Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries

David Cash[1], Stanislaw Jarecki[2], Charanjit Jutla[3], Hugo Krawczyk[3], Marcel-Cătălin Roşu[3], and Michael Steiner[3]

[1] Rutgers University
david.cash@cs.rutgers.edu
[2] University of California Irvine
stasio@ics.uci.edu
[3] IBM Research
{csjutla,hugokraw,rosu,msteiner}@us.ibm.com

Abstract. This work presents the design and analysis of the first searchable symmetric encryption (SSE) protocol that supports conjunctive search and general Boolean queries on outsourced symmetrically-encrypted data and that scales to very large databases and arbitrarily-structured data including free text search. To date, work in this area has focused mainly on single-keyword search. For the case of conjunctive search, prior SSE constructions required work linear in the total number of documents in the database and provided good privacy only for structured attribute-value data, rendering these solutions too slow and inflexible for large practical databases.

In contrast, our solution provides a realistic and practical trade-off between performance and privacy by efficiently supporting very large databases at the cost of moderate and well-defined leakage to the outsourced server (leakage is in the form of data access patterns, never as direct exposure of plaintext data or searched values). We present a detailed formal cryptographic analysis of the privacy and security of our protocols and establish precise upper bounds on the allowed leakage. To demonstrate the real-world practicality of our approach, we provide performance results of a prototype applied to several large representative data sets, including encrypted search over the whole English Wikipedia (and beyond).

1 Introduction

Outsourcing data storage to external servers ("the cloud") is a major industry trend that offers great benefits to database owners. At the same time, data outsourcing raises confidentiality and privacy concerns. Simple encryption of outsourced data is a hindrance to search capabilities such as the data owner wanting to search a backup or email archive, or query a database via attribute-value pairs. This problem has motivated much research on advanced searchable encryption schemes that enable searching on the encrypted data while protecting the confidentiality of data and queries.

R. Canetti and J.A. Garay (Eds.): CRYPTO 2013, Part I, LNCS 8042, pp. 353–373, 2013.
© International Association for Cryptologic Research 2013

Searchable Symmetric Encryption (SSE) is a cryptographic primitive addressing encrypted search. To securely store and search a database with an SSE scheme, a client first uses a special encryption algorithm which produces an encrypted version of the database, including encrypted metadata, that is then stored on an external server. Later, the client can interact with the server to carry out a search on the database and obtain the results (this is called the symmetric setting as there is only one writer to the database, the owner, who uses symmetric encryption – the public key variant of the problem has also been extensively studied, see further related work).

An important line of research (e.g., [23,11,6,8,7,19]) gives practical constructions of SSE that support searching for documents that contain a *single* specified keyword. In these schemes, the server's work scales with the size of the result set (independently of the database size) and the leakage to the server is limited to the set of returned (encrypted) documents and a few global parameters of the system, such as total data size and number of documents. While efficient and offering good privacy, all of these SSE schemes are severely limited in their expressiveness during search: A client can *only* specify a single keyword to search on, and then it receives *all* of the documents containing that keyword. In practical settings, like remotely-stored email or large databases, a single-keyword search will often return a large number of documents that the user must then download and filter herself to find the relevant results.

Conjunctive and Boolean Search. To provide a truly practical search capability, a system needs to at least support conjunctive search, namely, given a set of keywords find all documents that contain all these keywords. Clearly, this problem can be reduced to the single-keyword case by performing a search for each individual keyword and then letting the server or client do the intersection between the resultant document sets. This often results in inefficient searches (e.g., half the database size if one of the conjunctive terms is "gender=male") and significant leakage (e.g., it reveals the set of documents matching each keyword). Yet, this naïve solution is the only known sublinear solution to SSE conjunctive search (other than those using generic techniques such as FHE or ORAM). All other dedicated solutions require server work that is *linear in the size of the database*. Of these solutions, the one that provides the best privacy guarantees is due to Golle et al. [13], with variants presented in later work, e.g., [1,3]. They show how to build for each conjunctive query a set of tokens that can be tested against *each* document in the database (more precisely, against an encoded version of the document's keywords) to identify matching documents. These solutions only leak the set of matching documents (and possibly the set of attributes being searched for) but are unsuited for large databases due to the $O(d)$ work incurred by the server, where d is the number of documents or records in the database. This cost is paid for every search regardless of the size of the result set or the number of documents matching each individual conjunctive term. Moreover, these solutions require either $O(d)$ communication and exponentiations between server and client or $O(d)$ costly pairing operations (as well as dedicated cryptographic assumptions). Another serious limitation of this

approach is that it works only for structured attribute-value type databases and does not support free text search. In addition, none of the above solutions extend to general Boolean queries.

The Challenge of Large Databases and the Challenge of Being Imperfect. In this work we investigate solutions to conjunctive queries and general Boolean queries that can be practical even for very large databases where linear search is prohibitively expensive. Our application settings include databases that require search over tens of millions documents (and billions of document-keyword pairs), with search based on attribute-value pairs (as in relational databases) and free text - see below for specific numbers used in evaluating our prototype. To support such scale in a truly practical way one needs to relax absolute privacy and allow for some leakage beyond the result set.

As an example, compare the case of a conjunction of two highly-frequent keywords whose intersection returns a small number of documents but whose individual terms are very frequent (e.g., search for "name=David AND gender=Female"), with the case of a conjunction that returns the same number of documents but all the individual terms in the conjunction are themselves infrequent. Search complexity in these two cases, *even in the case of plaintext data (hence in any encrypted solution),* is likely to be different and noticeable to the searching server, except if searches are artificially padded to a full database search hence leading to $O(d)$ complexity[1]. Note that even powerful tools, such as ORAM, that can be used to search on encrypted data in smaller-scale databases already incur non-trivial leakage if the search performance is to be sublinear. Indeed, the mere computational cost, in number of ORAM operations, of a given search is sufficient to distinguish between the two cases above (of all high-frequency conjunctive terms vs. all small-frequency terms) unless the searches are padded to the maximal search size, resulting in $O(d)$ search cost. Thus, resorting to weaker security guarantees is a necessity for achieving practical conjunctive search. Not only this presents design challenges but also raises non-trivial theoretical challenges for analyzing and characterizing in a precise way the form and amount of leakage incurred by a solution.

Ideally, we would like to run the search with complexity proportional to the number of matches of the *least frequent term in the conjunction*, which is the standard of plaintext information retrieval algorithms. In addition, the computational efficiency of database processing and of search is of paramount importance in practice. Generic tools such as FHE [10] or ORAM [12] are too costly for very large databases, although they may be used as sub-components of a solution if applied to small data subsets.

Our Contributions. We develop the first non-generic sublinear SSE schemes supporting conjunctive keyword search (and more general Boolean queries, see below) with a non-trivial combination of security and efficiency. The schemes performance scales to very large datasets and arbitrarily-structured data, including free-text search. We attain efficiency by allowing some forms of access-pattern

[1] A costly alternative is to pre-compute all n-term conjunctions in time $O(|\mathsf{W}|^n)$.

leakage, but with a much better leakage profile than the naïve solution implied by single-keyword SSE, as discussed above. Further, we establish the security of our solution via an explicit and precise leakage profile and a *proof that this is all the leakage* incurred by this solution. Our formal setting follows a simulation-based abstraction that adapts the SSE models of Curtmola et al. [8] and Chase and Kamara [7], and assumes an adaptive adversarial model. The essence of the security notion is that the view of the server (the attacker in this setting) can be efficiently simulated given a precisely-defined *leakage profile* but without access to the actual plaintext data. Such a profile may include leakage on the total size of the database, on access patterns (e.g., the intersection between two sets of results) and on queries (e.g., repetition of queries), but never the direct exposure of plaintext data or searched values. Thus, a protocol proven secure ensures that the server holding the encrypted data and serving the queries does not learn anything about the data and queries other than what can be deduced from the specified leakage[2]. The characterization of leakage and the involved proof of security that we present are central technical contributions that complement our protocol design work.

The centerpiece of the protocol design is a "virtual" secure two-party protocol in which the server holds encrypted pointers to documents, the client holds a list of keywords, and the output of the protocol is the set of encrypted pointers that point to documents containing all the client's keywords. The client is then able to decrypt these pointers and obtain the matching (encrypted) documents but the server cannot carry this decryption nor can it learn the keywords in the client's query. While this underlying protocol is interactive, the level of performance targeted by our solutions requires avoiding multiple rounds of interaction. We achieve this by a novel approach that pre-computes parts of the protocol messages and stores them in encrypted form at the server. Then, during search, the client sends information to the server that allows to unlock these pre-computed messages without further interaction. Our implementation of this protocol, which we name OXT, uses only DH-type operations over any Diffie-Hellman group which enables the use of the secure and most efficient DH elliptic curves (with additional common-base optimizations).[3] The complexity of our search protocols is *independent* of the number of documents in the database. To search for documents containing $w_1, \ldots, w_n$, the search complexity of our scheme scales with the number of documents matching the estimated *least frequent keyword in the conjunction*. We note that while building a search based on term frequency is standard in information retrieval, our solution seems to be the first to exploit this approach in the encrypted setting. This leads not only to good performance but also improves privacy substantially. All our solutions support search on structured data (e.g., attribute-value databases) as well as on free text, and combinations of both.

[2] See the discussion in Section 6 on "semantic leakage".

[3] We also present a scheme (BXT in Section 3.1) that only uses symmetric-key operations but provides less privacy, and a pairings-based scheme (PXT in the full version [5]) that optimizes communication at the expense of more computation.

Boolean Queries. Our solution to conjunctive queries extends to answer *any Boolean query.* This includes negations, disjunctions, threshold queries, and more. The subset of such queries that we can answer efficiently includes any expression of the form "$w_1 \wedge \phi(w_2, \ldots, w_m)$" (intended to return any document that matches keyword w_1 and in addition satisfies the (unconstrained) formula ϕ on the remaining keywords)[4]. The search complexity is proportional to the number of documents that contain w_1. Surprisingly, the leakage profile for such complex expressions can be reduced to the leakage incurred by a conjunction with the same terms $w_1, w_2, \ldots, w_n$, hence allowing us to re-use the analysis of the conjunctive case to the much more general boolean setting. Finally, any disjunction of the above forms can also be answered with an additive cost over the disjunction expressions.

Further Extensions. In [4] we report on further practical enhancements to our protocols, including support for dynamic databases (i.e., allowing additions, deletions and modification of documents in the database). Our protocols can also be applied to the *multi-client setting* [7,17,18] where a data owner outsources its encrypted data to an external server and enables *other parties* to perform queries on the encrypted data by providing them with search tokens for specific queries. In this case, one considers not only leakage to the server but also leakage to clients beyond the information that their tokens are authorized to disclose. In subsequent work [16], we address issues of authorization in this setting as well as the challenging problem of hiding the queries not only from the server but also from the token provider - see for example IARPA's SPAR program and its requirement for supporting private queries on very large databases [14]. See also [21] for an independent, concurrent work in the latter setting from which a solution to the SSE problem can also be extracted. Finally, in ongoing work, we are extending the set of supported queries with range queries, substring matching, and more.

Implementation. To show the practical viability of our solution we prototyped OXT and ran experiments with two data sets: the Enron email data set [9] with more than 1.5 million documents (email messages and attachments) where all words, including attachments and envelope information, have been indexed; and the ClueWeb09 [20] collection of crawled web-pages from which we extracted several databases of increasing size with the largest one consisting of 13 million documents (0.4TB of HTML files). Approximately one third of the latter database is a full snapshot of the English Wikipedia. The results of these tests show not only the suitability of our conjunction protocols for data sets of medium size (such as the Enron one) but demonstrate the scalability of these solutions to much larger databases (we target databases of one or two orders of magnitude larger). Existing solutions that are linear in the number of documents would be mostly impractical even for the Enron dataset. Refer to Section 5 for more information on implementation and performance. More advanced results are reported in [4].

[4] An example of such query on an email repository is: Search for messages with Alice as Recipient, not sent by Bob, and containing at least two of the words {searchable, symmetric, encryption}.

Other Related Work and Research Questions. See full version [5] for more discussion on related work and Section 6 for several interesting research questions arising from our work.

2 Definitions and Tools

2.1 SSE Syntax and Security Model

Searchable Symmetric Encryption. A database is composed of a collection of d documents, each comprised of a set of keywords W_i (we use "documents" generically; they can represent text documents, records in a relational database - in which case keyword are represented as attribute-value pairs, a combination of both, etc.). The output from the SSE protocol for a given search query are indices (or identifiers) ind corresponding to the documents that satisfy the query. A client program can then use these indices to retrieve the encrypted documents and decrypt them. This definition allows to decouple the storage of payloads (which can be done in a variety of ways, with varying types of leakage) from the storage of metadata that is the focus of our protocols.

SSE Scheme Syntax and Correctness. Let λ be the security parameter. We will take identifiers and keywords to be bit strings. A database $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$ is represented as a list of identifier/keyword-set pairs, where $\mathsf{ind}_i \in \{0,1\}^\lambda$ and $\mathsf{W}_i \subseteq \{0,1\}^*$. We will always write $\mathsf{W} = \bigcup_{i=1}^d \mathsf{W}_i$ (we think of the ind values as identifiers that can be revealed to the outsourced server, e.g., a randomization of the original document identifiers; in particular these are the identifiers that will be used to retrieve query-matching documents). A *query* $\psi(\bar{w})$ is specified by a tuple of keywords $\bar{w} \in \mathsf{W}^*$ and a boolean formula ψ on $\bar{w}$. We write $\mathsf{DB}(\psi(\bar{w}))$ for the set of identifiers of documents that "satisfy" $\psi(\bar{w})$. Formally, this means that $\mathsf{ind}_i \in \mathsf{DB}(\psi(\bar{w}))$ iff the formula $\psi(\bar{w})$ evaluates to true when we replace each keyword w_i with true or false depending on if $w_i \in \mathsf{W}_i$ or not. Below we let d denote the number of documents in DB, $m = |\mathsf{W}|$ and $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.

A *searchable symmetric encryption (SSE) scheme Π* consists of an algorithm $\mathsf{EDBSetup}$ and a protocol Search between the client and server, all fitting the following syntax. $\mathsf{EDBSetup}$ takes as input a database DB, and outputs a secret key K along with an encrypted database EDB. The search protocol is between a *client* and *server*, where the client takes as input the secret key K and a query $\psi(\bar{w})$ and the server takes as input EDB. At the end of the protocol the client outputs a set of identifiers and the server has no output. We say that an SSE scheme is *correct* if for all inputs DB and queries $\psi(\bar{w})$ for $\bar{w} \in \mathsf{W}^*$, if $(K, \mathsf{EDB}) \stackrel{\$}{\leftarrow} \mathsf{EDBSetup}(\mathsf{DB})$, after running Search with client input $(K, \psi(\bar{w}))$ and server input EDB, the client outputs the set of indices $\mathsf{DB}(\psi(\bar{w}))$.

Security of SSE. We recall the semantic security definitions from [8,7]. The definition is parametrized by a *leakage function $\mathcal{L}$*, which describes what an adversary (the server) is allowed to learn about the database and queries when

interacting with a secure scheme. Formally, security says that the server's view during an adaptive attack (where the server selects the database and queries) can be simulated given only the output of $\mathcal{L}$.

Definition 1. *Let* $\Pi = (\mathsf{EDBSetup}, \mathsf{Search})$ *be an SSE scheme and let* $\mathcal{L}$ *be a stateful algorithm. For algorithms* A *and* S*, we define experiments (algorithms)* $\mathbf{Real}^{\Pi}_{A}(\lambda)$ *and* $\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$ *as follows:*

$\mathbf{Real}^{\Pi}_{A}(\lambda)$ *:* $A(1^\lambda)$ *chooses* DB*. The experiment then runs* $(K, \mathsf{EDB}) \leftarrow \mathsf{EDBSetup}(\mathsf{DB})$ *and gives* EDB *to* A*. Then* A *repeatedly chooses a query* q*. To respond, the game runs the* Search *protocol with client input* (K, q) *and server input* EDB *and gives the transcript and client output to* A*. Eventually* A *returns a bit that the game uses as its own output.*

$\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$ *: The game initializes a counter* $i = 0$ *and an empty list* $\mathbf{q}$*.* $A(1^\lambda)$ *chooses* DB*. The experiment runs* $\mathsf{EDB} \leftarrow S(\mathcal{L}(\mathsf{DB}))$ *and gives* EDB *to* A*. Then* A *repeatedly chooses a query* q*. To respond, the game records this as* $\mathbf{q}[i]$*, increments* i*, and gives to* A *the output of* $S(\mathcal{L}(\mathsf{DB}, \mathbf{q}))$*. (Note that here,* $\mathbf{q}$ *consists of all previous queries in addition to the latest query issued by* A*.) Eventually* A *returns a bit that the game uses as its own output.*

We say that Π *is* $\mathcal{L}$*-semantically-secure against adaptive attacks if for all adversaries* A *there exists an algorithm* S *such that* $\Pr[\mathbf{Real}^{\Pi}_{A}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{A,S}(\lambda) = 1] \leq \mathrm{neg}(\lambda)$.

We note that in the security analysis of our SSE schemes we include the client's output, the set of indices $\mathsf{DB}(\psi(\bar{w}))$, in the adversary's view in the real game, to model the fact that these ind's will be used for retrieval of encrypted document payloads.

2.2 T-Sets

We present a definition of syntax and security for a new primitive that we call a *tuple set*, or *T-set*. Intuitively, a T-set allows one to associate a list of fixed-sized data tuples with each keyword in the database, and later issue keyword-related tokens to retrieve these lists. We will use it in our protocols as an "expanded inverted index". Indeed, prior single-keyword SSE schemes, e.g. [8,7], can be seen as giving a specific T-set instantiation and using it as an inverted index to enable search – see Section 2.3. In our SSE schemes for conjunctive keyword search, we will use a T-set to store more data than a simple inverted index, and we will also compose it with other data structures. The abstract definition of a T-set will allow us to select an implementation that provides the best performance for the size of the data being stored.

T-Set Syntax and Correctness. Formally, a T-set implementation $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ will consist of three algorithms with the following syntax: $\mathsf{TSetSetup}$ will take as input an array $\mathbf{T}$ of lists of equal-length bit strings indexed by the elements of W. The $\mathsf{TSetSetup}$ procedure outputs

a pair (TSet, K_T). $\mathsf{TSetGetTag}$ takes as input the key K_T and a keyword w and outputs stag. $\mathsf{TSetRetrieve}$ takes the TSet and an stag as input, and returns a list of strings. We say that Σ is *correct* if for all W, $\mathbf{T}$, and any $w \in \mathsf{W}$, $\mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag}) = \mathbf{T}[w]$ when $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$ and $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w)$. Intuitively, $\mathbf{T}$ holds lists of tuples associated with keywords and correctness guarantees that the $\mathsf{TSetRetrieve}$ algorithm returns the data associated with the given keyword.

T-Set Security and Implementation. The security goal of a T-set implementation is to hide as much as possible about the tuples in $\mathbf{T}$ and the keywords these tuples are associated to, except for vectors $\mathbf{T}[w_1], \mathbf{T}[w_2], \ldots$ of tuples revealed by the client's queried keywords $w_1, w_2, \ldots$. (For the purpose of T-set implementation we equate client's query with a single keyword.) The formal definition is similar to that of SSE (think of the SSE setting for single-keyword queries) and we provide it in [5] where we also show a specific T-set implementation that achieves optimal security, namely, it only reveals (an upper bound on) the aggregated database size $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$ *We refer to such a T-set implementation as* optimal.

2.3 T-Sets and Single Keyword Search

Here we show how a T-set can be used as an "secure inverted index" to build an SSE scheme for single-keyword search. The ideas in this construction will be the basis for our conjunctive search SSE schemes later, and it essentially abstracts prior constructions [8,7]. The details of the scheme, called SKS, are given in

<u>EDBSetup(DB)</u>

- Select key K_S for PRF F, and parse DB as $(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$.
- Initialize $\mathbf{T}$ to an empty array indexed by keywords from $\mathsf{W} = \cup_{i=1}^d \mathsf{W}_i$.
- For each $w \in \mathsf{W}$, build the tuple list $\mathbf{T}[w]$ as follows:
  - Initialize t to be an empty list, and set $K_e \leftarrow F(K_S, w)$.
  - For all $\mathsf{ind} \in \mathsf{DB}(w)$ in random order: $\mathsf{e} \xleftarrow{\$} \mathsf{Enc}(K_e, \mathsf{ind})$; append e to t.
  - Set $\mathbf{T}[w] \leftarrow \mathsf{t}$.
- $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$.
- Output the key (K_S, K_T) and $\mathsf{EDB} = \mathsf{TSet}$.

<u>Search protocol</u>

- The client takes as input the key (K_S, K_T) and a keyword w to query. It computes $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w)$ and sends stag to the server.
- The server computes $\mathsf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$, and sends t to the client.
- Client sets $K_e \leftarrow F(K_S, w)$; for each e in t, it computes $\mathsf{ind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ and outputs ind.

Fig. 1. SKS: Single-Keyword SSE Scheme

Figure 1. It uses as subroutines a PRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$, and a CPA secure symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ that has λ-bit keys.

3 SSE Schemes for Conjunctive Keyword Search

Existing SSE schemes for conjunctive queries ([13] and subsequent work) work by encoding each document individually and then processing a search by testing *each* encoded document against a set of tokens. Thus the server's work grows linearly with the number of documents, which is infeasible for large databases. In addition, these schemes only work for attribute-value type databases (where documents contain a single value per attribute) but not for unstructured data, e.g., they cannot search text documents.

Here *we develop the first sub-linear conjunctive-search solutions for arbitrarily-structured data, including free text.* In particular, when querying for the documents that match all keywords $w_1, \ldots, w_n$, our search protocol scales with the size of the (estimated) *smallest* $\mathsf{DB}(w_i)$ set among all the conjunctive terms w_i.

The naïve solution. To motivate our solutions we start by describing a straightforward extension of the single-keyword case (protocol SKS from Figure 1) to support conjunctive keyword searching. On input a conjunctive query $\bar{w} = (w_1, \ldots, w_n)$, the client and server run the search protocol from SKS independently for each term w_i in $\bar{w}$ with the following modifications. Instead of returning the lists $\mathbf{t}$ to the client, the server receives K_{e_i}, $i = 1, ..., n$, from the client and decrypts the e values to obtain a set of ind's for each w_i. Then, the server returns to client the ind values in the intersection of all these sets. The search complexity of this solution is proportional to $\sum_{i=1}^{n} |\mathsf{DB}(w_i)|$ which improves, in general, on solutions whose complexity is linear in the number of documents in the whole database. However, this advantage is reduced for queries where one of the terms is a very high-frequency word (e.g., in a relational database of personal records, one may have a keyword $w = (gender, male)$ as a conjunctive term, thus resulting in a search of, say, half the documents in the database). In addition, this solution incurs excessive leakage to the server who learns the complete sets of indices ind for each term in a conjunction.

Our goal is to reduce both computation and leakage in the protocol by tying those to the less frequent terms in the conjunctions (i.e., terms w with small sets $\mathsf{DB}(w)$).

3.1 Basic Cross-Tags (BXT) Protocol

To achieve the above goal we take the following approach that serves as the basis for our main SSE-conjunctions scheme OXT presented in the next subsection. Here we exemplify the approach via a simplified protocol, BXT. Assume (see below) that the client, given $\bar{w} = (w_1, \ldots, w_n)$, can choose a term w_i with a relatively small $\mathsf{DB}(w_i)$ set among $w_1, \ldots, w_n$; for simplicity assume this is w_1. The parties could run an instance of the SKS search protocol for the keyword w_1 after which the client gets all documents matching w_1 and locally searches for the remaining conjunctive terms. This is obviously inefficient as it may require

retrieving many more documents than actually needed. The idea of BXT is indeed to use SKS for the server to retrieve $\mathsf{TSet}(w_1)$ but then perform the intersection with the terms $w_2, \ldots, w_n$ at the server who will only return the documents matching the full conjunction. We achieve this by augmenting SKS as follows.

During $\mathsf{EDBSetup}(\mathsf{DB})$, in addition to TSet, a set data structure XSet is built by adding to it elements xtag computed as follows. For each $w \in \mathsf{W}$, a value $\mathsf{xtrap} = F(K_X, w)$ is computed where K_X is a PRF key chosen for this purpose; then for each $\mathsf{ind} \in \mathsf{DB}(w)$ a value $\mathsf{xtag} = f(\mathsf{xtrap}, \mathsf{ind})$ is computed and added to XSet where f is an unpredictable function of its inputs (e.g., f can be a PRF used with xtrap as the key and ind as input). The Search protocol for a conjunction $(w_1, \ldots, w_n)$, chooses the estimated least frequent keyword, say w_1, and sets, as in SKS, $K_e \leftarrow F(K_S, w_1)$, $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w_1)$. Then, for each $i = 2, \ldots, n$, it sets $\mathsf{xtrap}_i \leftarrow F(K_X, w_i)$ and sends $(K_e, \mathsf{stag}, \mathsf{xtrap}_2, \ldots, \mathsf{xtrap}_n)$ to the server. The server uses stag to retrieve $\mathbf{t} = \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$. Then, for each ciphertext e in $\mathbf{t}$, it decrypts $\mathsf{ind} = \mathsf{Dec}(K_e, \mathsf{e})$ and if $f(\mathsf{xtrap}_i, \mathsf{ind}) \in \mathsf{XSet}$ for all $i = 2, \ldots, n$, it sends ind to the client.[5]

Correctness of the BXT protocol is easy to verify. Just note that a document indexed by ind includes a word w represented by stag if and only if $\mathsf{xtag} = f(\mathsf{xtrap}, \mathsf{ind}) \in \mathsf{XSet}$. Regarding implementation of XSet, it can use any set representation that is history-independent, namely, it is independent of the order in which the elements of the set were inserted. For TSet security and implementation see Section 2.

Terminology (s-terms and x-terms): We will refer to the conjunctive term chosen as the estimated least frequent term among the query terms as the *s-term* ('s' for SKS or "small") and refer to other terms in the conjunction as *x-terms* ('x' for "cross"); this is the reason for the 's' and 'x' in names such as $\mathsf{stag}, \mathsf{xtag}, \mathsf{stag}, \mathsf{xtrap}$, etc.

The server's work in BXT scales with $n \cdot |\mathsf{DB}(w_1)|$, where w_1 is the conjunction's s-term. This represents a major improvement over existing solutions which are linear in $|\mathsf{DB}|$ and also a significant improvement over the naïve solution whenever there is a term with relatively small set $\mathsf{DB}(w_1)$ that can be identified by the client, which is usually the case as discussed in Section 3.1. Communication is optimal ($O(n)$-size token plus the final results set) and computation involves only PRF operations.

Security-wise this protocol *improves substantially* on the above-described naïve solution by leaking only the (small) set of ind's for the s-term and not for x-terms. Yet, this solution lets the server learn statistics about x-terms by correlating information from different queries. Specifically, the server can use the value xtrap_i received in one query and check it against any ind found through an s-term of another query. But note that direct intersections between x-terms of different queries are not possible other than via the s-terms (e.g., if two queries (w_1, w_2) and (w_1', w_2') are issued, the server can learn the (randomly permuted) results of (w_1, w_2') and (w_1', w_2) but not (w_2, w_2').

[5] While in SKS one can choose to let the server decrypt the ind's directly instead of the client, in BXT this is necessary for computing the xtag's.

In settings where computation and communications are very constrained BXT may provide for an acceptable privacy-performance balance. In general, however, we would like to improve on the privacy of this solution even if at some performance cost. We do so in the next section with the OXT protocol, so we omit a formal analysis BXT – we note that the security of BXT needs the set of ind's to be unpredictable, a condition not needed in the other protocols.

Choosing the S-Term. The performance and privacy of our conjunction protocols improves with "lighter" s-terms, namely, keywords w whose $\mathsf{DB}(w)$ is of small or moderate size. While it is common to have such terms in typical conjunctive queries, our setting raises the question of how can the client, who has limited storage, choose adequate s-terms. In the case of relational databases one can use general statistics about attributes to guide the choice of the s-term (e.g., prefer a last-name term to a first-name term, always avoid gender as the s-term, etc.). In the case of free text the choice of s-term can be guided by term frequency which can be made available, requiring a small state stored at the client or retrieved from the server. We extend on this topic in the full version [5].

3.2 Oblivious Cross-Tags (OXT) Protocol

The BXT scheme is vulnerable to the following simple attack: When the server gets xtrap_i for a query $(w_1, \ldots, w_n)$, it can save it and later use it to learn if any revealed ind value is a document with keyword w_i by testing if $f(\mathsf{xtrap}_i, \mathsf{ind}) \in \mathsf{XSet}$. This allows an honest-but-curious server to learn, for example, the number of documents matching each queried s-term with each queried x-term (even for terms in different queries). This attack is possible because BXT reveals the keys that enable the server to compute $f(\mathsf{xtrap}_i, \cdot)$ itself.

One way to mitigate the attack is to have the client evaluate the function for the server instead of sending the key. Namely, the server would send all the encrypted ind values that it gets in t (from the TSet) to the client who will compute the function $f(\mathsf{xtrap}_i, \mathsf{ind})$ and send back the results. However, this fix adds a round of communication with consequent latency, it allows the server to cheat by sending ind values from another query's s-term (from which the server can compute intersections not requested by the client), and is unsuited to the multi-client SSE setting [7] discussed in the introduction (since the client would learn from the inds it receives the results of conjunctions it was not authorized for). Note that while the latter two issues are not reflected in our current formal model, avoiding them expands significantly the applicability of OXT.

These issues suggest a solution where we replace the function $f(\mathsf{xtrap}, \cdot)$ (where $\mathsf{xtrap} = F(K_X, w)$) with a form of *oblivious shared computation between client and server.* A first idea would be to use *blinded exponentiation* (as in Diffie-Hellman based oblivious PRF) in a group G of prime order p: Using a PRF F_p with range Z_p^* (and keys K_I, K_X), we derive a value $\mathsf{xind} = F_p(K_I, \mathsf{ind}) \in Z_p^*$ and define each xtag to be $g^{F_p(K_X,w)\cdot \mathsf{xind}}$. The shared computation would proceed by the client first sending the value $g^{F_p(K_X,w_i)\cdot z}$ where $z \in Z_p^*$ is a blinding factor;

EDBSetup(DB)

- Select key K_S for PRF F, keys K_X, K_I, K_Z for PRF F_p (with range in Z_p^*), and parse DB as $(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$.
- Initialize $\mathbf{T}$ to an empty array indexed by keywords from W.
- Initialize XSet to an empty set.
- For each $w \in \mathsf{W}$, build the tuple list $\mathbf{T}[w]$ and XSet elements as follows:
  - Initialize $\mathbf{t}$ to be an empty list, and set $K_e \leftarrow F(K_S, w)$.
  - For all ind in DB(w) in random order, initialize a counter $c \leftarrow 0$, then:
    - Set $\mathsf{xind} \leftarrow F_p(K_I, \mathsf{ind})$, $z \leftarrow F_p(K_Z, w \,\|\, c)$ and $y \leftarrow \mathsf{xind} \cdot z^{-1}$.
    - Compute $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{ind})$, and append (e, y) to $\mathbf{t}$.
    - Set $\mathsf{xtag} \leftarrow g^{F_p(K_X, w) \cdot \mathsf{xind}}$ and add xtag to XSet.
  - $\mathbf{T}[w] \leftarrow \mathbf{t}$.
- $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$.
- Output the key $(K_S, K_X, K_I, K_Z, K_T)$ and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.

Search protocol

- The client's input is the key $(K_S, K_X, K_I, K_Z, K_T)$ and query $\bar{w} = (w_1, \ldots, w_n)$.

  It sends to the server the message $(\mathsf{stag}, \mathsf{xtoken}[1], \mathsf{xtoken}[2], \ldots)$ defined as:
  - $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w_1)$.
  - For $c = 1, 2 \ldots$ and until server sends stop
    - For $i = 2, \ldots, n$, set $\mathsf{xtoken}[c, i] \leftarrow g^{F_p(K_Z, w_1 \,\|\, c) \cdot F_p(K_X, w_i)}$
    - Set $\mathsf{xtoken}[c] = \mathsf{xtoken}[c, 2], \ldots, \mathsf{xtoken}[c, n]$.
- The server has input (TSet, XSet). It responds as follows.
  - It sets $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$.
  - For $c = 1, \ldots, |\mathbf{t}|$
    - retrieve (e, y) from the c-th tuple in $\mathbf{t}$
    - if $\forall i = 2, \ldots, n : \mathsf{xtoken}[c, i]^y \in \mathsf{XSet}$ then send e to the client.
  - When last tuple in $\mathbf{t}$ is reached, send stop to $\mathcal{C}$ and halt.
- Client sets $K_e \leftarrow F(K_S, w_1)$; for each e received, computes $\mathsf{ind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ and outputs ind.

Fig. 2. OXT: Oblivious Cross-Tags Protocol

the server would raise this to the power xind and finally the client would de-blind it by raising to the power z^{-1} mod p to obtain $g^{F_p(K_X, w_i) \cdot \mathsf{xind}}$. Unfortunately, this idea does not quite work as the server would learn $\mathsf{xtag} = g^{F_p(K_X, w_i) \cdot \mathsf{xind}}$ and from this, and its knowledge of xind, it would learn $g^{F_p(K_X, w_i)}$, which allows it to carry out an attack similar to the one against BXT. This also requires client-server interaction on a per-xind basis, a prohibitive cost.

In the design of OXT we address these two problems. The idea is to *precompute* (in EDBSetup) the blinding part of the oblivious computation and store it in the EDB. I.e., in each tuple corresponding to a keyword w and document xind, we store a blinded value $y_c = \mathsf{xind} \cdot z_c^{-1}$, where z_c is an element in Z_p^* derived (via a PRF) from w and a tuple counter c (this counter, incremented for each tuple in $\mathbf{t}$ associated with w, serves to ensure independent blinding values z).

During search, the server needs to compute the value $g^{F_p(K_X,w_i)\cdot \mathsf{xind}}$ for each xind corresponding to a match for w_1 and then test these for membership in XSet. To enable this, the client sends, for the c-th tuple in $\mathbf{t}$, a n-long array $\mathsf{xtoken}[c]$ defined by $\mathsf{xtoken}[c, i] := g^{F_p(K_X,w_i)\cdot z_c}$, $i = 1, \ldots, n$, where z_c is the precomputed blinding derived by from w (via a PRF) and the tuple counter c. The server then performs the T-set search to get the results for w_1, and filters the c-th result by testing if $\mathsf{xtoken}[c, i]^{y_c} \in \mathsf{XSet}$ for all $i = 2, \ldots, n$. This protocol is correct because $\mathsf{xtoken}[c, i]^{y_c} = g^{F_p(K_X,w_i)\cdot z_c \cdot \mathsf{xind} \cdot z_c^{-1}} = g^{F_p(K_X,w_i)\cdot \mathsf{xind}}$, meaning that the server correctly recomputes the pseudorandom values in the XSet.

Putting these ideas together results in the OXT protocol of Figure 2. Note that the client sends the xtoken arrays (each holding several values of the form $g^{F_p(K_X,w_i)\cdot z}$) until instructed to stop by the server. There is no other communication from server to client (alternatively, server can send the number of elements in $\mathsf{TSet}(w)$ to the client who will respond with such number of xtoken arrays).[6]

Note that while the description above is intended to provide intuition for the protocol's design, assessing the security (leakage) of OXT is non-trivial, requiring an intricate security analysis that we provide in Section 4.

OXT consists of a single round of interaction, where the message sent by the client is of size proportional to $|\mathsf{DB}(w_1)|$, [7] and the response to the client is minimal, consisting only of the result set (i.e., the set of encrypted ind's matching the query). The computational cost of OXT lies in the use of exponentiations, however, thanks to the use of very efficient elliptic curves (we only require the group to be DDH) and *fixed-base exponentiations*, this cost is practical even for very large databases as demonstrated by the performance numbers in Section 5.

OXT leaks much less information to the server than BXT. Indeed, since the server, call it $\mathcal{S}$, learns neither the ind values nor $\mathsf{xtrap}_j, j = 2, \ldots, n$, its ability to combine conjunctive terms from one query with terms from another query is significantly reduced. In particular, while in BXT $\mathcal{S}$ learns the intersection between s-terms of any two queries, in OXT this is possible only in the following case: the two queries can have different s-terms, but same x-term and there is a document containing both s-terms (the latter is possible since if the s-terms of two queries share a document ind and an x-term xtrap then the xtag value $f(\mathsf{xtrap}, \mathsf{ind})$ will be the same in both queries indicating that ind and xtrap are the same). The only other leakage via s-terms is that $\mathcal{S}$ learns when two queries have the same s-term w_1 and the size of the set $\mathsf{DB}(w_1)$. Finally, regrading intra-query leakage if $\mathcal{C}$ responds with the values $\mathsf{xtag}_j, j = 2, \ldots, n$, in the same order for all ind's, then in case $n > 2$, $\mathcal{S}$ learns the number of documents matching any sub-conjunction that includes w_1 and any subset of $w_2, \ldots, w_n$. If, instead, $\mathcal{C}$

[6] The same protocol supports single-keyword search (or 1-term conjunctions) by skipping the $c = 1, 2, \ldots$ at both client and server, hence falling back to the SKS protocol of Figure 1.

[7] For typical choices of w_1, such message will be of small or moderate size. For large values of $|\mathsf{DB}(w_1)|$ one can cap the search to the first k tuples for a threshold k, say 1000. For example, in the case of a 3-term conjunction and xtag values of size 16 bytes, this will result in just 32 Kbyte message.

randomly permutes the values $\mathsf{xtag}_j, j = 2, \ldots, n$ before sending these values to $\mathcal{S}$, then $\mathcal{S}$ learns the maximal number of satisfied terms per tuple in $\mathsf{TSet}(w_1)$, but not the size of sets matching $w_1 \wedge w_i, i = 1, \ldots, n$, or any other proper sub-conjunctions (except for what can be learned in conjunction with other leakage information). *In Section 4 we formally analyze the security of* OXT *making the above description of leakage precise.*

As noted before, even a leakage profile as the above that only reveals access patterns can still provide valuable information to an attacker that possesses prior information on the database and queries. We don't discuss here specific countermeasures for limiting the ability of an attacker to perform such statistical inference – see [15] for an example of potential masking techniques.

3.3 Processing Boolean Queries with OXT

We describe an extension to OXT that can handle *arbitrary* Boolean query expressions. We say that a Boolean expression in n terms is in *Searchable Normal Form* (SNF) if it is of the form $w_1 \wedge \phi(w_2, \ldots, w_n)$ where ϕ is an *arbitrary* Boolean formula (e.g., "$w_1 \wedge (w_2 \vee w_3 \vee \neg w_4)$"). OXT can be extended to answer such queries: On input a query of the form $w_1 \wedge \phi(w_2, \ldots, w_n)$, the client creates a modified boolean expression $\hat{\phi}$ in new boolean variables v_i $(i = 2, \ldots, n)$, which is just ϕ but with each w_i replaced by v_i. Thus, the client uses w_1 as the s-term and computes its stag as in OXT, and computes the xtrap (i.e. the xtoken array) for all the other terms w_i $(i > 1)$. It then sends the stag and the xtraps in the order of their index. It also sends the server the above modified boolean expression $\hat{\phi}$. The server fetches the TSet corresponding to the stag as in OXT. It also computes the xtag corresponding to each x-term, also as in OXT. But, it decides on sending (to the Client) the encrypted ind corresponding to each tuple in the TSet based on the following computation (which is the only different part from OXT): for each $i = 2, \ldots, n$, the server treats the variable v_i as a boolean variable and sets it to the truth value of the expression $(\mathsf{xtoken}[c, i])^y \in \mathsf{XSet}$. Then it evaluates the expression $\phi(v_2, \ldots, v_n)$. If the result is true, it returns the e value in that tuple to the Client.

OXT can also be used to answer any disjunction of SNF expressions. Actually, note that OXT can answer *any* Boolean query by adding to the database a field TRUE which all documents satisfy. Then a search for any expression $\phi(w_1, \ldots, w_n)$ can be implemented as "TRUE $\wedge\, \phi(w_1, \ldots, w_n)$", which is in SNF and can be searched as in the SNF case above. Clearly, this will take time linear in the number of documents but it can be implemented if such functionality is considered worth the search complexity.

4 Security of OXT

In the full version [5] we provide a complete detailed analysis of OXT and its extension to Boolean queries. Due to space constraints we illustrate the security claim for the particular case of two-term conjunctions, but this restricted case is

representative of the general case. We start by describing the protocol's leakage profile, $\mathcal{L}_{\text{oxt}}(\mathsf{DB}, \mathbf{q})$, followed by a security theorem showing that this is *all* of the information leaked by the protocol.

In describing the leakage profile of the OXT protocol we will assume an *optimal* T-set implementation (see Section 2) as the one presented in [5], namely, with optimal leakage $N = \sum_{i=1}^{d} |\mathsf{W}_i|$. We represent a sequence of Q two-term conjunction queries by $\mathbf{q} = (\mathbf{s}, \mathbf{x})$ where an individual query is a two-term conjunction $\mathbf{s}[i] \wedge \mathbf{x}[i]$ which we write as $\mathbf{q}[i] = (\mathbf{s}[i], \mathbf{x}[i])$.

We define $\mathcal{L}_{\text{oxt}}(\mathsf{DB}, \mathbf{q})$, for $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^{d}$ and $\mathbf{q} = (\mathbf{s}, \mathbf{x})$, as a tuple $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{RP}, \mathsf{IP})$ formed as follows:

- $N = \sum_{i=1}^{d} |\mathsf{W}_i|$ is the total number of appearances of keywords in documents.
- EP is the *equality pattern* of $\mathbf{s} \in \mathsf{W}^Q$ indicating which queries have the equal s-terms. Formally, $\mathsf{EP} \in [m]^Q$ is formed by assigning each keyword an integer in $[m]$ determined by the order of appearance in $\mathbf{s}$. For example, if $\mathbf{s} = (a, a, b, c, a, c)$ then $\mathsf{EP} = (1, 1, 2, 3, 1, 3)$. To compute $\mathsf{EP}[i]$ one finds the least j such that $\mathbf{s}[j] = \mathbf{s}[i]$ and then lets $\mathsf{EP}[i] = |\{\mathbf{s}[1], \ldots, \mathbf{s}[j]\}|$ be the number of unique keywords appearing at indices less than or equal to j.
- SP is the *size pattern* of the queries, i.e. the number of documents matching the first keyword in each query. Formally, $\mathsf{SP} \in [d]^Q$ and $\mathsf{SP}[i] = |\mathsf{DB}(\mathbf{s}[i])|$.
- RP is the *results pattern*, which consists of the results sets $(R_1, \ldots, R_Q)$, each defined by $R_i = I_\pi(\mathbf{s}[i]) \cap I_\pi(\mathbf{x}[i])$.
- IP is the *conditional intersection pattern*, which is a Q by Q table with entries defined as follows: $\mathsf{IP}[i, j]$ is an empty set if either $\mathbf{s}[i] = \mathbf{s}[j]$ or $\mathbf{x}[i] \neq \mathbf{x}[j]$. However, if $\mathbf{s}[i] \neq \mathbf{s}[j]$ and $\mathbf{x}[i] = \mathbf{x}[j]$ then $\mathsf{IP}[i, j] = \mathsf{DB}(\mathbf{s}[i]) \cap \mathsf{DB}(\mathbf{s}[j])$.

Understanding the Leakage Components. The parameter N can be replaced with an upper bound given by the total size of EDB but leaking such a bound is unavoidable. The equality pattern EP leaks repetitions in the s-term of different queries; this is a consequence of our optimized search that singles out the s-term in the query. This leakage can be mitigated by having more than one TSet per keyword and the client choosing different incarnations of the Tset for queries with repeated s-terms. SP leaks the number of documents satisfying the s-term in a query and is also a direct consequence of our approach of optimizing search time via s-terms; it can be mitigated by providing an upper bound on the number of documents rather than an exact count by artificially increasing Tset sizes. RP is a the result of the query and therefore no real leakage. Finally, the IP component is the most subtle and it means that if two queries have different s-terms but same x-term, then the indexes which match both the s-terms are leaked (if there are no documents which match both s-terms then nothing is leaked). This "conditional intersection pattern" can be seen as the price for the rich functionality enabled by our x-terms and XSet approach that allows for the computation of arbitrary boolean queries. Note that on query $\mathbf{q}[i] = (\mathbf{s}[i] \wedge \mathbf{x}[i])$ the OXT protocol lets the server compute a deterministic function $f(\mathsf{xtrap}(\mathbf{x}[i]), \mathsf{ind})$ of the x-term $\mathbf{x}[i]$ for all ind's that match the s-term $\mathbf{s}[i]$. Therefore a repeated xtag value in two queries $\mathbf{q}[i]$ and $\mathbf{q}[j]$ implies

that $\mathbf{x}[i] = \mathbf{x}[j]$ and that $\mathsf{DB}(\mathbf{s}[i])$ and $\mathsf{DB}(\mathbf{s}[j])$ contain a common index. Even though this index is not revealed to the server, we still model this information by simply revealing $\mathsf{DB}(s[i]) \cap \mathsf{DB}(s[j])$ if $\mathbf{x}[i] = \mathbf{x}[j]$. This "pessimistic" upper bound on the leakage simplifies the leakage representation. As said, if the above intersection is empty then no information about the equality of the x-terms $\mathbf{x}[i], \mathbf{x}[j]$ is revealed. The probability of non-empty intersections is minimized by consistently choosing low-frequent s-terms. Note also that for queries with s-terms belonging to the same field with unique per-document value (e.g., both s-terms containing different last names in a database with a last-name field), the IP leakage is empty.

Theorem 1. *The SSE scheme* OXT *implemented with an optimal T-set is* $\mathcal{L}_{\mathrm{oxt}}$*-semantically-secure if all queries are 2-conjunctions, assuming that the DDH assumption holds in G, that F and F_p are secure PRFs, and* $(\mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA secure symmetric encryption scheme.*

Proof Sketch. The proof of the theorem is delicate and lengthy, and is presented in [5] for the general case of multi-term conjunctions (with extensions to the case of Boolean queries). To get some intuition for why the scheme is secure, we start by examining why each of the outputs of $\mathcal{L}$ is necessary for a correct simulation. Of course, this does nothing to show that they are *sufficient* for simulation, but it will be easier to see why this is all of the leakage once their purpose is motivated. For the sake of this sketch we assume a non-adaptive adversary.

The size of the XSet is equal to the value N leaked. The equality pattern EP (or something computationally equivalent to it) is necessary due to the fact that the stag values are deterministic, so a server can observe repetitions of stag values to determine if $\mathbf{s}[i] = \mathbf{s}[j]$ for all i, j. The size pattern is also necessary as the server will always learn the number of matches for the first keyword in the conjunction by observing the number of tuples returned by the T-set. We include the results pattern to enable the simulator to produce the client results for queries in way consistent the conditional intersection pattern.

The final and most subtle part of the leakage is the conditional intersection pattern IP. The IP is present in the leakage because of the following passive attack. During the computation of the search protocol, the values tested for membership in the XSet by the server have the form $f(\mathsf{xtrap}(w_i), \mathsf{ind}) = g^{F_p(K_X, w_i) \cdot F_p(K_I, \mathsf{ind})}$, where w_i is the i-th x-term in a search query and ind is an identifier for a document that matched the s-term in that query. As we explain above, the leakage comes from the fact that f is a deterministic function, and so these values will *repeat* if and only if (except for a negligible probability of a collision) (1) the two queries involve the same x-term and (2) the sets of indexes which match the s-terms involved in these queries have some indexes in common.

Our proof makes formal the claim that the output of $\mathcal{L}$ is sufficient for a simulation. We outline a few of the technical hurdles in the proof without dealing with the details here. For this discussion, we assume that reductions to PRF security and encryption security go through easily, allowing us to treat PRF outputs as random and un-opened ciphertexts as encryptions of zeros.

We first handle the information leaked by the XSet. An unbounded adversary could compute the discrete logarithms of the XSet elements and derive information about which documents match which keywords. We want to show however that a poly-time adversary learns nothing from the XSet due to the assumed hardness of the DDH problem. Formally, we need to show that we can replace the elements of XSet with random elements that carry no information about the database, but there is a technical difficulty: some of the exponents (specifically, the xind values) that will play the roll of hidden exponents in the DDH reduction are used in the computation of the xtrap values, and these are revealed in the transcripts. A careful rearrangement of the game computation will show that this is not as bad as it seems, because the xind values are "blinded out" by the z values. We stress that this requires some care, because the z values are also used twice, and we circumvent this circularity by computing the XSet first and then computing the transcripts "backwards" in way that is consistent with the XSet. Now a reduction to DDH becomes clear, as the XSet values can be dropped in obliviously as real-or-random group elements.

With the XSet leakage eliminated, the rest of the work is in showing that the simulator can arrange for a correct-looking pattern of "repeats" in the documents matched and in the values tested against the XSet. While riddled with details, this is intuitively a rather straightforward task that is carried out in the latter games of the proof.

5 OXT Implementation and Experimental Results

This section reports the status of the OXT implementation and several latency and scalability measurements, which should be viewed as providing empirical evidence to the performance and scalability claims made earlier in the paper. Additional details on the implementation can be found in the full version [5].

Prototype. The realization of OXT consists of EDBSetup, which generates the EDB, Client, which generates the stag and the xtoken stream, and EDBSearch, which uses the EDB to process the Client's request. All three use the same cryptographic primitives, which leverage the OpenSSL 1.0.1 library. As the DH groups we use NIST 224p elliptic curve. The overall C code measures roughly 16k lines.

To scale beyond the server's RAM, the TSet is realized as a disk-resident paged hash table. Each tuple list $\mathbf{T}[w]$ in the TSet is segmented into fixed-size blocks of tuples keyed by a tag stag_c. This tag is derived by a PRF from the list's stag and a segment counter. EDBSearch uses page-level direct I/O to prevent buffer cache pollution in the OS, as the hash table pages are inherently uncachable, and parallelizes disk accesses using asynchronous I/O (`aio_*` system calls). The XSet is realized as a RAM-resident Bloom filter [2], which enables the sizing the false positive rate to the lowest value that the server's RAM allows. For the experiments presented next, the false positive rate is 2^{-20} and the Bloom filter XSet still occupies only a small fraction of our server's RAM.

Data Sets. To show the practical viability of our solution we run tests on two data sets: the Enron email data set [9] with more than 1.5 million documents

(email messages and attachments) which generate 145 million distinct (keyword, docId) pairs, and the ClueWeb09 [20] collection of crawled web-pages from which we extracted several databases of increasing size, where the largest one is based on 0.4TB of HTML files which generate almost 3 billion distinct (keyword, docId) pairs. For the Enron email data, the TSet hash table and the XSet Bloom filter are 12.4 GB and 252 MB, respectively. The corresponding sizes for the largest ClueWeb09 data set are 144.4 GB and 9.7 GB, respectively.

Experimental Results. All experiments were run on IBM Blades HS22 attached to a commodity SAN system. Figure 3 shows the latency of queries on one-term, called v, and three variants of two-term conjunctive queries on the Enron data set. In one-term queries, the selectivity of v (the number of documents matching v) varies from 3 to 690492. As this query consists only of an s-term, the figure illustrates that its execution time is linear in the cardinality of $\mathsf{TSet}(v)$. The two-term conjunctive queries combine the previous queries with a fixed reference term. In the first of these queries, the fixed term acts as an x-term: each tuple retrieved from the TSet is checked against the XSet at the cost of an exponentiation. However, as we perform these operation in parallel to retrieving the TSet buckets from the disk, their cost is completely hidden by the disk I/O latency. Micro-benchmarks show that the average cost of retrieving a bucket which has a capacity of 10 tuples is comparable to $\sim$1,000 single-threaded exponentiations. Similarly, the client-side exponentiation in the OXT protocol can be overlapped with disk and network I/O. It illustrates the fact that exponentiations, over fast elliptic curves, are relatively cheap when compared to the cost of accessing storage systems. The last two conjunctive queries use two fixed terms with different selectivity, α and β, as s-terms. Their invariable execution time is dominated by the cost of retrieving the TSet tuples corresponding to their s-terms, irrespective the variable selectivity of the xterm v: the two horizontal lines intersect with the single-term curve exactly where v corresponds to α and β, respectively. This illustrates the importance of s-term selection, as discussed in Section 3.1.

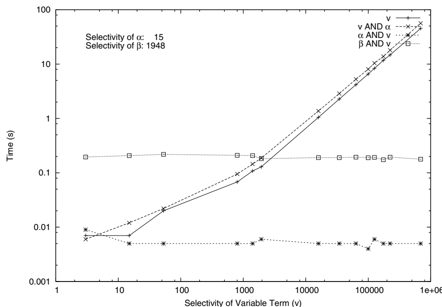

Fig. 3. Enron Performance Measurement: Single Term & Conjunction

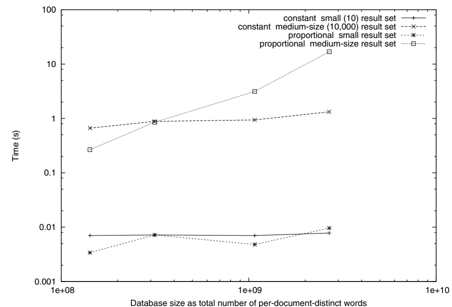

Fig. 4. Clueweb09 Performance Measurement: Scaling Database Size

To further assess the scalability of EDBSearch, we generated several EDBs from increasingly larger subsets of the ClueWeb09 data set ranging from 408, 450 to 13, 284, 801 HTML files having a size from 20 to 410 GBs and from 142, 112, 027 to 2, 689, 262, 336 distinct (keyword, docId) pairs. To make these databases comparable, we injected several artificial and non-conflicting keywords to randomly selected documents simulating words of various selectivity. Figure 4 confirms that our implementation matches the scalability of our (theoretical) algorithms even when our databases exceed the size of available RAM: If the size of the result set is constant, then query time is largely independent of the size of the database and for result sets where the size is proportional to the database size, the cost is linear in the database size.

6 Conclusions and Research Questions

The premise of this work is that in order to provide truly practical SSE solutions one needs to accept a certain level of leakage; therefore, the aim is to achieve an acceptable balance between leakage and performance, with formal analysis ensuring upper bounds on such leakage. Our solutions strike such a practical balance by offering performance that scales to very large data bases; supporting search in both structured and textual data with general Boolean queries; and confining leakage to access (to encrypted data) patterns and some query-term repetition only, with formal analysis defining and proving the exact boundaries of leakage. We stress that while in our solutions leakage never occurs in the form of direct exposure of plain data or searched values, when combined with side-information that the server may have (e.g., what are the most common searched words), such leakage can allow for statistical inference on plaintext data. Nonetheless, it appears that in many practical settings the benefits of search would outweigh moderate leakage (especially given the alternatives of outsourcing the plaintext data or keeping it encrypted but without the ability to run useful searches).

Our report on the characteristics and performance of our prototype points to the fact that scalability can only be achieved by low-complexity protocols which admit highly parallelizable implementations of their computational and I/O paths. Our protocols are designed to fulfill these crucial performance requirements.

There are interesting design and research challenges arising from this work. What we call “the challenge of being imperfect” calls for trade-offs between privacy and performance that can only be evaluated on the basis of a formal treatment of leakage. Understanding the limits of what is possible in this domain and providing formal lower bounds on such trade-offs appears as a non-trivial problem that deserves more attention. Some of these problems may still be unresolved even for plaintext data. The seemingly inherent difference pointed out in the introduction between the complexity of resolving conjunctions with high-frequency terms versus conjunctions with low-frequency terms, but with a similar-size result set, may be such a case. We do not know of a proven lower

bound in this case although the work of Patrascu [22], for example, may point to some relevant conjectured bounds.

Another important evaluation of leakage is what we refer to as "semantic leakage." How much can an attacker learn from the data given the formal leakage profile and side knowledge on the plaintext data? Clearly, the answer to this question is application-dependent but one may hope for some general theory in which these questions can be studied. The success of differential privacy in related domains opens some room for optimism in this direction. Demonstrating specific attacks in real-world settings is also an important direction to pursue. We note that in some settings just revealing the size of the number of documents matching a query may leak important information on the query contents (e.g., [15]). Therefore, developing masking techniques that include dummy or controlled data to obscure statistical information available to the attacker seems as an important research direction to strengthen the privacy of solutions as those developed here. ORAM-related techniques can be certainly help in this setting, especially given the progress on the practicality of these techniques in last years.

Yet another research direction is to characterize plaintext-search algorithms that lend themselves for adaptation to the encrypted setting. The s-term and x-term based search that we use is such an example: It treats data in "blackbox" form that translates well to the encrypted setting. In contrast, search that looks at the data itself (e.g., sorting it) may not work in this setting or incur in significantly increased leakage (e.g., requiring order-preserving or deterministic encryption). Finally, it would be interesting to see more examples (in other two-party, or multi-party, protocols) of our approach, which is central to the design of OXT, of removing interaction from protocols by pre-computing and storing some of the protocol messages during a pre-computation phase.

Acknowledgment. Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI / NBC) contract number D11PC20201. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

References

1. Ballard, L., Kamara, S., Monrose, F.: Achieving efficient conjunctive keyword searches over encrypted data. In: Qing, S., Mao, W., López, J., Wang, G. (eds.) ICICS 2005. LNCS, vol. 3783, pp. 414–426. Springer, Heidelberg (2005)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the Association for Computing Machinery 13(7), 422–426 (1970)
3. Byun, J.W., Lee, D.-H., Lim, J.-I.: Efficient conjunctive keyword search on encrypted data storage system. In: Atzeni, A.S., Lioy, A. (eds.) EuroPKI 2006. LNCS, vol. 4043, pp. 184–196. Springer, Heidelberg (2006)

4. Cash, D., Jagger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation (manuscript, 2013)
5. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Highlyscalable searchable symmetric encryption with support for boolean queries. Report 2013/169, Cryptology ePrint Archive (2013), http://eprint.iacr.org/2013/169
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. 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 2006, pp. 79–88. ACM Press (October 2006)
9. EDRM (edrm.net). Enron data set, http://www.edrm.net/resources/data-sets/edrm-enron-email-data-set-v2
10. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC, pp. 169–178. ACM Press (May/June 2009)
11. Goh, E.-J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003), http://eprint.iacr.org/
12. Goldwasser, S., Ostrovsky, R.: Invariant signatures and non-interactive zeroknowledge proofs are equivalent (extended abstract). In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 228–245. Springer, Heidelberg (1993)
13. Golle, P., Staddon, J., Waters, B.: Secure conjunctive keyword search over encrypted data. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 31–45. Springer, Heidelberg (2004)
14. IARPA. Security and Privacy Assurance Research (SPAR) Program - BAA (2011), http://www.iarpa.gov/solicitations_spar.html/
15. Islam, M., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2012), San Diego, CA. Internet Society (February 2012)
16. Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.C., Steiner, M.: Outsourced symmetric private information retrieval (manuscript 2013)
17. Kamara, S., Lauter, K.: Cryptographic cloud storage. In: Sion, R., Curtmola, R., Dietrich, S., Kiayias, A., Miret, J.M., Sako, K., Sebé, F. (eds.) RLCPS, WECSR, and WLC 2010. LNCS, vol. 6054, pp. 136–149. Springer, Heidelberg (2010)
18. Kamara, S., Papamanthou, C., Röder, T.: CS2: A searchable cryptographic cloud storage system (2011), http://research.microsoft.com/pubs/148632/CS2.pdf
19. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proc. of CCS 2012 (2012)
20. Lemur Project. ClueWeb09 dataset, http://lemurproject.org/clueweb09.php/
21. Pappas, V., Vo, B., Krell, F., Choi, S.G., Kolesnikov, V., Keromytis, A., Malkin, T.: Blind Seer: A Scalable Private DBMS (manuscript, 2013)
22. Patrascu, M.: Towards polynomial lower bounds for dynamic problems. In: 42nd ACM STOC, pp. 603–610. ACM Press (2010)
23. 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 (May 2000)