributes $d_1, \ldots, d_\ell$ thus allowing $\mathcal{D}$ to provide the required search tokens to $\mathcal{C}$ as specified by the OXT protocol for the OSPIR setting [12] (OXT requires the keyword attribute to generate such token). However, before providing these tokens, $\mathcal{D}$ needs to verify that the total size of the range is under the bound that $\mathcal{C}$ is authorized for. $\mathcal{D}$ computes this size using her knowledge of the depths $d_1, \ldots, d_\ell$ by the formula $\sum_{i=1}^{\ell} 2^{t-d_i}$ which gives the number of leaves covered by these depths. This ensures the total size of the range to be under a given bound but the range can be formed of non-consecutive intervals. Importantly, this authorization approach works with any cover selection strategy used by the client.

Cover Selection. There remains one *crucial* element to take care of: Making sure that the knowledge of the cover depths $d_1, \ldots, d_\ell$ does not reveal to $\mathcal{D}$ any information other than the total size of the range. Note that the way clients select covers is essentially independent of the mechanisms for processing of range queries described above. Here we analyze some choices for cover selection. The considerations for these choices are both *performance* (e.g. size of the cover) and *privacy*. Privacy-wise the goal is to limit the leakage to server $\mathcal{E}$ and, in the OSPIR case, also to $\mathcal{D}$. In the latter case, the goal is to avoid leakage beyond the size of the range that $\mathcal{D}$ needs to learn in order to check policy compliance. These goals raise general questions regarding *canonical covers* and *minimal over-covers* which we outline below.

A natural cover selection for a given range is one that minimizes the number of nodes in the cover (hence minimizes the number of disjuncts in the search expression). Unfortunately, such cover leaks information beyond the size of a range, namely, it allows to distinguish between ranges of the same size. E.g., ranges $[0, 3]$ and $[1, 4]$ are both of size 4 but the first has a single node as its minimal cover while the latter requires 3 nodes. Clearly, if $\mathcal{C}$ uses such a cover, $\mathcal{D}$ (and possibly $\mathcal{E}$) will be able to distinguish between the two cases.

Canonical Profiles and Universal Covers. The above example raises the following question: Given that authorization allows $\mathcal{D}$ to learn the depths of nodes in a cover, is there a way of choosing a cover that only discloses the total size of the range (i.e., does not allow to distinguish between two different ranges of the same size even when the depths are disclosed)? In other words, we want a procedure that given a range produces a cover with a number of nodes and depths that is the same for any two ranges of the same size. We call such covers universal. The existence of universal covers is demonstrated by the cover that uses each leaf in the range as a singleton node in the cover. Can we have a *minimal universal cover*? Next, we answer this question in the affirmative.

Definition 1. *The* profile *of a range cover is the multi-set of integers representing the heights of the nodes in the cover. (The* height *of a tree node is its distance from a leaf, i.e., leaves have height 0, their parents height 1, and so on up to the root which has height $t - 1$.) A profile for a range of size n is* universal *if any range of size n has a cover with this profile. A* universal cover *is one whose profile is universal. A universal profile for n is* minimal *if there is a range of size n for which all covers have that profile. (For example, for $n > 2$ the all-leaves cover is universal but not minimal.)*

Definition 2 (Canonical Profile). *A profile for ranges of size n is called* canonical *if it is composed of the heights $0, 1, 2, \ldots, L-1$, where $L = \lfloor \log(n + 1) \rfloor$, plus the set of powers ('1' positions) in the binary representation of $n' = n - 2^L + 1$. A* canonical cover *is one whose profile is canonical.*

Example: for $n = 20$ we have $L = 4, n' = 5$, and the canonical profile is $\{0, 1, 2, 3, 0, 2\}$ where the last 0, 2 correspond to the binary representation 101 of 5 (note that $20 = 2^0 + 2^1 + 2^2 + 2^3 + 2^0 + 2^2$).

Lemma 1. *For every integer $n > 0$ the canonical profile of ranges of size n is universal and minimal (and the only such profile).*

The proof of this lemma is presented in the full version [10] where we also present a simple procedure to compute a canonical cover for any range. (A similar notion has been used, independently and in a different context, in [15]).

3-node Universal Over-Covers. The canonical cover has the important property of not leaking any information to $\mathcal{D}$ beyond the size of the range (that $\mathcal{D}$ needs to learn anyway to authorize a query). However, the number of nodes in a canonical cover can leak information on the range size to server $\mathcal{E}$ (assuming

that $\mathcal{E}$ knows that a given disjunction corresponds to a range query). Another drawback is that canonical covers may include $2 \log n$ nodes. Ideally, we would like to use covers with a small and fixed number of nodes that also have *universal profiles*, i.e., any two ranges of a given size will always be represented by covers with the same depths profile. While we show this to be impossible for exact covers, we obtain covers with the above properties by allowing false-positives, i.e., covers that may include elements outside the requested range, hence we call them *over-covers*. In the full version [10] we instantiate this approach for 3-node universal over-covers.

4 Substring Queries

Our substring-search capable SSE scheme is based on the conjunctive-search SSE protocol OXT of [5], and it extends that protocol as follows: Whereas the OXT scheme of [5] supported efficient retrieval of records containing several required keywords at once (i.e. satisfying a *conjunction* of several keyword-equality search terms), our extension supports efficient retrieval of records containing the required keywords *at required relative positions to one another*. This extension of conjunctive search with positional distance criteria allows us to handle several query types common in text-based information retrieval. To simplify the description, and using the notation from Sect. 2, consider a database $\mathsf{DB} = (\mathsf{ind}_i, T_i)$ containing records with just one free text attribute, i.e. where each record T_i is a text string. We support the following types of queries q:

Substring Query. Here q is a text string, and $\mathsf{DB}(q)$ returns all ind_i s.t. T_i contains q as a substring.

Wildcard Query. Here q is a text string which can contain wildcard characters $'?'$ (matching any single character), and $\mathsf{DB}(q)$ returns all ind_i s.t. T_i contains a substring q' s.t. for all j from 1 to $|q|$, $q_j =' ?' \ \vee \ q_j = q'_j$, where q_j and q'_j denote j-th characters in strings q and q'. If the query should match only prefixes (suffixes) of T_i, the query can be prefixed (suffixed) with a $'\hat{}'$ ($'\$'$).

Phrase Query. Here q is a sequence of words, i.e. text strings, $q = (q^1, \ldots, q^l)$, where each q^i can equal to a wildcard character $'?'$. Records T_i in DB are also represented as sequences of words, $T_i = (T_i^1, \ldots, T_i^n)$. $\mathsf{DB}(q)$ returns all ind_i s.t. for some k and for all j from 1 to l, it holds that $q^j =' ?' \ \vee \ q^j = T_i^{k+j}$. (Note that phrase queries allow a match of a single wildcard with a whole word of any size, while in a wildcard query a single wildcard can match only a single character).

All these query types utilize the same crypto machinery that we describe next for the substring case. In Sect. 4.2 we explain briefly how to adapt the techniques to these queries too.

4.1 Basic SSE Substring Search

Here we present protocol SUB-SSE-OXT that supports substring search in the basic SSE model (i.e., a single client $\mathcal{C}$ outsources its encrypted database to server $\mathcal{E}$) and where the query consists of a single substring. This simpler case allows us to explain and highlight the basic ideas that we also use for addressing the general case of boolean expressions that admit substrings as the expression terms as well as for extending these solutions to the more involved MC and OSPIR settings.

Figure 1 describes the protocol where shadowed text highlights the changes with respect to the original OXT protocol from [5] for resolving conjunctive queries in the SSE model (the reader can visualize the underlying OXT protocol by omitting the shadowed text). We first explain the basic rationale and functioning of the conjunctive-search OXT protocol, and then we explain how we extend it by imposing additional constraints on *relative positions* of the searched terms, and how this translates into support for substring-search SSE.

The Conjunctive SSE Scheme OXT. Let $q = (w_1, \ldots, w_n)$ be a conjunctive query where $\mathsf{DB}(q) = \cap_{i=1}^{n} \mathsf{DB}(w_i)$. Let F_G be a Pseudorandom Function (PRF) with key K_G. (This PRF will map onto a cyclic group G, hence the name). Let the setup algorithm create as metadata a set of (keyed) hashes XSet, named for "cross-check set", containing the hash values $\mathsf{xtag}_{w,\mathsf{ind}} = F_G(K_G, (w, \mathsf{ind}))$ for all keywords $w \in \mathsf{W}$ and records $\mathsf{ind} \in \mathsf{DB}(w)$. Let the setup also create the metadata needed to quickly retrieve the set of record indexes $\mathsf{DB}(w)$ matching any given *single* keyword $w \in \mathsf{W}$. The OXT protocol is based on a simple conjunctive *plaintext* search algorithm which identifies all records corresponding to a conjunctive query $q = (w_1, \ldots, w_n)$ as follows: It first identifies the set of indexes $\mathsf{DB}(w_1)$ satisfying the first term w_1, called an *s-term*, and then for each $\mathsf{ind} \in \mathsf{DB}(w_1)$ it returns ind as part of $\mathsf{DB}(q)$ if and only if hash value $\mathsf{xtag}_{w_i,\mathsf{ind}} = F_G(K_G, (w_i, \mathsf{ind}))$ is in XSet for all *x-terms* (i.e. "cross-check terms") $w_2, \ldots, w_n$. If group G is sufficiently large then except for negligible collision probability, if $\mathsf{xtag}_{w_i,\mathsf{ind}} \in \mathsf{XSet}$ for $i \geq 2$ then $\mathsf{ind} \in \cap_{i=2}^{n} \mathsf{DB}(w_i)$, and since ind was taken from $\mathsf{DB}(w_1)$ it follows that $\mathsf{ind} \in \mathsf{DB}(q)$. Since this algorithm runs in $O(|\mathsf{DB}(w_1)|)$ time w_1 should be chosen as the least frequent keyword in q.

To implement the above protocol over *encrypted* data the OXT protocol modifies it in three ways: First, the metadata supporting retrieval of $\mathsf{DB}(w)$ is implemented using single-keyword SSE techniques, specifically the *Oblivious Storage* data structure TSet [4,5], named for "tuples set", which reveals to server $\mathcal{E}$ only the total number of keyword occurrences in the database, $\sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$, but hides all other information about individual sets $\mathsf{DB}(w)$ except those actually retrieved during search. (A TSet can be implemented very efficiently as a hash table using PRF F whose key K_T is held by client $\mathcal{C}$, see [4,5]). Secondly, the information stored for each w in the TSet datastructure, denoted $\mathsf{TSet}(w)$, which $\mathcal{E}$ can recover from TSet given $F(K_T, w)$, is not the plaintext set of indexes $\mathsf{DB}(w)$ but the encrypted version of these indexes using a special-purpose encryption. Namely, a tuple corresponding to the c-th index ind_c in $\mathsf{DB}(w)$ (arbitrarily ordered) contains value $y_c = F_p(K_I, \mathsf{ind}_c) \cdot F_p(K_z, c)^{-1}$, an

element in a prime-order group Z_p where F_p is a PRF onto Z_p, and K_I, K_z are two PRF keys where K_I is global and K_z is specific to keyword w. This encryption enables fast secure computation of hash $\mathsf{xtag}_{w_i,\mathsf{ind}_c}$ between client $\mathcal{C}$ and server $\mathcal{E}$, where $\mathcal{E}$ holds ciphertext $y_c = F_p(K_I, \mathsf{ind}_c) \cdot F_p(K_z, c)^{-1}$ of c-th index ind_c taken from $\mathsf{TSet}(w_1)$ and $\mathcal{C}$ holds keyword w_i and keys K_I, K_z. Let $F_G(K_G, (w, \mathsf{ind})) = g^{F_p(K_X,w)\cdot F_p(K_I,\mathsf{ind})}$ where g generates group G and $K_G = (K_X, K_I)$ where K_X is a PRF key. $\mathcal{C}$ then sends to $\mathcal{E}$:

$$\mathsf{xtoken}[c, i] = g^{F_p(K_X,w_i)\cdot F_p(K_z,c)}$$

for $i = 2, \ldots, h$ and $c = 1, \ldots, |\mathsf{TSet}(w_1)|$, and $\mathcal{E}$ computes $F_G(K_G, (w_i, \mathsf{ind}_c))$ for each c, i as:

$$(\mathsf{xtoken}[c, i])^{y_c} = (\mathsf{xtoken}[c, i])^{F_p(K_I,\mathsf{ind}_c)\cdot F_p(K_z,c)^{-1}}$$

Since K_z is specific to w_1 mask $z_c = F_p(K_z, c)$ applied to ind_c in y_c is a one-time pad, hence this protocol reveals only the intended values $F_G(K_G, (w_i, \mathsf{ind}_c))$ for all $\mathsf{ind}_c \in \mathsf{DB}(w_1)$ and $w_2, \ldots, w_n$.

Extending OXT to Substring SSE. The basic idea for supporting substring search is first to represent a substring query as a conjunction of k-grams (strings of length k) at given relative distances from each other (e.g., a substring query *'yptosys'* can be represented as a conjunction of a 3-gram *'tos'* and 3-grams *'ypt'* and *'sys'* at relative distances -2 and 2 from the first 3-gram, respectively), and then to extend the conjunctive search protocol OXT of [5] so that it verifies not only whether the conjunctive terms all occur within the same document, but also that they occur at positions whose relative distances are specified by the query terms. We call representation of a substring q as a set of k-grams with relative distances a *tokenization* of q. We denote the *tokenizer* algorithm as T, and we denote its results as $T(q) = (\mathsf{kg}_1, (\Delta_2, \mathsf{kg}_2), \ldots, (\Delta_\mathsf{h}, \mathsf{kg}_\mathsf{h}))$ where Δ_i are any non-zero integer values, including negatives, e.g. T('*yptosys*') can output ('*tos*',(-2,'*ypt*'),(2,'*sys*')), but many other tokenizations of the same string are possible. We call k-gram kg_1 an *s-gram* and the remaining k-grams *x-grams*, in parallel to the s-term and x-term terminology of OXT, and as in OXT the s-gram should be chosen as the least frequent k-gram in the tokenization of q. Let KG be a list of k-grams which occur in DB. Let $\mathsf{DB}(\mathsf{kg})$ be the set of $(\mathsf{ind}, \mathsf{pos})$ pairs s.t. $\mathsf{DB}[\mathsf{ind}]$ contains k-gram kg at position pos, and let $\mathsf{DB}(\mathsf{ind}, \mathsf{kg})$ be the set of pos's s.t. $(\mathsf{ind}, \mathsf{pos}) \in \mathsf{DB}(\mathsf{kg})$.

The basic idea of the above conjuctive-search protocol to handling substrings is that the hashes xtag inserted into the XSet will use PRF F_G applied to a *triple* $(\mathsf{kg}, \mathsf{ind}, \mathsf{pos})$ for each $\mathsf{kg} \in \mathsf{KG}$ and $(\mathsf{ind}, \mathsf{pos}) \in \mathsf{DB}(\mathsf{kg})$, and when processing search query q where $T(q) = (\mathsf{kg}_1, (\Delta_2, \mathsf{kg}_2), \ldots, (\Delta_\mathsf{h}, \mathsf{kg}_\mathsf{h}))$, server $\mathcal{E}$ will return (encrypted) index ind corresponding to some $(\mathsf{ind}_c, \mathsf{pos}_c)$ *pair* in $\mathsf{DB}(\mathsf{kg}_1)$ if and only if

$$F_G(K_G, (\mathsf{kg}_i, \mathsf{ind}_c, \mathsf{pos}_c + \Delta_i)) \in \mathsf{XSet} \text{ for } i = 2, \ldots, \mathsf{h}$$

Setup(DB, RDK)

- Select keys K_S, K_T for PRF F_τ and K_I, K_X for PRF F_p, and parse DB as $(\mathsf{ind}_i, \mathsf{pos}_i, \mathsf{kg}_i)_{i=1}^{d}$. (PRF F_τ maps onto $\{0,1\}^\tau$ and F_p onto Z_p.)
- Initialize $\mathbf{T}$ to an empty array and XSet to an empty set. For each k-gram $\mathsf{kg} \in \mathsf{KG}$ do the following:
  - Set $\mathsf{strap} \leftarrow F_\tau(K_S, \mathsf{kg})$, $(K_z, K_e, K_u) \leftarrow (F_\tau(\mathsf{strap}, 1), F_\tau(\mathsf{strap}, 2), F_\tau(\mathsf{strap}, 3))$.
  - For $c = 1, \ldots, |\mathsf{DB}(\mathsf{kg})|$, for $(\mathsf{ind}, \mathsf{pos})$ a c-th tuple in $\mathsf{DB}(\mathsf{kg})$ (randomly permuted) do:
    - Set $\mathsf{rdk} \leftarrow \mathsf{RDK}(\mathsf{ind})$, $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, (\mathsf{ind}|\mathsf{rdk}))$, $\mathsf{xind} \leftarrow F_p(K_I, \mathsf{ind})$.
    - Set $\mathsf{xtag} \leftarrow g^{F_p(K_X, \mathsf{kg}) \cdot \mathsf{xind}^{\mathsf{pos}}}$ and add xtag to XSet.
    - Set $z \leftarrow F_p(K_z, c)$, $u \leftarrow F_p(K_u, c)$, $y \leftarrow \mathsf{xind} \cdot z^{-1}$, $v \leftarrow \mathsf{xind}^{\mathsf{pos}} \cdot u^{-1}$.
    - Append (e, y, v) to $\mathbf{T}[\mathsf{kg}]$.
- Set $\mathsf{TSet} \leftarrow \mathsf{TSetSetup}(\mathbf{T}, \langle F_\tau \rangle, K_T)$. Output $K = (K_S, K_X, K_T)$ and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.

Search protocol

Client $\mathcal{C}$, on input $K = (K_S, K_X, K_T)$ defined above and query q s.t. $T(q) = (\mathsf{kg}_1, (\Delta_2, \mathsf{kg}_2), \ldots, (\Delta_\mathsf{h}, \mathsf{kg}_\mathsf{h}))$:

- Set $\mathsf{stag} \leftarrow F_\tau(K_T, \mathsf{kg}_1)$, $\mathsf{strap} \leftarrow F_\tau(K_S, \mathsf{kg}_1)$.
- $(K_z, K_e, K_u) \leftarrow (F_\tau(\mathsf{strap}, 1), F_\tau(\mathsf{strap}, 2), F_\tau(\mathsf{strap}, 3))$, and $\{\mathsf{xtrap}_i \leftarrow g^{F_p(K_X, \mathsf{kg}_i)}\}_{i=2}^{\mathsf{h}}$.
- Send $(\mathsf{stag}, \Delta_2, \ldots, \Delta_\mathsf{h})$ to $\mathcal{E}$, and for $c = 1, 2, \ldots$, until $\mathcal{E}$ sends stop, do the following:
  - Set $z_c \leftarrow F_p(K_z, c)$, $u_c \leftarrow F_p(K_u, c)$, and $\{\mathsf{xtoken}[c, i] \leftarrow (\mathsf{xtrap}_i)^{((z_c)^{\Delta_i} \cdot (u_c))}\}_{i=2}^{\mathsf{h}}$.
  - Send $\mathsf{xtoken}[c] = (\mathsf{xtoken}[c, 2], \ldots, \mathsf{xtoken}[c, \mathsf{h}])$ to $\mathcal{E}$.

Server $\mathcal{E}$, on input $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$, responds with a set ESet formed as follows:

- On message $(\mathsf{stag}, \Delta_2, \ldots, \Delta_n)$ from $\mathcal{C}$, retrieve $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$ from TSet.
- For $c = 1, ..., |\mathbf{t}|$, retrieve c-th tuple (e, y, v) in $\mathbf{t}$.
- On $\mathsf{xtoken}[c]$ from $\mathcal{C}$, add e to ESet if $\forall i = 2, \ldots, \mathsf{h} : (\mathsf{xtoken}[c, i])^{(y^{\Delta_i} \cdot v)} \in \mathsf{XSet}$. When $c = |\mathbf{t}|$ send stop to $\mathcal{C}$.

Client $\mathcal{C}$ computes $(\mathsf{ind}|\mathsf{rdk}) \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ for each e in ESet and adds $(\mathsf{ind}, \mathsf{rdk})$ to its output.

Fig. 1. SUB-SSE-OXT: SSE Protocol for Substring Search (shadowed text indicates additions to the basic OXT protocol for supporting substring queries)

To support this modified search over encrypted data the setup procedure $\mathsf{Setup}(\mathsf{DB}, \mathsf{RDK})$ forms EDB as a pair of data structures TSet and XSet as in OXT, except that keywords are replaced by k-grams and both the encrypted tuples in TSet and the hashes xtag in XSet will be modified by the position-related information as follows. First, the tuple corresponding to the c-th (index,position) pair $(\mathsf{ind}_c, \mathsf{pos}_c)$ in $\mathsf{DB}(\mathsf{kg})$ will contain value $y_c = F_p(K_I, \mathsf{ind}_c) \cdot F_p(K_z, c)^{-1}$ together with a new position-related value $v_c = F_p(K_I, \mathsf{ind}_c)^{\mathsf{pos}_c} \cdot F_p(K_u, c)^{-1}$, where K_z, K_u are independent PRF keys specific to kg. Secondly, XSet will contain values computed as:

$$F_G((K_X, K_I), (\mathsf{kg}, \mathsf{ind}, \mathsf{pos})) = g^{F_p(K_X, \mathsf{kg}) \cdot F_p(K_I, \mathsf{ind})^{\mathsf{pos}}} \tag{1}$$

In the Search protocol, client $\mathcal{C}$ will tokenize its query q as $T(q) = (\mathsf{kg}_1, (\Delta_2, \mathsf{kg}_2), \ldots, (\Delta_\mathsf{h}, \mathsf{kg}_\mathsf{h}))$, send $\mathsf{stag}_{\mathsf{kg}_1} = F_T(K_T, \mathsf{kg}_1)$ to server $\mathcal{E}$, who uses it to retrieve $\mathsf{TSet}(\mathsf{kg}_1)$ from TSet, send the position-shift vectors $(\Delta_2, \ldots, \Delta_\mathsf{h})$ to $\mathcal{E}$, and then, in order for $\mathcal{E}$ to compute $F_G(K_G, (\mathsf{kg}_i, \mathsf{ind}_c, \mathsf{pos}_c + \Delta_i))$ for all c, i pairs, client $\mathcal{C}$ sends to $\mathcal{E}$:

$$\mathsf{xtoken}[c, i] = g^{F_p(K_X, \mathsf{kg}_i) \cdot (F_p(K_z, c))^{\Delta_i} \cdot F_p(K_u, c)}$$

which lets $\mathcal{E}$ compute $F_G(\mathsf{kg}_i, \mathsf{ind}_c, \mathsf{pos}_c + \Delta_i)$ as $(\mathsf{xtoken}[c, i])$ exponentiated to power $(y_c)^{\Delta_i} \cdot v_c$ for (y_c, v_c) in the c-th tuple in $\mathsf{TSet}(\mathsf{kg}_1)$, which computes correctly because

$$y_c^{\Delta_i} \cdot v_c = F_p(K_I, \mathsf{ind}_c)^{\Delta_i + \mathsf{pos}_c} \cdot F_p(K_z, c)^{-\Delta_i} \cdot F_p(K_u, c)^{-1}$$

4.2 Wildcards and Phrase Queries

Any sequence of single character wildcards within regular substring queries can be handled by changing tokenization to allow gaps in the query string covered by the computed tokens, e.g. $T('ypt??yst')$ would output $('ypt', (5,' yst'))$.

In addition to support *wildcard queries* matching prefixes and/or suffixes, we add special "anchor" tokens at the beginning ($'\hat{}'$) and end ($'\$'$) of every record to mark the text boundaries. These anchors are then added during tokenization. This allows searching for substrings at fixed positions within a record. For these queries $T('ypt??yst')$ would output $('\hat{}yp', (1,' ypt'), (6,' yst'), (7,' st\$'))$.

Still, this simple change limits us to queries which contain k consecutive characters in-between every substring of wildcards. However, we can remove this restriction if we add to the XSet all unigrams (i.e. $k = 1$) occurring in a text in addition to the original k-grams.

Adding support for phrase queries is another simple change to the way we parse DB. Instead of parsing by (k-gram, position) pairs, we parse each record by (word, position). Tokenization of q then becomes splitting q into its component words and relative position of each word to the s-term word. As with substrings, wildcards in q result in a gap in the returned Δ's.

4.3 Substring Protocol Extensions

Due to space limitations (this material is available from the authors upon request), we only discuss briefly the extensions to the above SUB-SSE-OXT protocol needed to support richer functionality as well as the MC and OSPIR settings. A first extension extends the single-substring of SUB-SSE-OXT to any Boolean query where atomic terms can be formed by any number of substring search terms and/or exact keyword terms. Moreover, the user can specify as an s-term either one of the exact keyword terms or a k-gram in one of the substring terms. We call the resulting protocol MIXED-SSE-OXT, so named because it freely *mixes* substring and exact keyword search terms. The ability to handle Boolean formulas on exact keywords together with substring terms comes

from the similarities between substring-handling SUB-SSE-OXT and Boolean-formula-handling OXT of [5]. However, one significant adjustment needed to put the two together is to disassociate the position-related information v_c in the tuples in $\mathsf{TSet}(\mathsf{kg})$ from the index-related information y_c in these tuples. This is because when all k-gram terms are x-terms (as would be the case e.g. when an exact keyword is chosen as an s-term) then $\mathcal{E}$ must identify the position-related information pertaining to a particular $(\mathsf{kg}, \mathsf{ind})$ pair given the $(\mathsf{kg}, \mathsf{ind})$-related xtoken value. Our MIXED-SSE-OXT protocol supports this by adding another oblivious TSet-like datastructure which uses $\mathsf{xtag}_{\mathsf{kg,ind}}$ to retrieve the position-related information, i.e. the v_c's, for all $\mathsf{pos} \in \mathsf{DB}(\mathsf{ind}, \mathsf{kg})$.

A second extension generalizes the SUB-SSE-OXT protocol to the OSPIR setting [12] where $\mathcal{D}$ can *obliviously* enable third-party clients $\mathcal{C}$ to compute the search-enabling tokens (see Sect. 2). The main ingredient in this extension is the usage of Oblivious PRF (OPRF) evaluation for several PRF functions used in MIXED-SSE-OXT for computing search tokens. Another important component is a novel protocol which securely computes the $\mathsf{xtag}_{\mathsf{kg,ind,pos}}$ values given these obliviously-generated trapdoors, in a way which avoids leaking any partial-match information to $\mathcal{C}$. This protocol uses bilinear maps which results in a significant slowdown compared to the MIXED-SSE-OXT in the Client-Server setting. Fortunately, for the Multi-Client (MC) setting where the third-party clients' queries are not hidden from the database owner $\mathcal{D}$, we can simplify this xtag-computation protocol, in particular eliminating the usage of bilinear maps, and making the resulting protocol MIXED-MC-OXT almost equal in cost to the underlying MIXED-SSE-OXT protocol.

5 Security Analysis

Privacy of an SSE scheme, in the SSE, Multi-Client, or OSPIR settings, is quantified by a *leakage profile* $\mathcal{L}$, which is a function of the database DB and the sequence of client's queries $\mathbf{q}$. We call an SSE scheme $\mathcal{L}$*-semantically-secure* against party P (which can be $\mathcal{C}$, $\mathcal{E}$, or $\mathcal{D}$) if for all DB and $\mathbf{q}$, the entirety of P's view of an execution of the SSE scheme on database DB and $\mathcal{C}$'s sequence of queries $\mathbf{q}$ is efficiently *simulatable* given only $\mathcal{L}(\mathsf{DB}, \mathbf{q})$. We say that the scheme is *adaptively* secure if the queries in $\mathbf{q}$ can be set adaptively by the adversary based on their current view of the protocol execution. An efficient simulation of a party's view in the protocol means that everything that the protocol exposes to this party carries no more information than what is revealed by the $\mathcal{L}(\mathsf{DB}, \mathbf{q})$ function. Therefore specification of the $\mathcal{L}$ function fully characterizes the privacy quality of the solution: What it reveals about data DB and queries $\mathbf{q}$, and thus also what it hides. (See [5,12] for a more formal exposition.)

Security of Range Queries. Below we state the security of the range query protocol for stand-alone range queries and we informally comment on the case of range queries that are parts of composite (e.g., Boolean) queries. See full version [10] for a more complete security treatment of range queries. We consider adaptive security against honest-but-curious and non-colluding servers $\mathcal{E}, \mathcal{D}$, and

against fully malicious clients. For query $q^j = \mathsf{RQ}(a^j, b^j)$, let $((d_1^j, c_1^j), \ldots, (d_t^j, c_t^j))$ be the tree cover of interval $[a^j, b^j]$ and let $w_i^j = (d_i^j, c_i^j)$. We define three leakage functions for $\mathcal{D}, \mathcal{E}, \mathcal{C}$, respectively:

- $\mathcal{L}_{\mathcal{D}}(\mathsf{DB}, (q^1, \ldots, q^m))$ includes the query type ("range" in this case), the attribute to which q^j pertains, and the size of the range $b^j - a^j + 1$, for each q^j.
- $\mathcal{L}_{\mathcal{E}}(\mathsf{DB}, (q^1, \ldots, q^m)) = \mathcal{L}_{\mathsf{OXT}}(\mathsf{DB}, (w_1^1, \ldots, w_t^m))$ where the latter function represents the leakage to server $\mathcal{E}$ in the OXT protocol for a query series that includes all w_i^j's. By the analysis of [5], this leakage contains the TSet leakage (which in our TSet implementation is just the total number of document-keyword pairs in DB), the sequence $\{(|\mathsf{DB}(w_i^j)| : (i,j) = (1,1), \ldots, (t,m)\}$, i.e., the number of elements in each $\mathsf{DB}(w_i^j)$, and the result set returned by the query (in the form of encrypted records).
- $\mathcal{L}_{\mathcal{C}}(\mathsf{DB}, (q^1, \ldots, q^m)) = \emptyset$.

Theorem 1. *The range protocol from Sect. 3 is secure in the OSPIR model with respect to $\mathcal{D}, \mathcal{E}, \mathcal{C}$ with leakage profiles $\mathcal{L}_{\mathcal{D}}, \mathcal{L}_{\mathcal{E}}, \mathcal{L}_{\mathcal{C}}$, respectively.*

The leakage functions for $\mathcal{D}$ and $\mathcal{C}$ are as good as possible: $\mathcal{D}$ only learns the information needed to enforce authorization, namely the attribute and size of the range, while there is no leakage at all to the client. The only non-trivial leakage is $\mathcal{E}$'s which leaks the number of documents matching each disjunct or, equivalently, the size of each sub-range in the range cover. The leakage to $\mathcal{D}$ remains the same also when the range query is part of a composite query. For the client this is also the case except that when the range query is the s-term of a Boolean expression, the client also learns an upper bound on the sizes $|\mathsf{DB}(w_i^j)|$ for all i, j. For $\mathcal{E}$, a composite query having range as its s-term is equivalent to tm separate expressions w_i^j as in [5] (with reduced leakage due to disjoint s-terms), and if the range term is an x-term in a composite query then w_i^j's leak the same as if they were x-terms in a conjunction.

Security of Substring Queries. Here we state the security of protocol SUB-SSE-OXT against server $\mathcal{E}$. Our security arguments are based on the following assumptions: the T-set implementation is secure against adaptive adversaries [4,5]; F_p and F_τ are secure pseudorandom functions; the hash function H is modeled as a random oracle; and the q-DDH assumption [1] holds in G.[1]

The security analysis follows the corresponding argument in [5] adapting the leakage profile to the substring case.

Leakage to Server $\mathcal{E}$. We represent a sequence of Q non-adaptive substring queries by $\mathbf{q} = (\mathbf{s}, \mathbf{x}, \mathbf{\Delta})$ s.t. $(\mathbf{s}[i], (\mathbf{x}[i], \mathbf{\Delta}[i]))$ is the tokenization $T(\mathbf{q}[i])$ of the i-th substring query $\mathbf{q}[i]$, where $\mathbf{s}[i], \mathbf{x}[i]$ are k-grams, and $\mathbf{\Delta}[i]$ is an integer between $-k+1$ and $k-1$. For notation simplicity we assume that vector $\mathbf{q}$ does not contain repeated queries, although $\mathcal{E}$ would learn that a repeated query has been made.

[1] Our extension to the OSPIR model relies on the One-More Gap Diffie-Hellman assumption and the linear DH assumption [2,22] on bilinear groups.

Function $\mathcal{L}_{\mathcal{E}}(\mathsf{DB}, \mathbf{q})$ which specifies leakage to $\mathcal{E}$ outputs $(N, \bar{\mathbf{s}}, \mathsf{SP}, \mathsf{RP}, \mathsf{DP}, \mathsf{IP})$, defined as follows:

- The $(N, \bar{\mathbf{s}}, \mathsf{SP}, \mathsf{RP})$ part of this leakage is exactly the same as in the conjunctive SSE protocol SSE-OXT of [5] on which our substring-search SUB-SSE-OXT protocol is based. $N = \sum_{i=1}^{d} |\mathsf{W}_i|$ is the total number of appearances of all k-grams in all the documents, and it is revealed simply by the size of the EDB metadata. $\bar{\mathbf{s}} \in [m]^Q$ is the *equality pattern* of $\mathbf{s} \in \mathsf{KG}^Q$ indicating which queries have the equal s-terms. For example, if $\mathbf{s} = (abc, abc, xyz, pqr, abc, pqr, def, xyz, pqr)$ then $\bar{\mathbf{s}} = (1, 1, 2, 3, 1, 3, 4, 2, 3)$. SP is the *s-term support size* which is the number of occurrences of the s-term k-gram in the database, i.e. $\mathsf{SP}[i] = |\mathsf{DB}(\mathbf{s}[i])|$. Finally, RP is the *results pattern*, i.e. $\mathsf{RP}[i]$ is the set of $(\mathsf{ind}, \mathsf{pos})$ pairs where ind is an identifier of document which matches the query q, and pos is a position of the s-term k-gram $\mathbf{s}[i]$ in that document.
- DP is the *Delta pattern* $\boldsymbol{\Delta}[i]$ of the queries, i.e. the shifts between k-grams in a query which result from the tokenization of the queries.
- IP is the *conditional intersection pattern*, which is a Q by Q table IP defined as follows: $\mathsf{IP}[i, j] = \emptyset$ if $i = j$ or $\mathbf{x}[i] \neq \mathbf{x}[j]$. Otherwise, $\mathsf{IP}[i, j]$ is the set of all triples $(\mathsf{ind}, \mathsf{pos}, \mathsf{pos}')$ (possibly empty) s.t. $(\mathsf{ind}, \mathsf{pos}) \in \mathsf{DB}(\mathbf{s}[i])$, $(\mathsf{ind}, \mathsf{pos}') \in \mathsf{DB}(\mathbf{s}[j])$, and $\mathsf{pos}' = \mathsf{pos} + (\boldsymbol{\Delta}[i] - \boldsymbol{\Delta}[j])$.

Understanding Leakage Components. Parameter N is the size of the meta-data, and leaking such a bound is unavoidable. The equality pattern $\bar{\mathbf{s}}$, which leaks repetitions in the s-term k-gram of different substring queries, and the s-term support size SP, which leaks the total number of occurrences of this s-term in the database, are both a consequence of the optimized search that singles out the s-term in the query, which we adopt from the conjunctive SSE search solution of [5]. RP is the result of the query and therefore no real leakage in the context of SSE. Note also that the RP over-estimates the information $\mathcal{E}$ observes, because $\mathcal{E}$ observes only a pointer to the encrypted document, and a pointer to the encrypted tuple storing a unique $(\mathsf{ind}, \mathsf{pos})$ pair, but not the pair $(\mathsf{ind}, \mathsf{pos})$ itself. DP reflects the fact that our protocols leak the relative shifts Δ between k-grams which result from tokenization of the searched string. If tokenization was canonical, and divided a substring into k-grams based only on the substring length, the shifts Δ would reveal only the substring length. (Otherwise, see below for how Δ's can be hidden from $\mathcal{E}$).

The IP component is the most subtle. It is a consequence of the fact that when processing the $\mathbf{q}[i]$ query $\mathcal{E}$ computes the (pseudo)random function $F_G(\mathbf{x}[i], \mathsf{ind}, \mathsf{pos} + \boldsymbol{\Delta}[i])$ for all $(\mathsf{ind}, \mathsf{pos}) \in \mathsf{DB}(\mathbf{s}[i])$, and hence can see collisions in it. Consequently, if two queries $\mathbf{q}[i]$ and $\mathbf{q}[j]$ have the same x-gram then for any document ind which contains the s-grams $\mathbf{s}[i]$ and $\mathbf{s}[j]$ in positions, respectively, pos and $\mathsf{pos}' = \mathsf{pos} + (\boldsymbol{\Delta}[i] - \boldsymbol{\Delta}[j])$, server $\mathcal{E}$ can observe a collision in F_G and triple $(\mathsf{ind}, \mathsf{pos}, \mathsf{pos}')$ will be included in the IP leakage. Note, however, that $\mathsf{IP}[i, j]$ defined above overstates this leakage, because $\mathcal{E}$ does not learn the $\mathsf{ind}, pos_i, \mathsf{pos}_j$ values themselves, but only establishes a link between two *encrypted* tuples, the one containing $(\mathsf{ind}, \mathsf{pos})$ in $\mathsf{TSet}(\mathbf{s}[i])$ and the one containing $(\mathsf{ind}, \mathsf{pos}')$ in $\mathsf{TSet}(\mathbf{s}[j])$. To visualize the type of queries which will trigger

this leakage, take $k = 3$, $\mathbf{q}[i]$ = *MOTHER*, $\mathbf{q}[j]$ = *OTHER*, and let $\mathbf{q}[i]$ and $\mathbf{q}[j]$ tokenize with a common x-gram, e.g. $T(\mathbf{q}[i]) = (\text{MOT}, (\text{HER}, 3))$ and $T(\mathbf{q}[j]) = (\text{OTH}, (\text{HER}, 2))$. The $\mathsf{IP}[i, j]$ leakage will contain tuple $(\mathsf{ind}, \mathsf{pos}, \mathsf{pos}')$ for $\mathsf{pos}' = \mathsf{pos} + (\boldsymbol{\Delta}[i] - \boldsymbol{\Delta}[j]) = \mathsf{pos} + 1$ iff record $\mathsf{DB}[\mathsf{ind}]$ contains 3-gram $\mathbf{s}[i] = \text{MOT}$ at position pos and 3-gram $\mathbf{s}[j] = \text{OTH}$ at position $\mathsf{pos} + 1$, i.e. iff it contains substring MOTH.

Theorem 2. *Protocol SUB-SSE-OXT (restricted to substrings which tokenize into two k-grams) is adaptively $\mathcal{L}_{\mathcal{E}}$-semantically-secure against malicious server $\mathcal{E}$, assuming the security of the PRF's, the encryption scheme* Enc*, and the* TSet *scheme, the random oracle model for hash functions, and the q-DDH assumption on the group G of prime order.*

The proof of Theorem 2 is presented in the full version [10].

A Implementation and Performance

Here we provide testing and performance information for our prototype implementation of the range and SUB-SSE-OXT protocols described in Sects. 3 and 4.1. The results confirm the scalability of our solutions to very large databases and complex queries. The prototype is an extension of the OXT implementation of [4]. Both the description of the changes and performance information are limited, to the extent possible, to the protocols introduced in this paper. An extensive evaluation of the prototype is outside of the scope of this paper as it would be highly dependent on previous work.

Prototype Summary. The three components of our system are the preprocessor, the server, and the client. The preprocessor generates the encrypted database from the cleartext data. The client, which implements a representative set of SQL commands, 'encrypts' end-user requests and 'decrypts' server responses. The server uses the encrypted database to answer client SELECT-type queries or expands the encrypted database on UPDATE, INSERT, and (even) DELETE queries [4].

To support range queries (see Sect. 3) the Boolean-query OXT prototype was augmented with generation of range-specific TSet's at pre-processing, and with range-specific authorization and range-cover computation at the client. Support for substring and wildcard queries required redesigning pre-processing to take into account the k-gram position information, adding support for 'k-gram'-based record tokenization to the client, and changing the Search protocol to support *position-enhanced* computation (see Sect. 4) and authorization. A few other changes were necessary in order to continue handling UPDATE, INSERT and DELETE queries. These extensions largely follow the update mechanics outlined in [4], with the addition of a new PSet^+ data structure.

To match the SQL standard, our implementation uses the `LIKE` operator syntax for substring and wildcard queries: '`_`' ('`%`') represent single-character (variable-length) wildcards and the query must match the complete field, i.e.,

unless a query must match the prefix (suffix) of fields, it should begin (end) with a '%'.

Experimental Platform. The experiments described in the remainder of this section were run on two Dell PowerEdge R710 systems, each one of them equipped with two Intel Xeon X5650 processors, 96 GB RAM (12x8 1066 MHz), an embedded Broadcom 1 GB Ethernet with TOE and a PERC H700 RAID controller with a 1 GB Non-Volatile Cache and 1 or 2 daisy-chained MD1200 disk controllers each with 12 2 TB 7.2 k RPM Near-Line SAS hard drives configured for Raid 6 (19 TB and 38 TB total storage per machine).

An automated test harness, written by an independent evaluator [25], drives the evaluation, including the set of queries and the dataset used in the experiments.

Dataset. The synthetic dataset used in the reported experiments is a US census-like table with twenty one columns of standard personal information, such as name (first, last), address (street, city, state, zipcode), SSN, etc. The values in each column are generated according to the distributions in the most recent US census. In addition, the table has one XML column with at most 10000 characters, four text columns with varying average lengths (a total of at most 12300 characters or $\approx$ 2000 words), and a binary column (payload) with a maximum size of 100 KB. Our system can perform structured queries on data in all but the XML and binary columns. The size of (number of records in) the table is a parameter of the dataset generator. We tested on a wide variety of database sizes, but we focus our results on a table with 100 million records or 10 TBytes.

Cryptographic Algorithms. Our implementation uses AES-128 for all symmetric key operations (including data encryption), SHA-256 for hashing, and NIST's Elliptic Curve p224 for group operations.

Experimental Methodology. In the initial step, the encrypted database is created from the cleartext data stored in a MariaDB (a variant of open-source MySQL RDBMS) table. Then, a per-protocol collection of SQL queries, generated by the harness to test its features, is run against the MariaDB sever and against our system. The queries are issued sequentially by the harness, which also records the results and the execution times of each query. Finally, the harness validates the test results by comparing the result sets from our system and from the MariaDB server. Not only does this step validate the correctness of our system, it also ensures our system meets our theoretical false positive threshold over large, automatically generated, collections of queries.

Encrypted Index. We built a searchable index on all personal information columns (twenty one) in the plaintext database but we only use a small subset of these indexes for the following experiments. Note that we support substring and wildcard queries simultaneously over a given column using a single shared index. We built a substring-wildcard index for four columns (average length of 12 characters) and a range index for five columns of varying types (one 64 bit integer, one date, one 32 bit integer, and one enum). Each substring-wildcard

index was constructed with a single k value of 4. Each range index has a granularity of one. For the date type, this equates to a day. We support date queries between 0-01-01 and 9999-12-31, and integer queries between 0 and integer max ($2^{32}-1$ or $2^{64}-1$).

On average each record generates 256.6 document-keyword pairs (tuples) among all indexes. This equates to a total encrypted index for our largest database of ≈ 20 TB. We back our XSet by an in memory Bloom filter with a false positive rate of 2^{-12}; this allows us to save unnecessary disk accesses and it does not influence the false positive rate of the system.

Query Flexibility. While many queries can be formed by using substring or wildcard queries independently, many queries are not computable. We can greatly increase the number of available queries by combining the two query types. This allows us to answer any query q s.t. all non-wildcard characters in q are part of at least one k length substring containing no wildcards and q starts and ends with a non-wildcard character. This may require a sufficiently large k (a performance benefit) but limit the type of queries supported. To further increase flexibility we can index fields with multiple values for k or with a different k for each data structure: k_x for XSet and k_s for TSet. The result is a very flexible policy that we can support any query q that meets the following: (1) there exists at least one consecutive k_s length sequence of non-wildcards in q, (2) all non-wildcard characters in q are part of at least one k_x length substring containing no wildcards, and (3) q starts and ends with a non-wildcard character.

Performance Costs by Query Type. Our complex query types have both increased storage overhead and query time costs as compared to the keyword only implementation of [4]. In order to support substring and wildcard queries on a column, we must store additional tuples: for a record of length l (for the indexed field) we must store $(l-k)+3$ tuples. Note that we must pay this cost for each k we chose to create the index for. The choice of k also affects query time performance. For a query q, it's performance is linearly dependent on the number of tokens generated by the tokenization $T(q)$. A smaller k results in a larger number of tokens. Specifically for subsequence queries there will be $\lceil |q|/k \rceil$-1 xtokens[2]. k also impacts the number of matching documents returned by the s-term. A larger k results in a higher entropy s-term. The choice of k is a careful trade-off between efficiency and flexibility.

Range queries incur storage costs linear in their bit depth. Specifically, $log_2(max\_value)$ tuples are stored for a record for each range field. Notably for date fields this value is 22. In addition we implemented the *canonical cover* from Sect. 3, which results in up to $2 * log_2(max\_value)$ disjunctions.

Phrase queries incur storage costs linear in the total number of words in a column. Specifically for every record with n free-text words, the index stores n tuples. Although phrase queries and free-text queries can be supported via the

[2] Wildcard queries pay a similar overhead, related to the size of each contiguous substring within the query.

same index, we have to pay the marginally higher price of the phrase index in which we must store even repeated words.

Encrypted Search Performance. We illustrate the performance of our system using the latency (i.e., total time from query issuing to completion) of a large number of representative `SELECT` queries. The independent evaluator selected a representative set of queries to test the correctness and performance of the range, substring and wildcard queries (phrase queries were not implemented). The two leftmost columns in Table 1 show how many unique queries were selected for each query type. The third, fourth and fifth columns characterize the 95 % fastest queries of each type. Finally, the rightmost column shows the percentage of queries that complete in less than two minutes.

All queries follow the pattern `SELECT id FROM CensusTable WHERE ...`, with each query having a specific WHERE clause. Range-type queries use the `BETWEEN` operator to implement two-sided comparison on numerical fields as well as date and enum fields. Specific queries were chosen to assess the performance effect of differing result set sizes and range covers. In particular, in order to assess the effect of cover size, queries with moderate result sets (of size under 10,000) were chosen while the size of cover sets range from a handful to several dozens. The results show relatively homogeneous latencies (all under 0.8 s) in spite of the large variations in cover size, highlighting the moderate effect of cover sizes.

Our instantiation of SUB-SSE-OXT includes extensions for supporting substring and wildcard searches simultaneously. However, to evaluate the effects of each specific extension we measure them individually. Both query types use the `LIKE` operator in the `WHERE` clause.

Substring queries use the variable-length wildcard '%' at the beginning, at the end, or at both ends of the `LIKE` operand, as in `WHERE city LIKE '%ttle Falls%'`. Wildcard queries use the single-character wildcard ('_') anywhere in the `LIKE` operand, provided the query criteria dictated by k is still met.

In addition, we noticed that the choice of s-gram dominates the latency of the substring queries. Our analysis shows that low performing queries can often be tied to high-frequency s-terms (e.g., "ing" or "gton"), which are associated with large Tsets. By default, the current implementation uses the first k characters in the pattern string as s-gram. Thus, implementing a tokenization strategy guided by the text statistics (which we leave for future work) can significantly reduce query latency for many of the slow performers. To estimate the potential benefits of such a strategy, we added the `STARTAT 'n'` option to the `LIKE 'pattern'` operator, where `'n'` is the starting position of the s-gram. Experiments using the `'%gton Colle%'` pattern show latency improvements of up to 32 times when the s-gram starts at the third or fourth character in the pattern string.

Comparison to Cleartext Search. Here we include the most relevant aspects of the performance comparison between our prototype and MariaDB. In the case of the 100 million record database, for ≈ 45 % of the range queries, the two systems have very similar performance. For the remaining 55 %, our system is increasingly (up to 500 times!) faster, admittedly due to MariaDB's lack of support for indexed substring search. The large variations in MariaDB performance

Table 1. Latency (in secs) for 10 TByte DB, 100 M records, 25.6 billion record-keyword pairs

Query type	# of queries	Fastest 95 %			% ≤ 120 s
		Avg	Min	Max	
Range	197	.37	.19	.61	100
Substring	939	40	0.22	166	93
Wildcard	511	31.22	6.7	224	93

seem to arise from its reliance on data (and index) caching, which is hindered by large DBs. In contrast, our system issues between $\log_2 s$ and $2\log_2 s$ disk accesses *in parallel* (where s is the size of the cover). On smaller census databases (with fewer records) that fit in RAM, MariaDB outperforms our system, sometimes by more than one order of magnitude, although in this case all query latencies (ours and MariaDB's) are under a second. For substring and wildcard queries and the largest, 100 million records database, our system outperforms MariaDB by such a large factor largely because MariaDB does not use any index for these queries. Instead, it scans the full database to resolve queries involving the `LIKE` operator.

References

1. Boneh, D., Boyen, X.: Efficient selective-id secure identity-based encryption without random oracles. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 223–238. Springer, Heidelberg (2004)
2. Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 41–55. Springer, Heidelberg (2004)
3. Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 535–554. Springer, Heidelberg (2007)
4. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very large databases: data structures and implementation. In: Symposium on Network and Distributed Systems Security (NDSS 2014) (2014)
5. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013)
6. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
7. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010)

8. Chase, M., Shen, E.: Pattern matching encryption. Cryptology ePrint Archive, Report 2014/638 (2014). http://eprint.iacr.org/
9. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Juels, A., Wright, R.N., Vimercati, S. (eds.) ACM CCS 06: 13th Conference on Computer and Communications Security, pp. 79–88. ACM Press, Alexandria (2006)
10. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M.C., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive (2015). http://eprint.iacr.org/2015
11. Goh, E.J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003). http://eprint.iacr.org/
12. Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Outsourced symmetric private information retrieval. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 875–888. ACM (2013)
13. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013)
14. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 12: 19th Conference on Computer and Communications Security, pp. 965–976. ACM Press, Raleigh (2012)
15. Kiayias, A., Tang, Q.: How to keep a secret: leakage deterring public-key cryptosystems. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 943–954. ACM (2013)
16. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012)
17. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Jonker, W., Petković, M. (eds.) SDM 2010. LNCS, vol. 6358, pp. 87–100. Springer, Heidelberg (2010)
18. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: 35th IEEE Symposium on Security and Privacy, pp. 639–654. IEEE Computer Society Press (2014)
19. Pappas, V., Vo, B., Krell, F., Choi, S., Kolesnikov, V., Keromytis, A., Malkin, T.: Blind seer: a scalable private DBMS. In: 35th IEEE Symposium on Security and Privacy, pp. 359–374. IEEE Computer Society Press (2014)
20. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011). ACM, October 2011
21. Raykova, M., Vo, B., Bellovin, S.M., Malkin, T.: Secure anonymous database search. In: Proceedings of the 2009 ACM Workshop on Cloud computing security, pp. 115–126. ACM (2009)
22. Shacham, H.: A cramer-shoup encryption scheme from the linear assumption and from progressively weaker linear variants. Cryptology ePrint Archive, Report 2007/074 (2007). http://eprint.iacr.org/
23. Shi, E., Bethencourt, J., Chan, T.H., Song, D., Perrig, A.: Multi-dimensional range query over encrypted data. In: IEEE Symposium on Security and Privacy, SP 2007, pp. 350–364. IEEE (2007)

24. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, pp. 44–55. IEEE Computer Society Press, Oakland, May 2000
25. Varia, M., Price, B., Hwang, N., Hamlin, A., Herzog, J., Poland, J., Reschly, M., Yakoubov, S., Cunningham, R.K.: Automated assesment of secure search systems. Operating Syst. Rev. **49**(1), 22–30 (2015)