Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries

David Cash* Stanislaw Jarecki† Charanjit Jutla‡
Hugo Krawczyk§ Marcel Rosu¶ Michael Steiner‖

Abstract

This work presents the design, analysis and implementation of the first sub-linear searchable symmetric encryption (SSE) protocol that supports conjunctive search and general Boolean queries on symmetrically-encrypted data and that scales to very large data sets 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). A key aspect of our protocols is that it allows the searcher to pivot its conjunctive search on the estimated least frequent keyword in the conjunction. We show that a decisional Diffie-Hellman (DDH) based pseudo-random function can be used not just to implement search tokens but also to hide query access pattern of non-pivot, and hence possibly highly frequent, keywords in conjunctive queries. We present a 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.

*Rutgers U. Email: david.cash@cs.rutgers.edu.
†U. California Irvine. Email: stasio@ics.uci.edu.
‡IBM Research. Email: csjutla@us.ibm.com
§IBM Research. Email: hugo@ee.technion.ac.il
¶IBM Research. Email: rosu@us.ibm.com
‖IBM Research. Email: msteiner@us.ibm.com

Contents

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.

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 that includes the original data in encrypted form along with additional encrypted metadata that is then stored on the 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 a single owner of the data that writes to the database – the public key variant of the problem has also been extensively studied, see below).

An important line of research (e.g., [30, 18, 12, 14, 13, 25]) gives practical constructions of SSE that support searching for documents that contain a *single* specified keyword. In these schemes, the server's work in a search is independent of the size of the database, scaling with the number of documents matching the query. Leakage to the server in these schemes is limited to the set of returned (encrypted) documents and some global parameters of the system such as total 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 larger data sets, a single-keyword search will often return a large number of documents that the user must then download and filter itself to find the relevant results.

Conjunctive 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 intersecion 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. All other known solutions require server's 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. [20], with variants presented in later work, e.g., [3, 11]. 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 total 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 text search.

The challenge of large datasets and the challenge of being imperfect. In this work we investigate solutions to conjunctive queries and more general Boolean queries that can be prac-

tical even for very large datasets where linear search is prohibitively expensive. Our application settings include databases that require search over tens of millions records and documents, with both attribute-value as well as textual search (see below for specific numbers used in testing 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"), and 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, would 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. 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, computational efficiency for processing the databases and individual searches is of paramount importance for a practical solution. The use of generic tools such as fully homomorphic encryption [17] or oblivious RAM [19] are out of the question, although they may be used as sub-components of a solution if applied to small subsets of data. Even the use of techniques based on pairings as in [20] would be too slow for such aplications.

Our Contributions. We develop the first *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 leakage, but with a much better leakage profile than the existing sublinear solutions 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. [14] and Chase and Kamara [13], and assumes an adaptive adversarial model. The essence of the security notion is that the view of the attacker can be efficiently simulated given a precisely-defined *leakage profile* but without access to the actual data. Such a profile may include leakage on sizes of data sets (e.g., the total size of a 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 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[1].

The characterization of leakage and the involved proof of security that we present, are central

[1] An important direction of future research is to develop a theory that allows to study the amount of "semantic leakage" on data based on formal leakage profiles as those we develop, taking into account the specifics of an application and possible side information on the data available to the attacker.

technical contributions of our work that complement the protocol design work.

The centerpiece of the design is a "virtual" secure two-party 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 targetted 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 uses only DH-type operations over any Diffie-Hellman group which enables the use of the fastest DH elliptic curves (with additional common-base optimizations).[2] The overall complexity of our solutions 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. In addition, the space requirements of our schemes - a main consideration for large databases where disk access often dominates the performance cost - is comparable to the storage requirements of the best existing single-keyword search solutions. All our solutions support search on structured data (e.g., attribute-value databases) as well as on free text, and combinations of both.

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 formula ϕ on the remaining keywords)[3]. The search complexity is proportional to the number of document that contain w_1. Any disjunction of the above forms can also be answered with an additive cost over the disjunction terms. 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.

Further extensions. Our protocols can also be applied to an important emerging setting [23, 13, 24] 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 (this is referred to as "controlled disclosure" in [13]). 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 future work, we intend to 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 [21]). Finally, our protocols can support database updates as long as the server is allowed to get some leakage differentiating between old and new documents. This and other forms of leakage can be mitigated to some extent via "masking techniques" that include dummy

[2] We also present a scheme that only uses symmetric-key operations but provides less privacy, and a pairing-based scheme that optimizes communication at the expense of more computation.

[3] 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 {Rivest, Shamir, Adelman}.

or controlled data to obscure statistical information available to the attacker. See [22] for some relevant techniques.

Implementation. To show the practical viability of our solution we implemented a prototype of our main protocol (OXT) and run tests on three data sets: a 100,000 record relational database synthesized from census data; the Enron email data set [15] with more than 1.5 million documents (email messages and attachments) where all words, including attachments and envelope information, have been indexed resulting in about 1.2 million distinct words; and the ClueWeb09 [27] collection of crawled web-pages from which we extracted several databases of increasing size where the largest one was based on 0.4TB of HTML files with almost 3 billion of per-document unique words. 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 medium-size sets as the Enron case. Refer to Appendix F for more information on implementation and performance.

Related work. See Appendix E on Page 41 for more discussion in related work.

2 Definitions and Tools

Notation. We write $[n]$ for the set $\{1, \ldots, n\}$. For a vector $\mathbf{v}$ we write $|\mathbf{v}|$ for the dimension (length) of $\mathbf{v}$ and for $i \in [|\mathbf{v}|]$ we write $\mathbf{v}[i]$ for the i-th component of $\mathbf{v}$. All algorithms (including adversaries) are assumed to be randomized polynomial-time unless otherwise specified. If A is an algorithm, then $y \leftarrow A(x)$ means that the y is the output of A when run on input x. If A is randomized then y is a random variable. For sets X, Y we write $\mathsf{Fun}(X, Y)$ for the set of all functions from X to Y, and $\mathsf{Perm}(X)$ for the set of all permutations on X.

2.1 SSE Syntax and Security Model

Searchable symmetric encryption. A database is composed of a collection of d documents, each identified with a set of keywords W_i. The output from the SSE protocol for a given search query are *randomized indices* rind corresponding to the documents that satisfy the query. A client program can then use these indices to retrieve the payload documents. 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 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 $\mathsf{W} = \bigcup_{i=1}^d \mathsf{W}_i$. 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 records in DB, $m = |\mathsf{W}|$ and $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.

A *searchable symmetric encryption (SSE) scheme* Π consists of two algorithms $\mathsf{EDBSetup}$, $\mathsf{GetRind}$ 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 randomized identifiers and the server has no output. The deterministic

algorithm $\mathsf{GetRind}$ takes as input the key K and an identifier $\mathsf{ind} \in \{0,1\}^\lambda$ and outputs another identifier, denoted rind. We require that $\mathsf{GetRind}(K,\cdot)$ induce a permutation on $\{0,1\}^\lambda$ for any key K output by $\mathsf{EDBSetup}$. We say that an SSE scheme is *correct* if for all inputs DB and queries $\psi(\bar{w}) \in \mathsf{W}^*$, if $(K, \mathsf{EDB}) \xleftarrow{\$} \mathsf{EDBSetup}(\mathsf{DB})$, after running Search with client input $(K, \bar{w})$ and server input EDB, the client outputs the set $\{\mathsf{GetRind}(K, \mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(\psi(\bar{w}))\}$, i.e., the correct result ind values permuted by the function $\mathsf{GetRind}(K,\cdot)$. We consider a computational relaxation of this notion, expressed via the following game. For an adversary A and an SSE scheme Σ, we define the game $\mathbf{Cor}_A^\Pi(\lambda)$, which lets A choose DB, generates $(K, \mathsf{EDB}) \leftarrow \mathsf{EDBSetup}(\mathsf{DB})$, gives EDB to A, which adaptively chooses queries $\psi(\bar{w})$, for each of which the game runs the Search protocol with client input $(K, \psi(\bar{w}))$ and server input EDB. If in any execution the client outputs something other than $\{\mathsf{GetRind}(K, \mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(\psi(\bar{w}))\}$, the game outputs 1, and otherwise it outputs 0. We say that a scheme Π is *computationally correct* if for all efficient adversaries A, $\Pr[\mathbf{Cor}_A^\Pi(\lambda) = 1] \leq \mathrm{neg}(\lambda)$.

Security of SSE. We recall the semantic security definitions from [14, 13]. The definition is parametrized by a *leakage function* $\mathcal{L}$, which describes what an adversary is allowed to learn about the database and queries when interacting with a secure scheme.

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

$\mathbf{Real}_A^\Pi(\lambda)$ *:* $A(1^\lambda)$ *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.*

$\mathbf{Ideal}_{A,S}^\Pi(\lambda)$ *: 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* $\Pr[\mathbf{Real}_A^\Pi(\lambda) = 1] - \Pr[\mathbf{Ideal}_{A,S}^\Pi(\lambda) = 1] \leq \mathrm{neg}(\lambda)$.

We note that in the security analysis of our SSE schemes we include the client's output, the set of rind values $\{\mathsf{GetRind}(K, \mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(\psi(\bar{w}))\}$, in the adversary's view in the real game, to model the fact that these rind's will be used for retrieval of encrypted record payloads. In Appendix B we include a version of this security notion to *non-adaptive* adversaries.

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, 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. [14, 13], can be seen as giving a specific T-set instantiation and using it as an inverted index to enable search – see Section 2.3 for details. 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. One specific T-set implementation is shown in Appendix C.

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. In other words, for any function $n(\lambda)$ of the security parameter λ, for each $w \in \mathsf{W}$, $\mathbf{T}[w]$ is a list $\mathbf{t} = (s_1, \ldots, s_{T_w})$ of strings s.t. each s_i is of length $n(\lambda)$, but $T_w = |\mathbf{T}[w]|$ may vary amongst the w. (In our applications of T-set, $\mathbf{T}[w]$ will contain one tuple per each DB document which matches w, i.e. $T_w = |\mathsf{DB}(w)|$.) 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. As in the case of SSE, we consider a computational relaxation of this perfect correctness notion: For adversary A and a T-set implementation Σ, we define a game $\mathbf{AdvCor}^{\Pi}_{A}(\lambda)$, where A chooses $\mathbf{T}$, the game generates $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$, gives TSet to A, which adaptively chooses keywords w, for each of which the game generates $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w)$ and $\mathbf{t}_w \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$. The game outputs 1 if for any w we have $\mathbf{t}_w \neq \mathbf{T}[w]$, and 0 otherwise. We say that a T-set implementation Π is *computationally correct* if for all efficient adversaries A, $\mathbf{AdvCor}^{\Pi}_{A}(\lambda)$, defined as the probability that the above game outputs 1, is a negligible function of λ.

T-Set security. 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.) Since the list of tuples associated to searched keywords can be seen as information provided to the server, this information is provided to the simulator in the security definition below.

We parametrize the T-set security definition with a leakage function $\mathcal{L}_T$ that describes what else the adversary is allowed to learn by looking at the TSet and stag values. For most implementations this leakage will reveal something about the structure of $\mathbf{T}$, and consequently also the structure of DB. For example, an implementation could reveal the size of $\mathbf{T}$, which is the number of keywords in DB, or the length of each list $\mathbf{T}[w]$ in $\mathbf{T}$, which reveals the number of occurrences of each keyword w in DB. Our more careful implementation given in Appendix C can be shown to leak significantly less, namely only $N = \sum_{w \in \mathsf{W}} |\mathbf{T}[w]|$, the total number of keyword occurences in DB.

Definition 2 *Let $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$ be a T-set implementation, and let A, S be an adversary and a simulator, and let $\mathcal{L}_T$ be a stateful algorithm. We define two games, $\mathbf{Real}^{\Sigma}_{A}$ and $\mathbf{Ideal}^{\Sigma}_{A}$ as follows.*

$\mathbf{Real}^{\Sigma}_{A}(\lambda)$ *: $A(1^\lambda)$ outputs $\mathsf{W}, \mathbf{T}$ with the above syntax. The game computes $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$ and gives TSet to A. Then A repeatedly issues queries $q \in \mathsf{W}$, and for each q the game gives $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K, q)$ to A. Eventually A outputs a bit which the game uses as its output.*

$\mathbf{Ideal}^{\Sigma}_{A,S}(\lambda)$ *: The game initializes a counter $i = 0$ and an empty list $\mathbf{q}$. $A(1^\lambda)$ outputs $\mathsf{W}, \mathbf{T}$ as above. The game runs $\mathsf{TSet} \leftarrow S(\mathcal{L}_T(\mathbf{T}))$ and gives TSet to A. Then A repeatedly issues queries $q \in \mathsf{W}$, and for each q the game stores q in $\mathbf{q}[i]$, increments i, and gives to A the output of $S(\mathcal{L}_T(\mathbf{T}, \mathbf{q}), \mathbf{T}[q])$. Eventually A outputs a bit which the game uses as its output.*

We say that Σ *is a* $\mathcal{L}_T$-adaptively-secure T-set implementation *if for all adversaries* A *there exists an algorithm* S *such that* $\Pr[\mathbf{Real}_A^{\Sigma}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{A,S}^{\Sigma}(\lambda) = 1] \leq \mathrm{neg}(\lambda)$.

A *non-adaptive* version of this definition is a straightforward modification of the above game where the adversary provides all of the queries at the start of the game.

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 [14, 13]. The details of the scheme, called SKS, are given in Figure 1. It uses as subroutines a PRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, PRP $P : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, and a CPA secure symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ that has λ-bit keys. The $\mathsf{GetRind}$ function for SKS is defined via $\mathsf{GetRind}((K_S, K_P, K_T), \mathsf{ind}) := P(K_P, \mathsf{ind})$. When combined with the specific adaptively-secure T-set construction shown in Appendix C, the SKS protocol results in a single-keyword SSE scheme that achieves the best storage and search performance among existing adaptive SSE protocols. Whereas our EDB stores $c{*}N$ short ciphertexts for a small constant $c \leq 3$, where $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$, i.e. the total number of keyword occurences in DB, the adaptive SSE scheme of [14] needs $d * \mathsf{max}$ storage where d is the number of records and max is the maximum number of keywords in any record, while the adaptive SSE scheme for keyword search of [13] needs $m * \mathsf{max}'$ storage where m is the number of keywords and max' is the maximum number of records matching any keyword. Consequently our T-set implementation improves over adaptive single-keyword SSE of [14] by factor equal to the proportion between maximum number of keywords in any record to the average number of keywords per record, and it improves over adaptive single-keyword SSE of [13] by factor equal to the proportion between maximum number of records matching any keyword to the average number of records per keyword.[4] Finally, the storage complexity of [25] is similar to ours, but their search procedure cannot be parallelized on the server because they represent a T-set as a linked list, and their reliance on the ROM model for security seems harder to do away with. (On the other hand, the scheme of [25] enables updates without growth in the datastructure size and search time.)

3 SSE Schemes for Conjunctive Keyword Search

Existing SSE schemes for conjunctive queries ([20] 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 will scale with the size of the (estimated) *smallest* $\mathsf{DB}(w_i)$ set among all the conjunctive terms w_i.

[4] As for communication costs, the adaptive scheme of [14] has $O(d)$ communication as stated, but it can be easily reduced to $O(|DB(w)|)$, the size of the result set. The scheme of [13] has $O(\mathsf{max}')$ communication, i.e. $O(|DB(w)|)$ for the worst case w. The schemes of [14, 13] do not rely on ROM, but our T-set implementation can also avoid ROM if instead of a single value strap the client streams to the server consecutive values $F(\mathsf{strap}, 1), F(\mathsf{strap}, 2), \ldots$ until receiving server's signal to stop.

EDBSetup(DB)

- Select keys K_S, K_P, for PRF F and PRP 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 $\mathsf{W} = \cup_{i=1}^d \mathsf{W}_i$.
- For each $w \in \mathsf{W}$, build the tuple list $\mathbf{T}[w]$ as follows:
  - Initialize $\mathbf{t}$ 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: $\mathsf{rind} \leftarrow P(K_P, \mathsf{ind})$, $\mathsf{e} \xleftarrow{\$} \mathsf{Enc}(K_e, \mathsf{rind})$; append e to $\mathbf{t}$.
  - Set $\mathbf{T}[w] \leftarrow \mathbf{t}$.
- $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$.
- Output the key (K_S, K_P, K_T) and $\mathsf{EDB} = \mathsf{TSet}$.

Search protocol

- The client takes as input the key (K_S, K_P, 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 $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$, and sends $\mathbf{t}$ to the client.
- Client sets $K_e \leftarrow F(K_S, w)$; for each e in $\mathbf{t}$, it computes $\mathsf{rind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ and outputs rind.

$\mathsf{GetRind}(K, \mathsf{ind})$ is defined as $P(K_P, \mathsf{ind})$ for $K = (K_S, K_P, K_T)$.

Figure 1: SKS: Single-Keyword SSE Scheme

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 rind's for each w_i. Then, the server returns to client the rind 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 rind 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 presented in Figure 2. Assume (see Section 3.1.1 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

Text in red indicates changes from SKS, Figure 1

EDBSetup(DB)

- Select keys K_S and K_X for PRF F, and key K_P for PRP P; 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 $K_e \leftarrow F(K_S, w)$.
  - Compute $\mathsf{xtrap} \leftarrow F(K_X, w)$
  - For all ind in $\mathsf{DB}(w)$ in random order:
    - Compute $\mathsf{rind} \leftarrow P(K_P, \mathsf{ind})$, $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rind})$ and append e to $\mathbf{t}$.
    - $\mathsf{xtag} \leftarrow f(\mathsf{xtrap}, \mathsf{rind})$ 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_P, K_T) and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.

Search protocol

- The client takes as input the key (K_S, K_X, K_P, K_T) and keywords $w_1, \ldots, w_n$ to query. It computes its messages as
  - $K_e \leftarrow F(K_S, w_1)$, $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w_1)$, then
  - For each $i = 2, \ldots, n$, it sets $\mathsf{xtrap}_i \leftarrow F(K_X, w_i)$.
  - It sends $(\mathsf{stag}, K_e, \mathsf{xtrap}_2, \ldots, \mathsf{xtrap}_n)$ to the server.
- The server has input $(\mathsf{TSet}, \mathsf{XSet})$. It responds as follows.
  - It sets $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$.
  - For each ciphertext e in $\mathbf{t}$, it computes
    - $\mathsf{rind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$
    - If $f(\mathsf{xtrap}_i, \mathsf{rind}) \in \mathsf{XSet}$ for all $i = 2, \ldots, n$, it sends rind to the client.
- The client outputs all of the received rinds.

GetRind(K, ind) is defined as $P(K_P, \mathsf{ind})$ for $K = (K_S, K_X, K_P, K_T)$.

Figure 2: BXT: Basic Cross-Tags Protocol

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 EDBSetup(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 $\mathsf{xtag} = f(\mathsf{xtrap}, \mathsf{rind})$ 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 rind as input). As in SKS, we have $\mathsf{rind} = P(K_P, \mathsf{ind})$. 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 $\mathsf{t} = \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$. Then, for each ciphertext e in t, it decrypts $\mathsf{rind} = \mathsf{Dec}(K_e, \mathsf{e})$ and if $f(\mathsf{xtrap}_i, \mathsf{rind}) \in \mathsf{XSet}$ for all $i = 2, \ldots, n$, it sends rind to the client.[5]

Correctness of the BXT protocol is easy to verify. Just note that a document indexed by rind includes a word w represented by stag if and only if $\mathsf{xtag} = f(\mathsf{xtrap}, \mathsf{rind}) \in \mathsf{XSet}$. 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 stag, xtag, stag, 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 |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 3.1.1. 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 rind'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 rind 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).

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 rind's to be unpredictable, a condition not needed in the other protocols.

3.1.1 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 structured attribute-based 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 is less immediate and we need to provide $\mathcal{C}$ with information to guide this choice. Fortunately, it is the common case that the number of very frequent words in a document collection is relatively small. This means that the client can keep a small state with sufficient information to choose light s-terms. For example, in the Enron data set [15] used in our testing, only 4% of 1,176,222 distinct words from 1,551,675 documents appear in more than 100 documents. With a state of

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

less than 100 Kbyte a client can keep information, e.g., via a Bloom filter, to differentiate between keywords with less than 100, 1000, 10,000, 100,000, matching documents, respectively (and if one omits the '> 100' category then the storage requirement reduces to less than 25KB). We note that in our experiments, retrieval of 1,000-hit s-terms takes less than a tenth of a second, so the level of granularity for the above tables can be tuned depending on the data and setting. A client that cannot afford storing a state of the above size, can retrieve it from the server (where this state is stored encrypted) at the beginning of a search session. And there is always the option to cap the number of searched and/or retrieved documents (after which the client may choose to refine its search). Finally, we observe that in the case of multi-client "controlled disclosure" SSE setting [13] discussed in the introduction, the data owner, who provides tokens to clients, will usually have enough storage to make optimal (or close-to-optimal) choices of s-terms.

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 rind value is a record with keyword w_i by testing if $f(\mathsf{xtrap}_i, \mathsf{rind}) \in \mathsf{XSet}$. This allows an honest-but-curious server to learn statistics about how the results of every s-term match every x-term of every query. 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 rind values that it gets in $\mathbf{t}$ (from the TSet) to the client who will compute the function $f(\mathsf{xtrap}_i, \mathsf{rind})$ and send back the results.

This, however, has its own drawbacks: it adds a round of communication with consequent latency, allows the server to cheat by sending rind 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 "controlled disclosure" SSE setting [13] discussed in the introduction (since the client would learn from the rind's 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 $\mathsf{xind} = F_p(K_I, \mathsf{rind}) \in Z_p^*$ 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; the server would raise this to the power xind and finally the client would de-blind it by raising to the power $z^{-1} \bmod p$ 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* (during EDBSetup) the blinding part of the oblivious computation and store it in the EDB. That is, 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 $\mathsf{xtoken}[c, i] := g^{F_p(K_X,w_i)\cdot z_c}$, $i = 1, \ldots, n$,

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^*), K_P for PRP 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 $\mathsf{DB}(w)$ in random order, initialize a counter $c \leftarrow 0$, then:
    - Set $\mathsf{rind} \leftarrow P(K_P, \mathsf{ind})$, $\mathsf{xind} \leftarrow F_p(K_I, \mathsf{rind})$, $z \leftarrow F_p(K_Z, w \,\|\, c)$ and $y \leftarrow \mathsf{xind} \cdot z^{-1}$.
    - Compute $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rind})$, 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 $(K_S, K_X, K_P, K_I, K_Z, K_T)$ and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.

Search protocol

- The client takes as input the key $(K_S, K_X, K_P, K_I, K_Z, K_T)$ and keywords $w_1, \ldots, w_n$ to query. 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 $(\mathsf{TSet}, \mathsf{XSet})$. 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 $\mathsf{rind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ and outputs rind.

GetRind(K, ind) is defined as $P(K_P, \mathsf{ind})$ for $K = (K_S, K_X, K_P, K_I, K_Z, K_T)$.

Figure 3: OXT: Oblivious Cross-Tags Protocol

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 3 (red text indicates changes from the BXT protocol). 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]

We 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 5.

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 rind's matching the query). The computational cost of OXT lies in the use of exponentiations, however thanks to the use of the fastest 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 our own performance numbers in Section F.

OXT leaks much less information to the server than BXT. Indeed, since the server, call it $\mathcal{S}$, learns neither the rind 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 record rind and an x-term xtrap then the xtag value $f(\mathsf{xtrap}, \mathsf{rind})$ will be the same in both queries indicating that rind 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 rind'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, $\mathcal{C}$ 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 5 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 [22] for an example of potential masking techniques.

4 Processing Boolean Queries with OXT

We now describe extensions to OXT that can handle *arbitrary* Boolean query expressions, but are efficient for a sub-class of expressions. We first consider conjunctions with negated terms.

Conjunction with negated terms. In the context of keyword-based search we refer to a "negated term" as one that returns documents that *do not contain* the given keyword. Consider a conjunction of n terms in which there is *at least one non-negated term.* To search for such a conjunction we

[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.

modify OXT as follows.

- The Client chooses one of the non-negated terms as the s-term, and computes stag and xtraps (i.e. the xtoken arrays) as in OXT. It then sends the stag and the xtraps to the Server, but indicating which xtraps are for negated terms.
- The Server's computation is exactly as in OXT, except that for negated terms it checks if the corresponding $(\mathsf{xtoken}[c,i])^y$ is *not* in the XSet (i.e. instead of checking if it is in the XSet).

A conjunction where *all* terms are negated cannot be executed as above since the Client cannot choose a negated term as s-term. We will see how we can accommodate, though inefficiently, such searches below. (Note that in most cases a negated term will have a very large number of matching documents; if this is the case for all the terms in the conjunction then efficient search, even on plaintext data, is not possible.)

Boolean expressions in Searchable Normal Form (SNF). 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., "$w_1 \wedge (w_2 \vee w_3 \vee \neg w_4)$"). Protocol OXT can be extended to answer such queries; the needed modifications are similar to those described above for the case of conjunctions with at least one non-negated term (a special case of SNF). Specifically, 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 rind 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.

Responding to arbitrary Boolean queries. OXT can be also be extended to 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.

Disjunctions. A disjunction of expressions that have efficient search under OXT or the extended OXT as described above can also be searched efficiently by running OXT (or the extended OXT) in each disjunct separately, and the client requesting the union of the sets of decrypted rind's for each disjunct.

5 Security Analysis of OXT

In this section we describe the OXT leakage profile $\mathcal{L}$ and analyze its security. While our ultimate goal is to prove adaptive security for boolean queries, we start by analyzing *non-adaptive* security for the special case where all queries are conjunctions of two keywords. This special case already captures essentially all of the difficulties in the analysis and dispenses with several distracting notational issues. Then we show how to extend the leakage profile to boolean queries (in particular for conjunctions with any number of terms) in Section 5.3, where we also analyze adaptive security.

5.1 Hardness assumptions

Decision Diffie-Hellman. Let $G = G_\lambda$ be a prime order cyclic group of order $p = p(\lambda)$ generated by g. We say that the *decision Diffie-Hellman (DDH) assumption* holds in G if $\mathbf{Adv}^{\mathrm{ddh}}_{G,A}(\lambda)$ is negligible for all efficient adversaries A, where

$$\mathbf{Adv}^{\mathrm{ddh}}_{G,A}(\lambda) = \Pr[A(g, g^a, g^b, g^{ab}) = 1] - \Pr[A(g, g^a, g^b, g^c) = 1]$$

where the probability is over the randomness of A and uniformly chosen a, b, c from Z_p^*.

For vectors $\mathbf{a} \in (Z_p^*)^\alpha, \mathbf{b} \in (Z_p^*)^\beta$ let $g^{\mathbf{a}} = (g^{\mathbf{a}[1]}, \ldots, g^{\mathbf{a}[\alpha]}) \in G^\alpha$ and $g^{\mathbf{a}\mathbf{b}^{\mathrm{T}}}$ be the matrix in $G^{\alpha\times\beta}$ where the (i,j)-th entry is $g^{\mathbf{a}[i]\cdot\mathbf{b}[j]}$. We will use the following standard lemma in our security proof.

Lemma 3 *Suppose the DDH assumption holds for in G. Then, for any integers α, β (polynomial in λ) any efficient adversary A, we have*

$$\Pr[A(g, g^{\mathbf{a}}, g^{\mathbf{b}}, g^{\mathbf{a}\mathbf{b}^{\mathrm{T}}}) = 1] - \Pr[A(g, g^{\mathbf{a}}, g^{\mathbf{b}}, \mathbf{M}) = 1] \leq \mathrm{neg}(\lambda),$$

where $\mathbf{a}$ is uniform over $(Z_p^*)^\alpha$, $\mathbf{b}$ is uniform over $(Z_p^*)^\beta$, and $\mathbf{M}$ is uniform over $G^{\alpha\times\beta}$.

PRF and PRP Security. Let X and Y be sets, and let $F : \{0,1\}^\lambda \times X \to Y$ be a function. We say that F is a *pseudorandom function (PRF)* if for all efficient adversaries A, $\mathbf{Adv}^{\mathrm{prf}}_{F,A}(\lambda)$ is negligible, where

$$\mathbf{Adv}^{\mathrm{prf}}_{F,A}(\lambda) = \Pr[A^{F(K,\cdot)}(1^\lambda) = 1] - \Pr[A^{f(\cdot)}(1^k) = 1]$$

where the probability is over the randomness of A, $K \xleftarrow{\$} \{0,1\}^\lambda$, and $f \xleftarrow{\$} \mathsf{Fun}(X, Y)$.

A *pseudorandom permutation* is a function $F : \{0,1\}^\lambda \times X \to X$ such that each $F(K,\cdot)$ induces a permutation on X and satisfies the same notion security, except that f is instead chosen at random from $\mathsf{Perm}(X)$. We will assume that F is efficiently invertible, meaning that given $K, y \in \{0,1\}^\lambda$, one can efficiently compute x such that $y = F(K, x)$. There we denote the adversary's advantage in attacking F by $\mathbf{Adv}^{\mathrm{prp}}_{F,A}(\lambda)$.

IND-CPA Encryption Security. A symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ is a pair of algorithms, the first randomized and the second deterministic. Enc takes as input a key $K \in \{0,1\}^\lambda$ and a message $M \in \{0,1\}^*$, and outputs a ciphertext C. Dec takes as input a key $K \in \{0,1\}^\lambda$ and a ciphertext C and outputs a message M. We require the usual correctness for all possible keys K and messages M.

We say that $\Sigma = (\mathsf{Enc}, \mathsf{Dec})$ is *IND-CPA secure* if for all efficient adversaries A $\mathbf{Adv}^{\mathrm{ind-cpa}}_{\Sigma,A}(\lambda)$ is negligible, where

$$\mathbf{Adv}^{\mathrm{ind-cpa}}_{\Sigma,A}(\lambda) = \Pr[A^{\mathcal{O}(K,0,\cdot,\cdot)}(1^\lambda) = 1] - \Pr[A^{\mathcal{O}(K,1,\cdot,\cdot)}(1^k) = 1],$$

where K is chosen at random from $\{0,1\}^\lambda$ and the oracle $\mathcal{O}(K, b, M_0, M_1)$ returns $\perp$ if $|M_0| \neq |M_1|$, and otherwise it samples $\mathsf{Enc}(K, M_b)$ and returns the result.

5.2 Warm up: Analysis for non-adaptive 2-conjunctions

We start by describing the function $\mathcal{L}_{\mathrm{oxt}}$ that describes the leakage of the OXT (beyond the T-set leakage) protocol under a simpler scenario where all queries are non-adaptive and are for conjunctions of two keywords only. This setting already confronts most of the intuitive difficulties in our

leakage profile and proof, but is less cluttered than the full setting, which we present in the next subsection.

Below our security theorem will show that this, in addition to the leakage from $\mathcal{L}_T$ which is defined by the T-set implementation, is *all* of the information leaked by our protocol.

We represent a sequence of Q non-adaptive 2-conjunction queries by $\mathbf{q} = (\mathbf{s}, \mathbf{x})$ where an individual query is a 2-term conjunction $\mathbf{s}[i] \wedge \mathbf{x}[i]$ which we write as $\mathbf{q}[i] = (\mathbf{s}[i], \mathbf{x}[i])$. $\mathcal{L}_{\text{oxt}}(\mathsf{DB}, \mathbf{q})$ gets $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$ and $\mathbf{q} = (\mathbf{s}, \mathbf{x})$ as input. It is a randomized algorithm that selects $\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$ and outputs $(N, \overline{\mathbf{s}}, \mathsf{SP}, \mathsf{RP}, \mathsf{IP})$, which are defined below. In our definition we use the notation $I_\pi(w) = \{\pi(\mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(w)\}$.

- $N = \sum_{i=1}^d |\mathsf{W}_i|$ is the total number of appearances of keywords in documents.
- $\overline{\mathbf{s}} \in [m]^Q$ is the *equality pattern* of $\mathbf{s} \in \mathsf{W}^Q$ indicating which queries have the equal s-terms. Formally, $\overline{\mathbf{s}} \in [m]^Q$ 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 $\overline{\mathbf{s}} = (1, 1, 2, 3, 1, 3)$. To compute $\overline{\mathbf{s}}[i]$ one finds the least j such that $\mathbf{s}[j] = \mathbf{s}[i]$ and then lets $\overline{\mathbf{s}}[i] = |\{\mathbf{s}[1], \ldots, \mathbf{s}[j]\}|$ be the number of unique keywords appearing at indices less than or equal to j.
- SP is the *size pattern* of the queries, which is 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*. For indices i, k between 1 and Q, we say that k is *relevant for* i if $\mathbf{s}[k] \neq \mathbf{s}[i]$ and $\mathbf{x}[k] = \mathbf{x}[i]$. IP consists of $(S_1, \ldots, S_Q)$ where for each $i = 1, \ldots, Q$ the set S_i defined by $S_i = I_\pi(\mathbf{s}[i]) \cap \bigcup_{k:k \text{ relevant for } i} I_\pi(\mathbf{s}[k])$.

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 $\overline{\mathbf{s}}$ 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 chosing 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 *if there is a document satisfying both s-terms* then the set of random indexes corresponding to the documents matching both s-terms is leaked (if no document matching both s-terms exist then nothing is leaked). It can be seen as the price we pay for the rich functionality enabled by our x-terms and XSet approach that allows for the computation of arbitrary boolean queries. Note, however, that since the s-terms are meant to be the least-frequently matched keywords, the number of instances with s-terms having a non-empty intersection of documents containing both terms is minimized. Moreover, in searches where the s-term is a unique per-document term (e.g., a last name on a database with a last-name field) the IP leakage cannot happen.

It also helps to compare the above leakage to the leakage incurred by the naïve solution that simply sends tokens for each keyword in the conjunction. In that case, the server would learn $I_\pi(\mathbf{s}[i])$ and $I_\pi(\mathbf{x}[i])$ for every i, as it would see the results of the individual searches (instead of just the results pattern, which would be ideal). This means it would learn the relationships between all

of the documents matching the individual keywords, and this is information is exactly what OXT is designed to minimize.

We also remark that d, the number of documents in DB is not leaked in the above profile but would like be leaked in an implementation that stores the encrypted documents. This is outside the formal model analyzed here but could be easily incorporated at the cost of extra notation.

OXT security theorem for two-term conjunctions. Theorem 5 below that states the security of OXT for two-term conjunctions requires the following lemma that establishes a bound on the correctness of the protocol, which for OXT means the probability that a document that does not match a query will be returned to the client. The next lemma shows that this happens only with negligible probability. See Section 5.1 for standard assumptions (PRF, PRP, DDH) used in the theorem.

Lemma 4 *For every adversary A there exists adversaries B and B' which run in essentially the same time as A, such that*

$$\Pr[\mathbf{Cor}_A^{\mathsf{OXT}}(\lambda) = 1] \leq 2 \cdot \mathbf{Adv}^{\mathrm{prf}}_{F_p,B}(\lambda) + \mathbf{AdvCor}^{\Pi}_{B'}(\lambda) + N^2/(p-1) + N/p,$$

where $N = \Sigma_{i=1}^{d} |\mathsf{W}_i|$ is the total number of appearances of keywords in all documents, p is the order of the group G, and Π is the T-set implementation.

The proof appears in Appendix A. Intuitively, the only way a failure in correctness can occur is if either the T-set fails or if two keyword-identifier pairs from DB map to the same xtag using $F_p(K_I, \cdot)$ and $F_p(K_X, \cdot)$. But since these values are pseudorandom over a large set, the chance of this happening is negligible.

The statement of our generic security theorem for a generic T-set is somewhat cumbersome due to an issue with how leakages compose, and is given in Section B. For the case of OXT that uses our T-set implementation from Section C, which is of primary interest, we have that $\mathcal{L}_{\mathrm{oxt}}$ constitutes all of the leakage.

Theorem 5 *Let $\mathcal{L}_{\mathrm{oxt}}$ be as defined above, and suppose that the T-set implementation Σ from Section C. Then SSE scheme* OXT *is $\mathcal{L}_{\mathrm{oxt}}$-semantically-secure against non-adaptive attacks where are all queries are 2-conjunctions, assuming that the DDH assumption holds in G, that F and F_p are secure PRFs, P is a secure PRP, that* (Enc, Dec) *is an IND-CPA secure symmetric encryption scheme, and the conditions from Theorem 10 hold.*

Proof sketch. The proof of the theorem is delicate and lengthy, and is given in Appendix B for the more general formulation in Theorem 8. 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.

The size of the XSet is equal to the value N leaked. The equality pattern for $\mathbf{s}$, $\overline{\mathbf{s}}$, (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 $g^{F_p(K_X,w_i)\cdot F_p(K_I,\mathsf{rind})}$, where w_i is the i-th keyword from a search and rind is a permuted identifier for a document that matched the s-term (by may or may not match the remaining keywords). The leakage comes from the fact that the values will sometimes *repeat* (when two queries repeat the same w_i and match the same rind with their s-terms) and they are all known to the adversary. The IP describes exactly what can be learned from this attack.

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.

5.3 Analysis for boolean queries

In this section we give the leakage profile for OXT under adaptive attacks where any boolean query in searchable normal form (SNF), i.e., of the form $\psi(s, x_1, \ldots, x_n) = s \wedge \phi(x_1, \ldots, x_n)$, is allowed ($n$ can vary between queries). In particular, this leakage profile extends the one in the previous section to conjunctions with any number of terms. As before, for simplicity we describe the leakage profile when OXT is used with our specific T-set implementation from Section C.

The new leakage profile $\mathcal{L}$ is a randomized stateful algorithm that responds to inputs as defined in Definition 1. On the initial input DB, it outputs $N = \sum_{i=1}^{d} |\mathsf{W}_i|$. It also selects a random permutation $\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$ and saves π as state. Later inputs consist of a vector of queries $\mathbf{q} = (\Phi, \mathbf{s}, \mathbf{x}_1, \ldots, \mathbf{x}_n)$, where Φ is a vector of boolean formulae and $\mathbf{s}, \mathbf{x}_1, \ldots, \mathbf{x}_n$ are vectors of keywords, all populated according to the queries issued so far in the straightforward way[8]. The leakage function $\mathcal{L}$ outputs $(N, \Phi, \overline{\mathbf{s}}, \mathsf{SP}, \mathsf{XP}, \mathsf{RP}, \mathsf{IP}, \mathsf{XMP})$, where $N, \overline{\mathbf{s}}, \mathsf{SP}$ are computed as in Section 5.2 and Φ is from $\mathbf{q}$. It computes the remaining elements as follows:

- The vector XP is initialized so that $\mathsf{XP}[i]$ is the number of x-terms in the i-th query.

[8]Formally, n is the maximum number of x-terms in any query, with smaller queries padded up with a special dummy symbol. Hence, if the i-th query uses boolean formula ϕ on $n' + 1$ inputs then we have $\Phi[i] = \phi$ and the input is represented by $(\mathbf{s}[i], \mathbf{x}_1[i], \ldots, \mathbf{x}_n[i])$ where the last $n - n'$ terms are set to the dummy symbol.

- For *results pattern* RP, for each $i = 1, \ldots, Q$ it computes $R_i = \{\pi(\mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(\Phi[i])\}$. It sets $\mathsf{RP} = (R_1, \ldots, R_Q)$.

- The *conditional intersection pattern* IP of the queries is computed as follows. Let i, k be indices between 1 and Q. We say that an index k is *relevant for* i if (1) $\mathbf{s}[k] \neq \mathbf{s}[i]$ and (2) there exist $j, \ell \in [n]$ such that $\mathbf{x}_j[k] = \mathbf{x}_\ell[i]$.

  IP consists of $(S_1, \ldots, S_Q)$ where for each $i = 1, \ldots, Q$ the set S_i is defined by

$$S_i = I_\pi(\mathbf{s}[i]) \cap \left(\bigcup_{k:k \text{ relevant for } i} I_\pi(\mathbf{s}[k]) \right).$$

  In words, this is the set of permuted identifiers that match the keyword $\mathbf{s}[i]$ *and* some other keyword $\mathbf{s}[j]$ that was queried with at least one x-term in common.

- The *x-term matching pattern* XMP represents leakage from the fact that for each query the server which x-terms match the preliminary results from the initial search from the T-set. Making this precise is delicate. We start by defining the *x-term matching pattern of an index* ind, denoted $\mathsf{xmp}_i(\mathsf{ind})$, as

$$\mathsf{xmp}_i(\mathsf{ind}) = \{j \in [n] : \mathsf{ind} \in \mathsf{DB}(x_j[i])\}.$$

  In words, $\mathsf{xmp}_i(\mathsf{ind})$ is the indices of x-terms in query i that are matched by ind.

  The $\mathsf{XMP} = (X_1, \ldots, X_Q)$ is an array of length Q that contains *multi-sets*. Each X_i is defined as $X_i := \{\mathsf{xmp}_i(\mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(\mathbf{s}[i])\}$. Note that since X_i is a multi-set, it is represented in some canonical order with repetitions allowed.

Theorem 6 *Let $\mathcal{L}$ be the leakage function defined above. Then OXT, when implemented using the T-set implemenation from Section C, is $\mathcal{L}$-semantically-secure against adaptive attacks, assuming that the DDH assumption holds in G, that F and F_p are secure PRFs, P is a secure PRP, that $(\mathsf{Enc}, \mathsf{Dec})$ is an IND-CPA secure symmetric encryption scheme, and that the conditions in Theorem 10 are satisfied.*

The proof of this theorem is an extension of the non-adaptive theorem in two ways. First, the proof must handle the simulation of more complicated general queries, and second it must do this adaptively. Handling general queries introduces few complications because the leakage of Φ and XMP gives the simulator enough information to properly program the results and the XSet in a way that generalizes what is done in the proof for 2-conjunctions. Thus we focus on how to handle adaptivity.

The adaptive simulator works as follows. To generate $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$, the simulator invokes the adaptive TSet simulator from the proof of Theorem 10 on input N and it generates XSet by choosing N random group elements and adding them to XSet.

To simulate queries responses, the simulator will adaptively "assign" elements of the XSet to keyword-rind pairs. This is in contrast with the non-adaptive simulator, which achieved this by initializing the array H and then adding the elements to the XSet as determined by the leakage. Here, the simulator is choosing the XSet values, and then initializing H entries adaptively, using either elements of XSet (with the proper repetitions) or independent random elements (again, with the proper repetitions).

In more detail, the simulator maintains H as before, but does not initialize any of its entries, and to start it marks all of the simulated XSet elements as "unused". To respond to the i-th query

$(\phi, \mathbf{s}[i], \mathbf{x}_1[i], \ldots, \mathbf{x}_n[i])$, it first invokes its adaptive T-set simulator to generate the stag value, where it gives the simulator a set of rinds that consists of $R_i \cup S_i$ padded with extra (e, y) pairs to appropriate number indicated by $\mathsf{SP}[i]$. To complete the response to a query, the simulator produces a xtoken using a similar "backwards" computation as in the non-adaptive case. For each entry in the xtoken, it must select and possibly initialize an entry in H. It logically associates each rind in $R_i \cup S_i$ with a random entry in the xtoken, and for each entry $H[\mathsf{rind}, \overline{\mathbf{x}}[i]]$ that is initialized, it uses that value in the backwards computation. For the uninitialized entries, if the rind was in R_i, then it chooses a random unused element of XSet, marks it as "used," and inserts that into the corresponding entry of H. If the rind was in $S_i \setminus R_i$, it chooses a random element to fill in H.

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] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 205–222. Springer, Aug. 2005.

[2] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, New York, 1992.

[3] L. Ballard, S. Kamara, and F. Monrose. Achieving efficient conjunctive keyword searches over encrypted data. In S. Qing, W. Mao, J. López, and G. Wang, editors, *ICICS 05*, volume 3783 of *LNCS*, pages 414–426. Springer, Dec. 2005.

[4] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 535–552. Springer, Aug. 2007.

[5] M. Bellare, P. Rogaway, and T. Spies. The FFX mode of operation for format-preserving encryption. Draft 1.1, NIST Block Cipher Modes Working-Group, 2010.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[7] D. Boneh and X. Boyen. Efficient selective-ID secure identity based encryption without random oracles. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 223–238. Springer, May 2004.

[8] D. Boneh and X. Boyen. Short signatures without random oracles. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 56–73. Springer, May 2004.

[9] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 506–522. Springer, May 2004.

[10] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 535–554. Springer, Feb. 2007.

[11] J. W. Byun, D. H. Lee, and J. Lim. Efficient conjunctive keyword search on encrypted data storage system. In *EuroPKI*, pages 184–196, 2006.

[12] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 442–455. Springer, June 2005.

[13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT 2010*, LNCS, pages 577–594. Springer, Dec. 2010.

[14] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. Vimercati, editors, *ACM CCS 06*, pages 79–88. ACM Press, Oct. / Nov. 2006.

[15] EDRM (edrm.net). Enron data set. `http://www.edrm.net/resources/data-sets/edrm-enron-email$\penalty\z@$-`

[16] S. D. Galbraith, K. G. Paterson, and N. P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[17] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[18] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/`.

[19] S. Goldwasser and R. Ostrovsky. Invariant signatures and non-interactive zero-knowledge proofs are equivalent (extended abstract). In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 228–245. Springer, Aug. 1993.

[20] P. Golle, J. Staddon, and B. R. Waters. Secure conjunctive keyword search over encrypted data. In M. Jakobsson, M. Yung, and J. Zhou, editors, *ACNS 04*, volume 3089 of *LNCS*, pages 31–45. Springer, June 2004.

[21] IARPA. Security and Privacy Assurance Research (SPAR) Program - BAA, 2011. `http://www.iarpa.gov/solicitations_spar.html/`.

[22] M. Islam, M. Kuzu, and M. Kantarcioglu. 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, Feb. 2012. Internet Society.

[23] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Financial Cryptography Workshops*, pages 136–149, 2010.

[24] S. Kamara, C. Papamanthou, and T. Roeder. "CS2: A searchable cryptographic cloud storage system", 2011. `http:research.microsoft.com/pubs/148632/CS2.pdf`.

[25] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proc. of CCS'2012*, 2012.

[26] K. Kurosawa and Y. Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography*, page 285, 2012.

[27] Lemur Project. ClueWeb09 dataset. `http://lemurproject.org/clueweb09.php/`.

[28] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, Oct. 2011.

[29] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *2007 IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society Press, May 2007.

[30] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society Press, May 2000.

[31] P. van Liesdonk, S. Sedhi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Proc. Workshop on Secure Data Management (SDM)*, pages 87–100, 2010.

[32] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS 2004*. The Internet Society, Feb. 2004.

A Proof of Lemma 4

Let A be an adversary and Π be the OXT protocol, and let $\mathrm{G}_0 = \mathbf{Cor}_A^{\Pi}(\lambda)$. We want to show

$$\Pr[\mathrm{G}_0 = 1] = \mathrm{neg}(\lambda).$$

We first modify G_0 to output 1 if the T-set ever errs in returning the correct data $\mathbf{t}$ when the server is processing a search in the game. It is straightforward to construct an adversary B' such that

$$\Pr[\mathrm{G}_1 = 1] - \Pr[\mathrm{G}_0 = 1] = \mathbf{AdvCor}_{B'}^{\Pi}(\lambda).$$

Next we consider a modification of G_0 that starts by choosing random functions f_X, f_I mapping $\{0,1\}^*$ to Z_p^*, and then replaces all evaluations of $F_p(K_X, \cdot)$ and $F_p(K_I, \cdot)$ with evaluations of $f_X(\cdot)$ and $f_I(\cdot)$ respectively.

Call this modification G_2. By a straightforward reduction the PRF security of F_p and a hybrid argument, we can build an efficient adversary B such that

$$\Pr[\mathrm{G}_2 = 1] - \Pr[\mathrm{G}_1 = 1] \leq 2 \cdot \mathbf{Adv}_{F_p,B}^{\mathrm{prf}}(\lambda).$$

We complete the proof by showing that

$$\Pr[\mathrm{G}_2 = 1] \leq N^2/(p-1) + (d+m)/p$$

and combining and rearranging the inequalities.

In G_2, let $\bar{w} = (w_1, w_2)$ be the query chosen by A after seeing EDB. The game will output 1 only if the simulated search protocol causes the client to have output not equal to the permuted results in $\mathsf{DB}(\bar{w})$. Let us examine the search protocol computation in more detail. By the correctness of the T-set implementation, the vector $\mathbf{t}$ computed by the server consists of ciphertexts decrypting exactly to the set $\{P(K_P, \mathsf{ind}) : \mathsf{ind} \in \mathsf{DB}(w_1)\}$. Of these, the server returns those that satisfy $g^{f_I(P(K_P,\mathsf{ind}))\cdot f_X(w_2)} \in \mathsf{XSet}$. If $\mathsf{ind} \in \mathsf{DB}(w_2)$ then this will clearly be the case, so we only need to rule out the possibility that a false positive occurs and the test with the XSet is true but $\mathsf{ind} \notin \mathsf{DB}(w_2)$ for this to happen, there would have to be some $(\mathsf{ind}', w') \neq (\mathsf{ind}, w_2)$ such that

$$g^{f_I(P(K_P,\mathsf{ind}))\cdot f_X(w_2)} = g^{f_I(P(K_P,\mathsf{ind}'))\cdot f_X(w')}.$$

Assume that for every ind, $f_I(P(K_P, \mathsf{ind})) \neq 0$ and for every keyword w, $f_X(w) \neq 0$. (This is true with all but $(d+m)/p$ probability.) Then the above question holds with probability $1/(p-1)$ over the choice of f_I and f_X. The desired bound follows from taking a union bound over all N^2 possibilities for (ind, w_2) and (ind', w'), and by adding the probability $\mathbf{AdvCor}_A^{\Pi}$ that in the SSE correctness game with adversary A there occurs a correctness break in the underlying T-set implementation Π.

B Proof of Theorem 5

Here we prove Theorem 5. More precisely, we prove a more general version of it that considers a generic instantiation of TSet. See Theorem 8 below.

We recall the non-adaptive security definition used in the theorem.

Definition 7 *Let* $\Pi = (\mathsf{EDBSetup}, \mathsf{Search}, \mathsf{GetRind})$ *be an SSE scheme and let* $\mathcal{L}$ *be an algorithm. For efficient 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 *and a list of queries* $\mathbf{q}$*. The experiment then runs* $(K, \mathsf{EDB}) \leftarrow \mathsf{EDBSetup}(\mathsf{DB})$*. For each* $i \in |\mathbf{q}|$*, it runs the* Search *protocol with client input* $(K, \mathbf{q}[i])$ *and server input* EDB *and stores the transcript and the client's output in* $\mathbf{t}[i]$ *Finally the game gives* EDB *and* $\mathbf{t}$ *to* A*, which returns a bit that the game uses as its own output.*

$\mathbf{Ideal}^{\Pi}_{A,S}(\lambda)$ *:* $A(1^{\lambda})$ *chooses* DB *and a list of queries* $\mathbf{q}$*. The experiment then runs* $S(\mathcal{L}(\mathsf{DB}, \mathbf{q}))$ *and gives its output to* A*, which returns a bit that the game uses as its own output.*

We say that Π *is* $\mathcal{L}$*-semantically-secure against non-adaptive attacks if for all efficient 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 describe the leakage function $\mathcal{L}$ that uses $\mathcal{L}_{\mathrm{oxt}}$ along with the T-set leakage function $\mathcal{L}_t$. On input DB and $(\mathbf{s}, \mathbf{x})$, it computes the vector $\mathbf{T}$ via

For $w \in \mathsf{W}$ do
  $K \xleftarrow{\$} \{0,1\}^{\lambda}$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do $y \xleftarrow{\$} Z_p^*$; $\mathsf{e} \leftarrow \mathsf{Enc}(K, 0^{\lambda})$; $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End.

Then it outputs $(\mathcal{L}_{\mathrm{oxt}}(\mathsf{DB}, (\mathbf{s}, \mathbf{x})), \mathcal{L}_T(\mathbf{T}, \mathbf{s}))$.

Theorem 8 *Let* $\mathcal{L}$ *be the leakage function defined above. Then SSE scheme* OXT *is* $\mathcal{L}$*-semantically-secure against non-adaptive attacks where are all queries are 2-conjunctions, assuming that the DDH assumption holds in* G*, that* F *and* F_p *are secure PRFs,* P *is a secure PRP, that* $(\mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA secure symmetric encryption scheme, and that* Σ *is a (non-adaptively)* $\mathcal{L}_T$*-secure and computationally correct T-set implementation.*

Proof. We structure our proof using several games $G_0, G_1, \ldots$. In each game, A starts by supplying $\mathsf{DB}, \mathbf{q}$, which is then given to an $\mathsf{Initialize}$ routine that produces an output, which is given to A who then outputs a bit that becomes the game output. Game G_0 is designed to generate exactly the same distribution as $\mathbf{Real}^{\Pi}_{A}(\lambda)$ (assuming no false positives occur) and the final game is structured so that it is easy to simulate exactly given the leakage profile instead of the actual $\mathsf{DB}, \mathbf{q}$ input. By relating the games, we can argue that the final simulator satisfies Definition 7 with OXT, completing the proof. For simplicity only, these games all model a version of the protocol where the client sends the maximum-sized xtrap (say, T group elements where T is some publicly known upper bound) on each query, and the proof is easily generalizable to the version where the server interactively tells the client to stop.

Game G_0. The first game G_0, which uses the routines in Figure 4, is an implementation of the real game with some minor changes that will make the analysis easier later. The game starts by running $\mathsf{Initialize}$, which passes $(\mathsf{DB}, \mathbf{s}, \mathbf{x})$ from A and simulates $\mathsf{EDBSetup}(\mathsf{DB})$, with the a few changes. It computes RDB, which is exactly like DB except that each ind_i is replaced with $P(K_P, \mathsf{ind}_i)$, for

use everywhere later in the game. Then, in building the data $\mathbf{T}$ it records the permutations σ in a vector WPerms indexed by keywords. These permutations will be used below when generating transcripts.

Before generating the transcripts, the game computes an array STags of all of the stag values used in the game. Specifically, for each $i = 1, \ldots, Q$ it lets $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, \mathbf{s}[i])$. To compute the transcript array $\mathbf{t}$, for $i = 1, \ldots, Q$ it sets $\mathbf{t}[i]$ to the output of $\mathsf{GenTrans}(\mathsf{EDB}, K_X, K_Z, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$, which is defined in the figure. There we use subroutine $\mathsf{ServerSearch}$ which we take to be the server's computation defined in OXT in response to the first client message.

The routine $\mathsf{GenTrans}$ generates a transcript as in the real game, except that it computes the $\mathsf{ResRinds}$ array differently: Instead of decrypting the ciphertexts returned with Res, it looks up the rind values that correspond to the results (specifically, it computes $\mathsf{RDB}(\mathbf{s}[i])$ and then finds the rind values amongst them that are also in $\mathsf{RDB}(x)$). It is here that the permutations stored in WPerms are used to ensure that the rind values are returned in the same order that they would in the real game. (The reason doing this rather than decrypting the ciphertexts is to enable a reduction to the IND-CPA security of the encryption scheme later.)

By design, G_0 is exactly $\mathbf{Real}^{\Pi}_A(\lambda)$, except that false positives are assumed to never happen. By Theorem 4, assuming that F_p is a secure PRF, we have

$$\Pr[G_0 = 1] \leq \Pr[\mathbf{Real}^{\Pi}_A(\lambda) = 1] + \mathrm{neg}(\lambda).$$

Game G_1. In the next game, G_1, is exactly the same except we replace every evaluation of $F(K_S, \cdot), F_p(K_X, \cdot), F_p(K_I, \cdot), F_p(K_Z, \cdot), P(K_P, \cdot)$ with evaluations of independent random functions with the appropriate domain and range. It is described in Figure 5. Note that since $F(K_S, \cdot)$ is never evaluated on the same input twice, we can equivalently replace its evaluations with random selections from the range (this is done in $\mathsf{Initialize}$ when selecting K_e). By a standard hybrid argument it is easy to show that there exist efficient adversaries $B_{1,1}, B_{1,2}, B_{1,3}$ such that

$$\Pr[G_1 = 1] - \Pr[G_0 = 1] \leq \mathbf{Adv}^{\mathrm{prf}}_{F,B_{1,1}}(\lambda) + 3 \cdot \mathbf{Adv}^{\mathrm{prf}}_{F_p,B_{1,2}}(\lambda) + \mathbf{Adv}^{\mathrm{prp}}_{P,B_{1,3}}(\lambda)$$

We omit the tedious and standard details of the adversaries. The salient feature of the game is that these keys are chosen at random, and then afterwards only used as key inputs to the corresponding functions.

Game G_2. In G_2 we modify G_1 to include the boxed code. This means that, during $\mathsf{Initialize}$, the ciphertext e is always overwritten with an encryption of 0^λ (under the same key K). We claim that there exists an efficient adversary B_2 such that

$$\Pr[G_2 = 1] - \Pr[G_1 = 1] \leq m \cdot \mathbf{Adv}^{\mathrm{ind-cpa}}_{\Sigma,B_2}(\lambda)$$

This follows by a standard hybrid argument over the m encryption keys used in building $\mathbf{T}$ for $\mathsf{TSetSetup}$. We omit the tedious details. We stress that the reduction is possible because the game never invokes the decryption algorithm Dec of the scheme, meaning that the reduction does not need to decrypt ciphertexts during the IND-CPA game.

Game G_3. In game G_3 (Figure 6) we alter the way the game is computed without changing its distribution. Intuitively, it precomputes the values in the XSet as well as all of the group elements that will be tested against the XSet, along with the group elements used xtoken arrays in transcripts that do not correspond to possible matches.

In $\mathsf{Initialize}$, two arrays H, Y are filled in for use in $\mathsf{XSetSetup}$ and $\mathsf{GenTranscript}$. H is indexed by a permuted identifier (string in $\{0,1\}^\lambda$) and a keyword from W, and holds group elements from G, and Y is indexed by a keyword and a number between 1 and T and also holds elements from G.

$\underline{\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})}$ // G_0
$K_S, K_I, K_X, K_Z, K_P \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{DB}$
For $i = 1, \ldots, d$ do $\mathsf{rind}_i \leftarrow P(K_P, , \mathsf{ind}_i)$
$\mathsf{RDB} \leftarrow (\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d$
For $w \in \mathsf{W}$
  $(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_w}) \leftarrow \mathsf{RDB}(w)$
  $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  $K_e \leftarrow F(K_S, w)$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do
    $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rind}'_{\sigma(c)})$
    $\mathsf{xind}' \leftarrow F_p(K_I, \mathsf{rind}'_{\sigma(c)})$
    $z \leftarrow F_p(K_Z, w \,\|\, c)$
    $y \leftarrow \mathsf{xind}' \cdot z^{-1} \bmod p$
    $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  End
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End
$(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$
For $i = 1, \ldots Q$ do $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, \mathbf{s}[i])$
$\mathsf{XSet} \leftarrow \mathsf{XSetSetup}(K_P, K_X, K_I, \mathsf{DB})$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet})$
For $i = 1, \ldots, Q$ do
  $\mathbf{t}[i] \leftarrow \mathsf{GenTrans}(K_X, K_Z, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$
End
Return $(\mathsf{EDB}, \mathbf{t})$

$\underline{\mathsf{XSetSetup}(K_X, K_I, \mathsf{DB})}$ // G_0
$(\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{RDB}$; $\mathsf{XSet} \leftarrow \emptyset$
For each $w \in \mathsf{W}$ do
  $e \leftarrow F_p(K_X, w)$
  For each $\mathsf{rind} \in \mathsf{RDB}(w)$ do
    $\mathsf{xind} \leftarrow F_p(K_I, \mathsf{rind})$; $h \leftarrow g^{e \cdot \mathsf{xind}}$
    $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{h\}$
  End
End
Return XSet

$\underline{\mathsf{GenTrans}(\mathsf{EDB}, K_X, K_Z, s, x, \mathsf{stag})}$ // G_0
$e \leftarrow F_p(K_X, x)$
For $c = 1, \ldots, T$ do
  $z \leftarrow F_p(K_Z, s \,\|\, c)$; $\mathsf{xtoken}[c] \leftarrow g^{e \cdot z}$
End
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{stag}, \mathsf{xtoken}))$
$\mathsf{ResRinds} \leftarrow \mathsf{RDB}(s) \cap \mathsf{RDB}(x)$
Return $((\mathsf{stag}, \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$

Figure 4: Game G_0.

Now XSetSetup is modified to use the values from the H array. For a given rind_i and $w \in \mathsf{W}_i$, it adds the value $H[\mathsf{rind}_i, w]$ to XSet - but this value is set to $g^{f_I(\mathsf{rind}_i)) \cdot f_X(w)}$ during Initialize, which is the value that was computed in XSetSetup in game G_2.

The H and Y arrays are both used in GenTranscript. We claim that, for any input DB and fixed randomness, each invocation of GenTranscript will return exactly the same output in G_2 and G_3. Clearly the stag, Res, ResRinds values will be the same, so we focus on the xtoken array.

In G_2, for each c, $\mathsf{GenTranscript}(\mathsf{EDB}, f_X, f_Z, s, x, \mathsf{stag})$ sets $\mathsf{xtoken}[c]$ to $g^{f_X(x) \cdot f_Z(s \,\|\, c)}$. In G_3, $\mathsf{GenTranscript}(\mathsf{EDB}, f_X, f_Z, s, x, \mathsf{stag})$ looks up $\sigma = \mathsf{WPerms}[s]$, $\mathsf{RDB}(s) = (\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_s})$ and $\mathbf{t}$. By the correctness of the TSet implementation we have $\mathbf{t} = (f_I(\mathsf{rind}'_{\sigma(c)})/f_Z(s \,\|\, c), \mathsf{e}_c)_{c=1}^{T_s}$, for some ciphertexts $\mathsf{e}_1, \ldots, \mathsf{e}_{T_s}$, where the rind'_c and σ as the same values as above.

Then for $c = 1, \ldots, T_s$, y is set to $f_I(\mathsf{rind}'_{\sigma(c)})/f_Z(s \,\|\, c)$ and $\mathsf{xtoken}[c]$ is set

$$H[\mathsf{rind}'_{\sigma(c)}, x]^{1/y} = g^{f_X(x) \cdot f_Z(s \,\|\, c)},$$

which shows that the $\mathsf{xtoken}[c]$ value is the same in both games for $c \leq T_s$. Note that here we have crucially used the fact that the set $\mathsf{RDB}(s)$ was parsed in the same canonical order in both places in the game.

$\underline{\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})}$ // G_1, $\boxed{G_2}$
$f_I, f_X, f_Z \xleftarrow{\$} \mathsf{Fun}(\{0,1\}^\lambda, Z_p^*)$
$\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$
$(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{DB}$
For $i = 1, \ldots, d$ do $\mathsf{rind}_i \leftarrow \pi(\mathsf{ind}_i)$
$\mathsf{RDB} \leftarrow (\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d$
For $w \in \mathsf{W}$
  $(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_w}) \leftarrow \mathsf{RDB}(w)$
  $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  $K_e \xleftarrow{\$} \{0,1\}^\lambda$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do
    $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rind}'_{\sigma(c)})$
    $\boxed{\mathsf{e} \leftarrow \mathsf{Enc}(K_e, 0^\lambda)}$
    $\mathsf{xind}' \leftarrow f_I(\mathsf{rind}'_{\sigma(c)})$
    $z \leftarrow f_Z(w \,\|\, c)$
    $y \leftarrow \mathsf{xind}' \cdot z^{-1} \bmod p$
    $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  End
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End
$(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$
For $i = 1, \ldots Q$ do $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, \mathbf{s}[i])$
$\mathsf{XSet} \leftarrow \mathsf{XSetSetup}(f_X, f_I, \mathsf{RDB})$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet})$
For $i = 1, \ldots, Q$ do
  $\mathbf{t}[i] \leftarrow \mathsf{GenTrans}(f_X, f_Z, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$
End
Return $(\mathsf{EDB}, \mathbf{t})$

$\underline{\mathsf{XSetSetup}(f_X, f_I, \mathsf{DB})}$ // G_1, G_2
$(\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{RDB}$
$\mathsf{XSet} \leftarrow \emptyset$
For each $w \in \mathsf{W}$ do
  $e \leftarrow f_X(w)$
  For each $\mathsf{rind} \in \mathsf{RDB}(w)$ do
    $\mathsf{xind} \leftarrow f_I(\mathsf{rind})$; $h \leftarrow g^{e \cdot \mathsf{xind}}$
    $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{h\}$
  End
End
Return XSet

$\underline{\mathsf{GenTrans}(\mathsf{EDB}, f_X, f_Z, s, x, \mathsf{stag})}$ // G_1, G_2
$e \leftarrow f_X(x)$
For $c = 1, \ldots, T$ do
  $z \leftarrow f_Z(s \,\|\, c)$; $\mathsf{xtoken}[c] \leftarrow g^{e \cdot z}$
End
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{stag}, \mathsf{xtoken}))$
$(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_s}) \leftarrow \mathsf{RDB}(s)$
$\sigma \leftarrow \mathsf{WPerms}[s]$; $i \leftarrow 0$
$\mathsf{ResRinds} \leftarrow \mathsf{RDB}(s) \cap \mathsf{RDB}(x)$
Return $((\mathsf{stag}, \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$

Figure 5: Games G_1 and G_2. G_2 includes the boxed code and G_1 does not.

For $c > T_s$, G_3 sets $\mathsf{xtoken}[c]$ to

$$Y[s, x, c] = g^{f_X(x) \cdot f_Z(s \,\|\, c)^{-1}},$$

which is exactly the same as $\mathsf{xtoken}[c]$ in G_2. This completes the claim that same value is returned in either game, so we have

$$\Pr[G_3 = 1] = \Pr[G_2 = 1].$$

Game G_4. Game G_4 is exactly like G_3 except that the boxed code is included: Now the game simply sets every y value of the $\mathbf{t}$ arrays to be an independently chosen element of Z_p^*. We claim that

$$\Pr[G_4 = 1] = \Pr[G_3 = 1].$$

This can be seen by observing that the random function f_Z is chosen during $\mathsf{Initialize}$ and then never evaluated in any of the other subroutines of G_3, thanks to the modifications made in moving to G_3. Moreover, for any $w \in \mathsf{W}$ and $c = 1, \ldots, T$, the value of $z = f_Z(w \,\|\, c)$ is used exactly once during $\mathsf{Initialize}$, and this value is uniform and independent of the rest of the randomness in the game.

$\underline{\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})}$ // G_3, $\boxed{G_4}$, $\boxed{\boxed{G_5}}$
$f_I, f_X, f_Z \xleftarrow{\$} \mathsf{Fun}(\{0,1\}^\lambda, Z_p^*)$
$\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$
$(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{DB}$
For $i = 1, \ldots, d$ do $\mathsf{rind}_i \leftarrow \pi(\mathsf{ind}_i)$
$\mathsf{RDB} \leftarrow (\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d$
For $w \in \mathsf{W}$ do
  $e \leftarrow f_X(w)$; $X[w] \leftarrow g^e$
  For $i = 1, \ldots, d$ do
    $\mathsf{xind} \leftarrow f_I(\mathsf{rind}_i))$
    $H[\mathsf{rind}_i, w] \leftarrow X[w]^{\mathsf{xind}}$
    $\boxed{\boxed{H[\mathsf{rind}_i, w] \xleftarrow{\$} G}}$
  End
End
For $w \in \mathsf{W}$
  $(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_w}) \leftarrow \mathsf{RDB}(w)$
  $K_e \xleftarrow{\$} \{0,1\}^\lambda$; $\mathbf{t} \leftarrow \perp$
  $\sigma \xleftarrow{\$} \mathsf{Perm}([T_w])$; $\mathsf{WPerms}[w] \leftarrow \sigma$
  For $c = 1, \ldots, T_w$ do
    $\mathsf{xind}' \leftarrow f_I(\mathsf{rind}'_{\sigma(c)}))$; $z \leftarrow f_Z(w \,\|\, c)$
    $y \leftarrow \mathsf{xind}' \cdot z^{-1} \bmod p$
    $\boxed{y \xleftarrow{\$} Z_p^*;}$ $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, 0^\lambda)$
    $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  End
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
  For $u \in \mathsf{W} \setminus \{w\}$ do
    For $c = T_w + 1, \ldots, T$ do
      $Y[w, u, c] \leftarrow X[u]^{f_Z(w \,\|\, c)}$
      $\boxed{\boxed{Y[w, u, c] \xleftarrow{\$} G}}$
    End
  End
End
$(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$
For $i = 1, \ldots, Q$ do
  $\mathsf{STags}[i] \leftarrow \mathsf{TSetGetTag}(K_T, \mathbf{s}[i])$
End
$\mathsf{XSet} \leftarrow \mathsf{XSetSetup}(\mathsf{RDB}, H)$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet})$
For $i = 1, \ldots, Q$ do
  $\mathbf{t}[i] \leftarrow \mathsf{GenTrans}(\mathsf{EDB}, H, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$
End
Return $(\mathsf{EDB}, \mathbf{t})$

$\underline{\mathsf{XSetSetup}(\mathsf{RDB}, H)}$ // G_3, G_4, G_5
$(\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{RDB}$
$\mathsf{XSet} \leftarrow \emptyset$
For each $w \in \mathsf{W}$ do
  For each $\mathsf{rind} \in \mathsf{RDB}(w)$ do
    $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{H[\mathsf{rind}, w]\}$
  End
End
Return XSet

$\underline{\mathsf{GenTrans}(\mathsf{EDB}, H, s, x, \mathsf{stag})}$ // G_3, G_4, G_5
$\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
$(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_s}) \leftarrow \mathsf{RDB}(s)$
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{stag}, \mathsf{xtoken}))$
$\sigma \leftarrow \mathsf{WPerms}[s]$
For $c = 1, \ldots, T_s$ do
  $(y, \mathsf{e}) \leftarrow \mathbf{t}[c]$; $\mathsf{xtoken}[c] \leftarrow H[\mathsf{rind}_{\sigma(c)}, x]^{1/y}$
End
For $c = T_s + 1, \ldots, T$ do
  $\mathsf{xtoken}[c] \leftarrow Y[s, x, c]$
End
$\mathsf{ResRinds} \leftarrow \mathsf{RDB}(s) \cap \mathsf{RDB}(x)$
Return $((\mathsf{stag}, \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$

Figure 6: Games G_3, G_4 and G_5. G_5 includes the doubly boxed code and singly boxed code, G_4 includes only the singly boxed code, and G_3 includes neither.

Thus the value $f_I(\mathsf{rind}'_{\sigma(c)})/f_Z(w \,\|\, c)$ which is placed in $Y[w, c]$ is also uniform and independent of the rest of the game, which justifies the change to simply selecting it at random. (We stress that

$\underline{\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})}$ // G_6, G_7, G_8
$\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$
$(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{DB}$
For $i = 1, \ldots, d$ do $\mathsf{rind}_i \leftarrow \pi(\mathsf{ind}_i)$
$\mathsf{RDB} \leftarrow (\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d$
For $w \in \mathsf{W}$ and $i \in [d]$ do
  $H[\mathsf{rind}_i, w] \xleftarrow{\$} G$
End
For $w \in \mathbf{s}$ do $\mathsf{WPerms}[w] \xleftarrow{\$} \mathsf{Perm}([T_s])$
For $w \in \mathsf{W}$ do
  $K \xleftarrow{\$} \{0,1\}^\lambda$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do
    $y \xleftarrow{\$} Z_p^*$; $\mathsf{e} \leftarrow \mathsf{Enc}(K, 0^\lambda)$; $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  End
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_T(\mathcal{L}_T(\mathsf{DB}, \mathbf{s}), \mathbf{T}[\mathbf{s}])$
$\mathsf{XSet} \leftarrow \mathsf{XSetSetup}(\mathsf{RDB}, H)$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet})$
For $i = 1, \ldots, Q$ do
  $\mathbf{t}[i] \leftarrow \mathsf{GenTrans}(\mathsf{EDB}, H, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$
End
Return $(\mathsf{EDB}, \mathbf{t})$

$\underline{\mathsf{XSetSetup}(\mathsf{RDB}, H)}$ // G_7, G_8
$\mathsf{XSet} \leftarrow \emptyset$
For $w \in \mathbf{x}$ do
  For $\mathsf{rind} \in \mathsf{RDB}(w)$ do
    If $\exists i : \mathsf{rind} \in \mathsf{RDB}(\mathbf{s}[i]) \wedge \mathbf{x}[i] = w$ then
      $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{H[\mathsf{rind}, w]\}$
    Else
      $h \xleftarrow{\$} G$; $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{h\}$
  End
End
Return XSet

$\underline{\mathsf{GenTrans}(\mathsf{EDB}, H, s, x, \mathsf{stag}, i)}$ // G_8
$\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$
$(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_s}) \leftarrow \mathsf{RDB}(s)$; $\sigma \leftarrow \mathsf{WPerms}[s]$
For $c = 1, \ldots, T_s$ do
  $(y, \mathsf{e}) \leftarrow \mathbf{t}[c]$
  If $\mathsf{rind}'_{\sigma(c)} \in \mathsf{RDB}(s) \cap \mathsf{RDB}(x)$ then
    $\mathsf{xtoken}[c] \leftarrow H[\mathsf{rind}'_{\sigma(c)}, x]^{1/y}$
  Else If $\exists j \neq i : \mathsf{rind}'_{\sigma(c)} \in \mathsf{RDB}(\mathbf{s}[j]) \wedge \mathbf{x}[j] = x$ then
    $\mathsf{xtoken}[c] \leftarrow H[\mathsf{rind}'_{\sigma(c)}, x]^{1/y}$
  Else
    $\mathsf{xtoken}[c] \xleftarrow{\$} G$
End
For $c = T_s + 1, \ldots, T$ do $\mathsf{xtoken}[c] \xleftarrow{\$} G$
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{stag}, \mathsf{xtoken}))$
$\mathsf{ResRinds} \leftarrow \mathsf{RDB}(s) \cap \mathsf{RDB}(x)$
Return $((\mathsf{stag}, \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$

Figure 7: Games G_6, G_7, G_8.

the random function f_Z *is* evaluated at two different lines of code in Initialize, but these lines always use distinct inputs $(w \,\|\, c)$.)

Game G_5. In G_5, instead of computing the values of the H and Y arrays as before, they are all selected at random from G. We claim there exist an efficient adversary B_5 such that

$$\Pr[G_5 = 1] - \Pr[G_4 = 1] \leq \mathbf{Adv}^{\mathrm{ddh}}_{G,B_5}(\lambda)$$

Let A be an adversary running in G_4 or G_5. Using A, we build an adversary B_5 for the DDH problem. By Lemma 3, it suffices to build an adversary B'_5 that solves the extended version of DDH defined in the lemma.

B_5' takes as input $(g^{\mathbf{a}}, g^{\mathbf{b}}, \mathbf{M})$ where $g^{\mathbf{a}} \in G^m$, $g^{\mathbf{b}} \in G^{d+T\cdot m}$ and $\mathbf{M} \in G^{m\times(d+T\cdot m)}$. Intuitively, it will use $g^{\mathbf{a}}$ vector in place the X array in the games, and the entries from $g^{\mathbf{b}}$ will play two rolls: The discrete logarithms of the first d entries will take the place of the xind values in the loop computing the H array, and the discrete logarithms of the rest of the values will take the place of the $f_Z(w \,\|\, c)$ values in the loop populating Y (note that some entries of $g^{\mathbf{b}}$ may not be needed, depending on the structure of the DB input).

B_5' starts by running A, which outputs $(\mathsf{DB}, (\mathbf{s}, \mathbf{x}))$. B then simulates G_5, except for the following changes: It does not choose f_X and f_I, and the loop that populates the X and H arrays is instead computed by taking the i-th keyword w (in arbitrary order), it places $g^{\mathbf{a}[i]}$ in $X[w]$, and for the ind_j, it places $\mathbf{M}[i, j]$ in $H[\mathsf{rind}_i, w]$. Finally, when computing the Y array, for the entry at (w, u, c), where w is the i-th keyword in the outer loop (in the same order as the H array loop) and u is the j-th keyword in the inner loop, it uses the value from $\mathbf{M}[i, d + j \cdot T + c]$.

It simulates the rest of the game exactly as specified, and finally outputs whatever A outputs. If the matrix $\mathbf{M}$ is computed as $\mathbf{M} = g^{\mathbf{a}\mathbf{b}^{\mathrm{T}}}$, then it is apparent that B_5' simulates G_4 for A, and thus

$$\Pr[B_5'(g, g^{\mathbf{a}}, g^{\mathbf{b}}, g^{\mathbf{a}\mathbf{b}^{\mathrm{T}}}) = 1] = \Pr[G_4 = 1].$$

If instead $\mathbf{M}$ is selected at random from $G^{m\times(d+mT)}$, then B_5' simulates G_5, so

$$\Pr[B_5'(g, g^{\mathbf{a}}, g^{\mathbf{b}}, \mathbf{M}) = 1] = \Pr[G_5 = 1].$$

Subtracting these inequalities shows that B_5' satisfies the desired relation.

Game G_6. The $\mathsf{Initialize}$ code for G_6 is described in Figure 7, and its other routines remain as defined in G_5. G_6 deletes some irrelevant code and generates the TSet using a simulator which we will show to be guaranteed to exist by the TSet security notion.

Before describing G_6 in detail, we consider the following TSet adversary B_6. In the T-set games, B_6 starts by generating $(\mathsf{DB}, \mathbf{s}, \mathbf{T})$ exactly as described in G_5. Upon receiving $(\mathsf{TSet}, \mathsf{STags})$ from its game, B_6 continues to simulate the game with these values until A halts, and it outputs whatever A output. By the $\mathcal{L}_T$-security of the T-set implementation, there exists an efficient simulator $\mathcal{S}_T$ for B_6.

In game G_6, we replace the call to $\mathsf{TSetSetup}$ and loop generating STags with a call to the simulator $\mathcal{S}_T$ on input $(\mathcal{L}_T(\mathsf{DB}, \mathbf{s}), \mathbf{T}[\mathbf{s}])$. By the construction of G_5 and G_6, we have that the T-set real game with adversary B_6 perfectly simulates G_5, while the T-set ideal game with B_6 perfectly simulates G_6. Thus a non-negligible difference in $\Pr[G_6 = 1]$ and $\Pr[G_5 = 1]$ would contradict the assumed $\mathcal{L}_T$-security of the T-set implementation, giving

$$\Pr[G_6 = 1] - \Pr[G_5 = 1] \leq \mathrm{neg}(\lambda).$$

Games G_7 and G_8. Games G_7 and G_8 change the way the H array is accessed in a way that will enable the final simulator to work with its given leakage. Intuitively, now whenever the game access the H array at an index (rind, x), it first tests to see if the game will ever access that index in H again. If it will come back to this position, it uses the value from H. It not, then the game replaces the H access with a random choice. Since that was to be the only usage of that position of H during the game, this doesn't affect the distribution of the game.

We now describe how this is implemented. G_7 only changes the way XSet works. Recall that after H is populated, it is used in two places: In $\mathsf{XSetSetup}$ and in $\mathsf{GenTranscript}$. The routine $\mathsf{XSetSetup}$ will never repeat an access to H, so for an index (rind, w) it only needs to test if $\mathsf{GenTranscript}$ will read this position. But $\mathsf{GenTranscript}$ will only read positions such that

$\mathsf{rind} \in \mathsf{RDB}(\mathbf{s}[i])$ and $x = \mathbf{x}[i]$ for some i, and this is exactly what XSetSetup tests for in G_7. By this observation and the discussion above, we have

$$\Pr[G_7 = 1] = \Pr[G_6 = 1].$$

For G_8 we change the way GenTranscript accesses H. To detect a possible repeated access, it must test if either XSetSetup will read that index, or if GenTranscript will read it again. In G_7 we modified XSetSetup so that it only accesses a position $H[\mathsf{rind}, w]$ if $\mathsf{rindRDB}(\mathbf{s}[i]) \cap \mathsf{RDB}(\mathbf{x}[i])$, so the first "If" statement in the G_8 version of GenTranscript tests for this. What remains to calculate are the positions which GenTranscript will access twice. First, we have that a repeated position must occur during two different calls to GenTranscript because one execution of the subroutine only touches unique indices of H. So for an index (rind, x) to be accessed twice, we must have that rind is a member of both $\mathsf{RDB}(\mathbf{s}[i])$ and $\mathsf{RDB}(\mathbf{s}[j])$, for some $i \neq j$. For the x to repeat, we must have $\mathbf{x}[i] = \mathbf{x}[j]$. This condition for a repeated access is exactly what the "Else If" statement in GenTranscript tests for. An argument similar to the one for the previous game transition gives

$$\Pr[G_8 = 1] = \Pr[G_7 = 1].$$

Game G_9. The final game, G_9, computes the same distribution as G_8, but processes the XSet and $\mathbf{t}$ in a way that is closer to what the simulator will do. The purpose of the changes here is to avoid requiring complete knowledge of $\mathsf{RDB}(\mathbf{s}[i])$ (the list of rind values matching $\mathbf{s}[i]$) while running the game, and instead work with the result pattern and intersection. Below let $\mathsf{RP}(\mathsf{DB}) = (R_1, \ldots, R_Q)$ and $\mathsf{IP}(\mathsf{DB}) = (S_1, \ldots, S_Q)$ as defined in the leakage profile $\mathcal{L}_{\text{oxt}}$.

In G_9, Initialize is unchanged. The changes to XSet are straightforward. The XSetSetup routine explicitly uses the result pattern $\mathsf{RP}_\pi(\mathsf{DB})$ and inserts the same entries from H into the XSet. Now the random additions to XSet are deferred to a later loop which does them all at once.

The changes to GenTranscript are more subtle. Before describing them, let us examine the computation in G_8 more closely. There GenTranscript computed $(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T_s}) \leftarrow \mathsf{RDB}(s)$, then permuted this array using $\sigma = \mathsf{WPerms}[s]$, and then used a subset of those values to index into the H array and fill in xtoken and ResRinds. This subset is determined by the code in the first "For" loop: The first "If" statement is true when $\mathsf{rind}_{\sigma(c)} \in R_i$, and the "Else If" is true when when $\mathsf{rind}_{\sigma(c)} \in S_i$. For the remaining values of c, $\mathsf{xtoken}[c]$ is filled with a random group element.

G_9 accomplishes the same results, but while only knowing the subset $R_i \cup S_i$ that cause the "If" statements to return true. One subtlety that must be handled is that in G_8, the permutation σ is reused across multiple calls to GenTranscript when the value of s repeats. So, in processing GenTranscript, G_9 first computes all of the rind values that will return true in any call to GenTranscript with the same value of s, and then randomly permutes all of the values and stores them $\mathbf{R}$, and then uses the ones relevant to the current query in the correctly permuted order. The rest of the processing is done exactly as in G_8, so we have

$$\Pr[G_9 = 1] = \Pr[G_8 = 1].$$

We complete the proof by giving an algorithm $\mathcal{S}$ that, given $\mathcal{L}(\mathsf{DB}, \mathbf{s}, \mathbf{x})$, outputs exactly the same distribution as $\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})$ in G_9. By collecting the relations between the games, we will show that $\mathcal{S}$ satisfies the requirement in the theorem.

$\mathcal{S}$ takes as input $\mathcal{L}(\mathsf{DB}, \mathbf{s}, \mathbf{x})$, which consists of $(N, \overline{\mathbf{s}}, \mathsf{SP}, \mathsf{RP}_\pi, \mathsf{IP}_\pi, \mathcal{L}_T(\mathsf{DB}, \mathbf{s}))$ and must output a simulated $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$ and transcript array $\mathbf{t}$.

Our simulator will use the IP to compute compute what we call the *restricted equality pattern* of $\mathbf{x}$, denoted $\widehat{\mathbf{x}}$. The vector $\widehat{\mathbf{x}} \in [m]^Q$ is computed as follows. First define $\mathbf{x}[i] \equiv \mathbf{x}[j]$ iff $S_i \cap S_j \neq \emptyset$,

$\underline{\mathsf{Initialize}(\mathsf{DB}, \mathbf{s}, \mathbf{x})}$ // G_9
$\pi \xleftarrow{\$} \mathsf{Perm}(\{0,1\}^\lambda)$
$(\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d \leftarrow \mathsf{DB}$
For $i = 1, \ldots, d$ do $\mathsf{rind}_i \leftarrow \pi(\mathsf{ind}_i)$
$\mathsf{RDB} \leftarrow (\mathsf{rind}_i, \mathsf{W}_i)_{i=1}^d$
For $w \in \mathbf{x}$ and $i \in [d]$ do
  $H[\mathsf{rind}_i, w] \xleftarrow{\$} G$
End
For $w \in \mathbf{s}$ do $\mathsf{WPerms}[w] \xleftarrow{\$} \mathsf{Perm}([T_s])$
For $w \in \mathsf{W}$ do
  $K \xleftarrow{\$} \{0,1\}^\lambda$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do
    $y \xleftarrow{\$} Z_p^*$; $\mathsf{e} \leftarrow \mathsf{Enc}(K, 0^\lambda)$; $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  End
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_T(\mathcal{L}_T(\mathsf{DB}, \mathbf{s}), \mathbf{T}[\mathbf{s}])$
$\mathsf{XSet} \leftarrow \mathsf{XSetSetup}(\mathsf{RDB}, H)$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathsf{XSet})$
For $i = 1, \ldots, Q$ do
  $\mathbf{t}[i] \leftarrow \mathsf{GenTranscript}(\mathsf{RDB}, H, \mathbf{s}[i], \mathbf{x}[i], \mathsf{STags}[i])$
End
Return $(\mathsf{EDB}, \mathbf{t})$

$\underline{\mathsf{XSetSetup}(\mathsf{RDB}, H)}$ // G_9
$\mathsf{XSet} \leftarrow \emptyset$
$(R_1, \ldots, R_Q) \leftarrow \mathsf{RP}_\pi(\mathsf{DB})$
$T' \leftarrow |\bigcup_{i \in [Q]} R_i|$
For $i = 1, \ldots, Q$ do
  For $\mathsf{rind} \in R_i$ do
    $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{H[\mathsf{rind}, \mathbf{x}[i]]\}$
  End
End
For $i = T' + 1, \ldots, T$ do
  $h \xleftarrow{\$} G$; $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{h\}$
End
Return XSet

$\underline{\mathsf{GenTrans}(\mathsf{EDB}, H, s, x, \mathsf{stag})}$ // G_9
$\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$; $\sigma \leftarrow \mathsf{WPerms}[s]$
$(R_1, \ldots, R_Q) \leftarrow \mathsf{RP}_\pi(\mathsf{DB})$
$(S_1, \ldots, S_Q) \leftarrow \mathsf{IP}_\pi(\mathsf{DB})$
$R \leftarrow \emptyset$; $T_s \leftarrow |\mathsf{DB}(s)|$; $\mathsf{xtoken} \leftarrow \perp$
For $j = 1, \ldots, Q$ do
  If $\mathbf{s}[j] = s$ then $R \leftarrow R \cup R_j \cup S_j$
End
$(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T'}) \leftarrow R$
$\mathbf{R} \leftarrow \perp$
For $c = 1, \ldots, T'$ do $\mathbf{R}[\sigma^{-1}(c)] \leftarrow \mathsf{rind}'_c$
For $c = 1, \ldots, T_s$ do
  $(y, \mathsf{e}) \leftarrow \mathbf{t}[c]$
  If $\mathbf{R}[c] \in R_i \cup S_i$ then
    $\mathsf{xtoken}[c] \leftarrow H[\mathbf{R}[c], x]^{1/y}$
  Else
    $\mathsf{xtoken}[c] \xleftarrow{\$} G$
End
For $c = T_s + 1, \ldots, T$ do $\mathsf{xtoken}[c] \xleftarrow{\$} G$
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{stag}, \mathsf{xtoken}))$
$\mathsf{ResRinds} \leftarrow R_i$
Return $((\mathsf{stag}, \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$

Figure 8: Game G_9.

and then make $\equiv$ an equivalence relation by taking its transitive closure. Then sort the partitions of the equivalence relation by their least element, and assign $\widehat{\mathbf{x}}[i]$ the index of the partition containing i.

Using IP_π and the procedure discussed in the leakage profile description earlier, $\mathcal{S}$ computes the restricted equality pattern for $\mathbf{x}$ and stores it as $\widehat{\mathbf{x}}$. Then $\mathcal{S}$ lets $\{\mathsf{rind}_1, \ldots, \mathsf{rind}_d\}$ be the set $\bigcup_{i=1}^d R_i$ along with enough unique random rinds to bring the set size to d. It begins by populating the H, WPerms, and $(\mathsf{TSet}, \mathsf{STags})$ via

For $w \in \widehat{\mathbf{x}}$ and $i \in [d]$ do $H[\mathsf{rind}_i, w] \xleftarrow{\$} G$

For $w \in \overline{\mathbf{s}}$ do $\mathsf{WPerms}[w] \xleftarrow{\$} \mathsf{Perm}([\mathsf{SP}[i]])$
For $w \in \overline{\mathbf{s}}$ do
  $K \xleftarrow{\$} \{0,1\}^\lambda$; $\mathbf{t} \leftarrow \perp$
  For $c = 1, \ldots, T_w$ do $y \xleftarrow{\$} Z_p^*$; $\mathsf{e} \leftarrow \mathsf{Enc}(K, 0^\lambda)$; $\mathbf{t}[c] \leftarrow (y, \mathsf{e})$
  $\mathbf{T}[w] \leftarrow \mathbf{t}$
End
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_T(\mathcal{L}_T(\mathsf{DB}, \mathbf{s}), \mathbf{T}[\overline{\mathbf{s}}])$.

We stress that $\mathcal{S}$ is using $\widehat{\mathbf{x}}$ and $\overline{\mathbf{s}}$ instead of $\mathbf{x}$ and $\mathbf{s}$ in populating H and WPerms. Below we will address this discrepancy by showing that this does not change the distribution of the game.

Next it computes XSet via

$\mathsf{XSet} \leftarrow \emptyset$
$(R_1, \ldots, R_Q) \leftarrow \mathsf{RP}_\pi(\mathsf{DB})$
$T' \leftarrow |\bigcup_{i \in [Q]} R_i|$
For $i = 1, \ldots, Q$ do
  For $\mathsf{rind} \in R_i$ do $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{H[\mathsf{rind}, \widehat{\mathbf{x}}[i]]\}$
End
For $i = T' + 1, \ldots, T$ do $h \xleftarrow{\$} G$; $\mathsf{XSet} \leftarrow \mathsf{XSet} \cup \{h\}$
Return XSet.

This code is exactly the same as $\mathsf{XSetSetup}$ in G_9, except that the H array is indexed using $\widehat{\mathbf{x}}$. This completes the generation of $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$. What remains is to compute $\mathbf{t}$. To fill in $\mathbf{t}[i]$, $\mathcal{S}$ computes

$\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{STags}[i])$; $\sigma \leftarrow \mathsf{WPerms}[\overline{\mathbf{s}}[i]]$
$S \leftarrow \emptyset$; $\mathsf{xtoken} \leftarrow \perp$; $T_s \leftarrow \mathsf{SP}[i]$
For $j = 1, \ldots, Q$ do If $\overline{\mathbf{s}}[j] = \overline{\mathbf{s}}[i]$ then $R \leftarrow R \cup R_j$
$(\mathsf{rind}'_1, \ldots, \mathsf{rind}'_{T'}) \leftarrow R \cup S_i$; $\mathbf{R} \leftarrow \perp$
For $c = 1, \ldots, T'$ do $\mathbf{R}[\sigma^{-1}(c)] \leftarrow \mathsf{rind}'_c$
For $c = 1, \ldots, T_s$ do
  If $\mathbf{R}[c] \in R_i \cup S_i$ then $(y, \mathsf{e}) \leftarrow \mathbf{t}[c]$; $\mathsf{xtoken}[c] \leftarrow H[\mathbf{R}[c], \widehat{\mathbf{x}}]^{1/y}$
  Else $\mathsf{xtoken}[c] \xleftarrow{\$} G$
End
For $c = T_s + 1, \ldots, T$ do $\mathsf{xtoken}[c] \xleftarrow{\$} G$
$\mathsf{Res} \leftarrow \mathsf{ServerSearch}(\mathsf{EDB}, (\mathsf{STags}[i], \mathsf{xtoken}))$; $\mathsf{ResRinds} \leftarrow R_i$
Return $((\mathsf{STags}[i], \mathsf{xtoken}), \mathsf{Res}, \mathsf{ResRinds})$.

The only meaningful difference between this code and the code in G_9 is the usage of $\overline{\mathbf{s}}$ in place of $\mathbf{s}$ and $\widehat{\mathbf{x}}$ in place of $\mathbf{x}$. This completes the description of $\mathcal{S}$.

We claim that $\mathcal{S}$ outputs exactly the same distribution as $\mathsf{Initialize}$ in G_9. This amounts to showing that using $\overline{\mathbf{s}}$ and $\widehat{\mathbf{x}}$ does not change anything. We first claim that a version of $\mathcal{S}$ that uses $\overline{\mathbf{s}}$ and $\mathbf{x}$ (but not $\widehat{\mathbf{x}}$) results in the same distribution as in G_9. Note that $\overline{\mathbf{s}}$ is only used in three places: In populating WPerms, in reading from WPerms, and in the first "For" loop of the code generating $\mathbf{t}[i]$. The key observation is that for all $i, j \in [Q]$, $\mathbf{s}$ and $\overline{\mathbf{s}}$ satisfy

$$\mathbf{s}[i] = \mathbf{s}[j] \iff \overline{\mathbf{s}}[i] = \overline{\mathbf{s}}[j].$$

Therefore the usage of $\overline{\mathbf{s}}$ does not change the computation of the R set by $\mathcal{S}$ versus what is done in G_9. Now we examine the distribution of the σ permutations used in generating $\mathbf{t}[i]$ in G_9. Let σ_i be the permutation read from WPerms when generating $\mathbf{t}[i]$. The marginal distribution of each σ_i is uniform, but the distribution of the vector $(\sigma_1, \ldots, \sigma_Q)$ is not. Specifically, the vector is uniform except that whenever $\mathbf{s}[i] = \mathbf{s}[j]$, we will have $\sigma_i = \sigma_j$. Now we can see that the $\mathcal{S}$ simulation will

use the same distribution of $(\sigma_1, \ldots, \sigma_Q)$ values, because it takes each σ_i to be uniform, except that it reuses them whenever $\overline{\mathbf{s}}[i] = \overline{\mathbf{s}}[j]$, which has the same truth value as $\mathbf{s}[i] = \mathbf{s}[j]$.

We claim that using $\widehat{\mathbf{x}}$ will not change the distribution. This vector is only used when indexing the H array. By an argument similar to the one above for the WPerms array, it is sufficient to show that the equality relations between the indexes are the same in G_9 and in the $\mathcal{S}$ simulation. Consider any two indices $(\mathsf{rind}_1, \mathbf{x}[i])$ and $(\mathsf{rind}_2, \mathbf{x}[j])$ read by G_9, and let $(\mathsf{rind}_1, \widehat{\mathbf{x}}[i])$ and $(\mathsf{rind}_2, \widehat{\mathbf{x}}[j])$ be the indices that $\mathcal{S}$ would read instead. We claim that

$$(\mathsf{rind}_1, \mathbf{x}[i]) = (\mathsf{rind}_2, \mathbf{x}[j]) \iff (\mathsf{rind}_1, \widehat{\mathbf{x}}[i]) = (\mathsf{rind}_2, \widehat{\mathbf{x}}[j]), \tag{1}$$

which will show that the simulation is the same. This will follow easily from the following lemma, which follows directly from the way $\widehat{\mathbf{x}}$ is computed.

Lemma 9 *Let* $\mathsf{DB} = (\mathsf{ind}_i, \mathsf{W}_i)_{i=1}^d$ *satisfy the syntax for a database,* $\pi \in \mathsf{Perm}(\{0,1\}^\lambda)$*, and* $\mathbf{s}, \mathbf{x} \in \mathsf{W}^Q$ *and let* $\widehat{\mathbf{x}}$ *be the restricted equality pattern. Then for all* $i, j \in [Q]$*,*

$$\widehat{\mathbf{x}}[i] = \widehat{\mathbf{x}}[j] \quad \Rightarrow \quad \mathbf{x}[i] = \mathbf{x}[j]$$

and

$$(\mathbf{x}[i] = \mathbf{x}[j]) \wedge (I_\pi(\mathbf{s}[i]) \cap I_\pi(\mathbf{s}[j]) \neq \emptyset) \quad \Rightarrow \quad \widehat{\mathbf{x}}[i] = \widehat{\mathbf{x}}[j].$$

To prove (1), consider any two such queries. The $\Leftarrow$ direction is easy by Lemma 9. For the $\Rightarrow$ direction, suppose that $(\mathsf{rind}_1, \mathbf{x}[i]) = (\mathsf{rind}_2, \mathbf{x}[j])$. Then we have $\mathsf{rind}_1 = \mathsf{rind}_2$, and also that they are members of $(S_i \cup R_i) \cap (S_j \cup R_j)$ because the games only ever use rind values from these sets as indices. But this means the rind value is in $I_\pi(\mathbf{s}[i]) \cap I_\pi(\mathbf{s}[j])$, meaning this intersection is not empty, and thus we have $\widehat{\mathbf{x}}[i] = \widehat{\mathbf{x}}[j]$ by the second conclusion in Lemma 9.

C T-Set Implementation

We show an efficient implementation of a T-set, whose syntax and security are defined in Section 2. We implement a T-set as a hash table with B buckets of size S each. The $\mathsf{TSetSetup}(\mathbf{T})$ procedure sets the parameters B and S depending on the total number $N = \sum_{w \in \mathsf{W}} |\mathbf{T}[w]|$ of tuples in $\mathbf{T}$ in such a way so that (1) the probability of an overflow of any bucket after storing N elements in this hash table is a sufficiently small constant; and (2) the total size $B \cdot S$ of the hash table is $O(N)$.

Figure 9 shows our T-set implementation $\Sigma = (\mathsf{TSetSetup}, \mathsf{TSetGetTag}, \mathsf{TSetRetrieve})$. We use the following notation: λ is a security parameter, $n(\lambda)$ is the bit length of strings s_i in each vector $\mathbf{T}$, record is a record type with two fields, label, storing bit strings of length λ, and value, storing bit strings of length $n(\lambda)+1$. Each entry $\mathsf{TSet}[i]$, for $i = 1, ..., B$, of array TSet, is itself an array of S records of type record. We denote the j-th element in the array $\mathsf{TSet}[i]$ as $\mathsf{TSet}[i, j]$. The protocol uses PRFs $F, \bar{F}$ and a hash function H, which for adaptive security is modeled as a random oracle in the analysis, where F maps integers between 1 and $\max_w |D[w]|$ onto $\{0,1\}^\lambda$, $\bar{F}$ maps the set W onto the key space of F, and H maps $\{0,1\}^\lambda$ onto $\{1, ..., B\} \times \{0,1\}^\lambda \times \{0,1\}^{n(\lambda)+1}$.

T-Set correctness. Note that procedure $\mathsf{TSetSetup}$ places the i-th element s_i of $\mathbf{T}[w]$ into bucket $\mathsf{TSet}[b]$ where b is the first element in tuple $(b, L, K) = H(F(\mathsf{stag}, i))$, for $\mathsf{stag} = \bar{F}(K_T, w)$. Therefore the search procedure $\mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$ will recover the same bucket $\mathsf{B} = \mathsf{TSet}[b]$ for $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w)$. Since that i-th element s_i was placed in a record r in $\mathsf{TSet}[b]$ s.t. $r.\mathsf{label} = L$, the same record is found in the $\mathsf{TSetRetrieve}$ procedure as $\mathsf{B}[j]$, unless bucket B contains some other record with the same label field L. However, the probability that this ever happens can be bounded by $BS^2 2^{-\lambda}$ (plus at most a negligible quantity bounding the maximal

$\underline{\mathsf{TSetSetup}(\mathbf{T})}$

- Initialize an array TSet of size B whose every element is an array of S records of type record.
- Initialize an array Free of size B whose elements are integer sets, initially all set to $\{1, ..., S\}$.
- Choose a random key K_T of PRF $\bar{F}$.
- Let W be the set of keywords in DB. For every $w \in \mathsf{W}$ do the following:
  - Set $\mathsf{stag} \leftarrow \bar{F}(K_T, w)$ and $\mathbf{t} \leftarrow \mathbf{T}[w]$.
  - For each $i = 1, ..., |\mathbf{t}|$, set s_i as the i-th string in $\mathbf{t}$, and perform the following steps:
    - Set $(b, L, K) \leftarrow H(F(\mathsf{stag}, i))$.
    - If $\mathsf{Free}[b]$ is an empty set, restart $\mathsf{TSetSetup}(\mathbf{T})$ with fresh key K_T.
    - Choose $j \xleftarrow{\$} \mathsf{Free}[b]$ and remove j from set $\mathsf{Free}[b]$, i.e. set $\mathsf{Free}[b] \leftarrow \mathsf{Free}[b] \setminus \{j\}$.
    - Set bit β as 1 if $i < |\mathbf{t}|$ and 0 if $i = |\mathbf{t}|$.
    - Set $\mathsf{TSet}[b, j].\mathsf{label} \leftarrow L$ and $\mathsf{TSet}[b, j].\mathsf{value} \leftarrow (\beta | s_i) \oplus K$.
- Output (TSet, K_T).

$\underline{\mathsf{TSetGetTag}(K_T, w)}$

- Output $\mathsf{stag} \leftarrow \bar{F}(K_T, w)$

$\underline{\mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})}$

- Initialize $\mathbf{t}$ as an empty list, bit β as 1, and counter i as 1.
- Repeat the following loop while $\beta = 1$:
  - Set $(b, L, K) \leftarrow H(F(\mathsf{stag}, i))$ and retrieve an array $\mathsf{B} \leftarrow \mathsf{TSet}[b]$
  - Search for index $j \in \{1, ..., S\}$ s.t. $\mathsf{B}[j].\mathsf{label} = L$.
  - Let $v \leftarrow \mathsf{B}[j].\mathsf{value} \oplus K$. Let β be the first bit of v, and s the remaining $n(\lambda)$ bits of v.
  - Add string s to the list $\mathbf{t}$ and increment i.
- Output $\mathbf{t}$.

Figure 9: T-Set Implementation Σ

distinguishing advantage between pseudorandom functions F and $\bar{F}$ and true random functions). If the correct record r is found then the $\mathsf{TSetRetrieve}$ procedure uncovers the correct bit β and string s_i from $r.\mathsf{value}$ because it is xored by the same one-time pad K, the third element in the same triple (b, L, K), with which $(\beta | s_i)$ was encrypted in $r.\mathsf{value}$ during $\mathsf{TSetSetup}$. Since $\beta = 0$ if and only if s_i is the last string in list $\mathbf{T}[w]$, $\mathsf{TSetRetrieve}$ finds all elements of $\mathbf{T}[w]$ and terminates when $\mathbf{t} = \mathbf{T}[w]$.

Note that since a single attempt to build the TSet table by procedure $\mathsf{TSetSetup}$ takes $O(m)$ steps, and parameters B, S are set so that a single attempt fails with at most a constant probability over the choice of the key K_T of PRF $\bar{F}$, the expected time for the $\mathsf{TSetSetup}$ procedure is $O(m)$.

T-Set security. The only leakage incurred by T-set implementation of Figure 9 is the total number $N = \sum_{w \in \mathsf{W}} |\mathbf{T}[w]|$ of tuples in $\mathbf{T}$, which in our SSE applications equals the total number $\sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$ of keyword occurrences in database DB.

Theorem 10 *For any keyword sequence* $\mathbf{q}$*, including an empty sequence, define* $\mathcal{L}_T(\mathbf{T}, \mathbf{q})$ *as* $\sum_{w \in \mathsf{W}} |\mathbf{T}(w)|$*, where* W *is the set of keywords used by* $\mathbf{T}$*. The T-set implementation* Σ *in Figure 9 is* $\mathcal{L}_T$*-*

adaptively-secure assuming that F and $\bar{F}$ are secure PRFs and that H is a random oracle.

Dispensing of the random oracle. Theorem 10 is the only result in this paper that uses the random oracle model. The model is used to prove adaptive security (and indeed to avoid the lower bound of [13] on token lengths). For non-adaptive security, we can prove security for a version of the protocol with H replaced with another PRF (and an additional key stored at the client). Moreover, it is possible to prove adaptive security without a random oracle at the cost of added communication. For this we dispense of H and let the client send the values $F(\mathsf{stag}, i)$ instead of stag. This increases the amount of communication, but when used with our OXT protocol it is not a dramatic increase. As with OXT, the server can send a "stop" message when the client has sent enough $F(\mathsf{stag}, i)$ values.

Achieving optimal leakage. The leakage $\mathcal{L}_T(\mathbf{T}, \mathbf{q}) = \sum_{w \in \mathsf{W}} |\mathbf{T}(w)|$ can be relaxed to only disclose an upper bound on the latter quantity by filling in all empty locations in all buckets with (pseudo) random values. In that way, the only leakage is the size of the full hash table which is optimal.

Proof sketch. To prove Theorem 10, we start with the simulator algorithm S. The simulator's initial input is $N = \mathcal{L}_T(\mathbf{T}) = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$, and for each query q the adversary makes, the simulator gets the corresponding vector $\mathbf{T}[q]$, and no additional leakage. At initialization $S(N)$ creates TSet as a $B \times S$ table (note that parameters B, S are determined by N) just like the $\mathsf{TSetSetup}$ algorithm, but S populates this table by choosing m random locations in it and filling them up with random entries. In other words, for each $i = 1, \ldots, m$, S randomly chooses a block b and a free location j within this block, and assigns $\mathsf{TSet}[b, j]$ to a record whose label and value fields are random bitstrings of appropriate length. S hands this table TSet over to A, but locally it marks all these m records in TSet as "unrevealed". In the Random Oracle Model for hash function H, S also simulates the outputs of H for A, initially setting $H(x)$ for every query x at random. On every query q which adversary A makes, S receives a list $\mathbf{t} = (s_1, \ldots, s_{|d|})$ s.t. $\mathbf{t} = \mathbf{T}[q]$. (Note that function $\mathcal{L}_T$ does not include any further leakage to S.) $S(\mathbf{t})$ picks value stag' at random in the key space of PRF F, chooses $|\mathbf{t}|$ random unrevealed records $r_1, \ldots, r_{|\mathbf{t}|}$ in TSet, locally marking them all as revealed, and for each $i = 1, \ldots, |\mathbf{t}|$, S sets $H(F(\mathsf{stag}', i))$ to a tuple (b, L, K) defined as follows: Bucket b is set to the bucket where record r_i is, label L is set to $r_i.\mathsf{label}$, and key K is set to $(r_i.\mathsf{value}) \oplus (\beta | s_i)$ where β is set to 1 if $i < |\mathbf{t}|$ and 0 if $i = |\mathbf{t}|$. If H was queried on any of the points $F(\mathsf{stag}', i)$ before, S aborts. Otherwise, S outputs stag' to A as its response to $\mathbf{t} = \mathbf{T}[q]$.

First, note that by the PRF property of $\bar{F}$, the real game is indistinguishable from a modification in which values stag'_w for all $w \in \mathsf{W}$ are replaced by random elements of the range of $\bar{F}$, which is the keyspace of PRF F. Secondly, because of the PRF property of F, this keyspace must be large enough so that there only negligible probability that A queries H on (stag'_w, i) for any $w \in \mathsf{W}$ and any i before A sees the corresponding stag'_w value. Finally, since records corresponding to any given $\mathbf{T}[q]$ sequence are assigned at random to the TSet table, under the condition that H isn't queried on any point (stag', i) before S releases the corresponding stag' value to A, the adversary's view of each $(b, L, K) = H(F(\mathsf{stag}', i))$ tuple and the corresponding record r_i is identical in the above simulation and the execution modified above, because in both cases b, L, K are random, $r_i.\mathsf{label} = L$ and $r_i.\mathsf{value} = K \oplus (\beta, s_i)$ where $\beta = 1$ for $i < |\mathbf{t}|$ and 0 for $i = |\mathbf{t}|$. This completes the argument for adaptive security of our T-set implementation.

Overflow probability. Here we provide some examples of overflow probabilities for a given number of buckets and their size. Consider a hash table with B buckets, each with space for S equal-sized items. Let there be N items, which are inserted randomly and independently into the hash table buckets. Let X_i^j denote the indicator variable for the event that the i-th item was inserted in j-th bucket. Then, for each bucket j, the event W^j that it overflows is given by $\sum_{i=1..N} X_i^j > S$. First note that for all i, j, $\Pr[X_i^j = 1] = 1/B$. Thus, by linearity of expectation,

for each j, $\mathbf{E}[\sum_{i=1..N} X_i^j] = N/B$. If k, the *space-overhead*, is defined as $B \cdot S/N$, then the above expectation is S/k. Now by Chernoff bound [2], for $k \geq 1$,

$$\Pr[W^j] = \Pr[\sum_{i=1..N} X_i^j > k \cdot (S/k)] < [e^{k-1}k^{-k}]^{S/k} < (e^{1-1/k}/k)^S.$$

By the union bound, the probability that any of the B buckets overflows is at most $B \cdot (e/k)^S$ which in turn equals $\frac{Nk}{S} \cdot (e^{1-1/k}/k)^S$. To illustrate the overflow probability with specific numerical examples we consider $N = 2^{30}$. Then with $k = 3, S = 80$ one gets overflow probability of less than 2^{-21} (and with $S = 100$ less than 2^{-37}). With $k = 2, S = 160$ the probability is 2^{-21} (and with $S = 200$ less than 2^{-32}). With $k = 1.5, S = 400$ the probability is about 2^{-20} (and with $S = 500$ less than 2^{-30}). Finally, if one uses moderately large buckets, then even with a mere 10% increase in database size one can get very low overflow probabilities; e.g., with $k = 1.1, S = 6000$ the probability is 2^{-20} (and with $S = 8000$ less than 2^{-33}).

Note that this is the probability that pre-processing fails and needs to be re-started; this probability is not adversarially controlled and has no security consequences.

D Oblivious Cross-Tags Protocol PXT Using Bilinear Pairings

EDBSetup(DB)

- Select keys K_S, K_X, K_P, K_I, K_Z, 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 $\mathsf{DB}(w)$ in random order, then:
    - Set $\mathsf{rind} \leftarrow P(K_P, \mathsf{ind})$, $\mathsf{xind} \leftarrow F_p(K_I, \mathsf{rind})$, $z \leftarrow F_p(K_Z, w)$ and $y \leftarrow g^{\mathsf{xind}/z}$.
    - Compute $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rind})$, and append (e, y) to $\mathbf{t}$.
    - Set $\mathsf{xtag} \leftarrow e(g, \mathcal{P})^{F_p(K_X, w) \cdot \mathsf{xind}}$ and add xtag to XSet.
  - $\mathbf{T}[w] \leftarrow \mathbf{t}$.
- $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSetSetup}(\mathbf{T})$.
- Output the key $(K_S, K_X, K_P, K_I, K_Z, K_T)$ and $\mathsf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$.

Search protocol

- The client takes an input the key $(K_S, K_X, K_P, K_I, K_Z, K_T)$ and keywords $w_1, \ldots, w_n$ to query. It computes $\mathsf{stag} \leftarrow \mathsf{TSetGetTag}(K_T, w_1)$ and then for $i = 2, \ldots, n$, $\mathsf{xtoken}_i \leftarrow \mathcal{P}^{F_p(K_Z, w_1) \cdot F_p(K_X, w_i)}$. Sends $(\mathsf{stag}, \mathsf{xtoken}_1, \ldots, \mathsf{xtoken}_n)$ to the server.
- The server computes $\mathbf{t} \leftarrow \mathsf{TSetRetrieve}(\mathsf{TSet}, \mathsf{stag})$. For each tuple (e, y) in $\mathbf{t}$, if $e(y, \mathsf{xtoken}_i) \in \mathsf{XSet}$ for all $i = 2, \ldots, n$, it sends e to the client.
- The client sets $K_e \leftarrow F(K_S, w_1)$, and for each e received, the client computes $\mathsf{rind} \leftarrow \mathsf{Dec}(K_e, \mathsf{e})$ and outputs rind.

Figure 10: PXT: Bilinear Pairings based Oblivious Cross-Tags Protocol

As we saw in the design of OXT in Section 3, the server stores in each tuple corresponding to a keyword w and document xind, a blinded value $y_c = \mathsf{xind} \cdot z_c^{-1}$, where z_c was derived from w and a tuple counter c. This counter served to ensure independent blinding values z_c. Similarly, the Client needed to send an *array* of values xtoken_i defined by $\mathsf{xtoken}[c, i] := g^{F_p(K_X, w_i) \cdot z_c}$.

A natural alternative to using counters to assure independence is to generate $y \leftarrow g^{\mathsf{xind}/z}$, where z is now just derived from w (without any counters). The blinding can be removed from the Client-supplied tokens by using bilinear pairings, if the underlying groups support such pairings. However, we now run into another problem, as the bilinear pairings usually render the DDH assumption invalid, unless we work in bilinear groups $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_T$, with a pairing $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$, and with no easy homomorphism between $\mathcal{G}_1$ and $\mathcal{G}_2$. Such groups do still satisfy DDH, and the assumption is well-known as the SXDH assumption (see [16]).

Thus, the Client can now just send a single xtrap $\mathcal{P}^{F_p(K_X, w_i) \cdot z}$, where $\mathcal{P}$ is a generator of group $\mathcal{G}_2$. However, the DDH assumption is not enough to randomize all y values corresponding to a w (i.e. that have the same z in the exponent) as z is also in the exponent of the revealed xtrap for some other w. It would be intriguing to see if one can build an SSE scheme with leakage limited to what OXT leaks by using just the SXDH assumption, although it seems unlikely that it would be as efficient as the one just described.

Surprisingly, we can show that the above scheme, which we call PXT and that is described in detail in Figure 10, is secure with leakage exactly as for OXT by assuming something stronger than SXDH. We call this stronger assumption the augmented SXDH assumption (SAXDH). This augmented assumption remains static, as opposed to many other dynamic assumptions usually employed to achieve efficiency (e.g. the strong Diffie-Hellman assumption [8], q-BDHI assumption etc.). It is also straightforward to verify that it holds in the generic group model (in particular, it is implied by the generalized DH-problem [7]).

D.1 Augmented SXDH Assumption

Consider cyclic groups $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_T$, with an efficient bilinear pairing $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$. The SXDH assumption says that the groups $\mathcal{G}_1$ and $\mathcal{G}_2$ are DDH groups (which also implies that the target group $\mathcal{G}_T$ is DDH). For our PXT construction we need a generalization of the SXDH assumption which we call the **symmetric augmented external Diffie-Hellman assumption (SAXDH)**: Let g and $\mathcal{P}$ be arbitrary generators of groups $\mathcal{G}_1$ and $\mathcal{G}_2$ resp. Let the order of g be p, and that of $\mathcal{P}$ be q. Let $m = \max(p, q)$. The **SAXDH** statement requires that the probability that any efficient algorithm can distinguish between the following two distributions is negligible, where both distributions are on $\mathcal{G}_1^4 \times \mathcal{G}_2^4$: the first distribution is obtained by choosing x, s_1, s_2, i and j randomly and independently from Z_m, and generating the tuple

$$\langle g^{xs_1}, g^{s_1}, g^x, g^{s_2}, \mathcal{P}^{i/s_1}, \mathcal{P}^{i/s_2}, \mathcal{P}^j, \mathcal{P}^{j/s_2} \rangle,$$

and the second distribution is generated identically except that the first element g^{xs_1} is replaced by g^y for a random y. The (symmetric) SAXDH assumption also requires that a similar assumption hold when the roles of $\mathcal{G}_1$ and $\mathcal{G}_2$ (i.e. g and $\mathcal{P}$) are reversed.

It is straightforward to see that the SAXDH assumption implies the SXDH assumption.

D.2 PXT Security Theorem

Theorem 11 *Let $\mathcal{L}$ be the leakage function defined by $\mathcal{L}(\mathsf{DB}, (\mathbf{s}, \mathbf{x})) = (\mathcal{L}_{\text{oxt}}(\mathsf{DB}, (\mathbf{s}, \mathbf{x})), \mathcal{L}_T(\mathsf{DB}, \mathbf{s}))$. The 2-conjunctive query SSE scheme PXT over bilinear groups $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_T$ is $\mathcal{L}$-semantically-secure against non-adaptive attacks, assuming that the SAXDH assumption holds in $\mathcal{G}_1$, $\mathcal{G}_2$ and*

$\mathcal{G}_T$, that F and F_p are secure PRFs, P is a secure PRP, that $(\mathsf{Enc}, \mathsf{Dec})$ *is an IND-CPA secure symmetric encryption scheme, and that Σ is an $\mathcal{L}_T$-secure T-set implementation.*

The proof for this theorem will be given in the full version of the paper. However, it is instructive to note some key differences from the security proof for OXT. The proof of Theorem 8 (Appendix B) describes how the table H can be populated with $g^{f_X(w)\cdot f_I(\pi(\mathsf{ind}))}$ values, and from this the xtoken can be computed in reverse as $\mathsf{xtoken}[c] \leftarrow H[\mathsf{rind}_c, w_2]^{1/y_c}$.

However, in PXT the values in the XSet are in the target group $\mathcal{G}_T$, i.e. computed using the pairing, and hence there is no way to go back to get xtoken. Thus, the Simulation is more difficult here, ad we describe the simulator in detail next.

D.3 The Simulator for SSE protocol PXT

Let $\mathcal{S}_T$ be the simulator for the $\mathcal{L}_T$-secure T-set implementation. The SSE ideal game provides the simulator $\mathcal{S}_{\mathrm{PXT}}$ with $\mathcal{L}(\mathsf{DB}, (\mathbf{s}, \mathbf{x})) = (\mathcal{L}_{\mathrm{oxt}}(\mathsf{DB}, (\mathbf{s}, \mathbf{x})), \mathcal{L}_T(\mathsf{DB}, \mathbf{s}))$, which includes the total number documents d, and sum N of the number of keywords in each document.

The simulator now maintains in addition to table H, two other tables I and Z, and starts by filling all three of them with random and independent values.

It is worth pointing out here that the $\overline{\mathbf{s}}$ does not reveal if two queries have the same x-term. However, from the intersection pattern IP the following matrix M_x of size $Q \times Q$ can be obtained: $M_x[i,j] = 1$ iff $S_i \cap S_j \neq \phi$. Now note, $M_x[i,j] = 1$ iff $\mathbf{q}[i]_x = \mathbf{q}[j]_x$ and $I_\pi(\mathbf{s}[i]) \cap I_\pi(\mathbf{s}[j]) \neq \phi$. Thus, M_x reveals partial repetition pattern of x-terms conditional on IP. Similar to profile $\overline{\mathbf{s}}$, one can also obtain a vector $\overline{\mathbf{x}}$ from M_x as follows:

- Define $\mu_x(i) = j$, where j is the least query number such that $M_x[i,j] = 1$.
- Define $\overline{\mathbf{x}}[i] = z$, where there are exactly z queries j, $j \leq i$, s.t. $\mu_x(j) = j$.

Using the leakage profiles IP and RP, $\mathcal{S}_{\mathrm{PXT}}$ does the following.

1. For each j in the range of $\overline{\mathbf{s}}$, define $CCC_j = \bigcup_{i:\overline{\mathbf{s}}[i]=j}(R_i \cup S_i)$.
2. For each $i \in [1..Q]$, it generates set $\mathbf{t}'[i] = \{g^{I[r]/Z[\overline{\mathbf{s}}[i]]}\}_{r \in CCC_{\overline{\mathbf{s}}[i]}}$.
3. It generates $\mathbf{t}[i]$ to be $\mathbf{t}'[i]$ union a set of random and independent values such that the total number of entries in $\mathbf{t}[i]$ is $\mathsf{SP}[i]$ (which is same for all j s.t. $\overline{\mathbf{s}}[i] = \overline{\mathbf{s}}[j]$).
4. The simulator $\mathcal{S}_{\mathrm{PXT}}$ calls $\mathcal{S}_T$ with $\mathcal{L}_T(\mathsf{DB}, \mathbf{s})$ and $\langle \mathbf{t}[i] \rangle_{i \in [1..Q]}$, which in turn returns TSet and STags, which $\mathcal{S}_{\mathrm{PXT}}$ outputs.
5. As a simulation of the i-th xtrap, $\mathcal{S}_{\mathrm{PXT}}$ outputs $\mathsf{xtoken}[i] = \mathcal{P}^{Z[\overline{\mathbf{s}}[i]] \cdot H[\overline{\mathbf{x}}[i]]}$.
6. As a simulation of the XSet, $\mathcal{S}_{\mathrm{PXT}}$ outputs the indicator function of the union of the following sets: for each $i \in [1..Q]$, the set consists of $e(g^{I[r]/Z[\overline{\mathbf{s}}[i]]}, \mathsf{xtoken}[i])$ for each r in R_i.

In the full version of the paper we will show that under the SAXDH assumption, the view of the adversary in the SSE ideal game using $\mathcal{S}_{\mathrm{PXT}}$ is indistinguishable from the view of the adversary in the real game using PXT.

E Related Work

We discuss here the most directly relevant papers to our work [14, 13, 20, 3, 11]. For more on related work see [14, 13]. Most work on SSE has focused on single-keyword search. After several solutions with complexity linear in the number of documents, Curtmola et al. [14] present the first sublinear solution for single-keyword search whose complexity is linear in the number of matching documents (hence optimal). They also improve on previous security models, in particular by providing an adaptive security definition and solutions in this model. The SKS protocol from Figure 1 essentially follows their approach. Chase and Kamara [13] extend and generalize the security model to more complex data (e.g., graphs) and introduce the notion of associated data that allows to compose different components of the protocol. We follow their model and modular approach. In addition, [13] introduce the notion of "controlled disclosure" that models the multi-client scenario mentioned above (see also the related "virtual private storage" setting of [23, 24]). Recently [26] introduced a strong security model of universal composability for searchable encryption, but their scheme supports only single-keyword search and it runs in time linear in the database size.

Conjunctions in the SSE setting were first considered by Golle, Staddon and Waters [20]. Their solutions are linear in the number d of all documents in the database, requiring $O(d)$ communication and same number of regular exponentiations at client and server, or communication that is only linear in the number of conjunctive terms but requires $O(d)$ pairing operations at the server. These solutions apply to structured (attribute-value) data only and leak the attributes being searched. On the positive side, they avoid leakage of keyword repetitions or access patterns other than the ciphertexts of matching documents. This work was followed up by Ballard et al. [3] and Byun et al. [11] but the above restrictions (linear in d complexity and application to structured-only data) remain. It needs to be noted that while our solutions have some crucial advantages in terms of performance and generality (applying to arbitrary data), they pay a price on privacy allowing significantly more leakage than [20, 3, 11]; on the other hand, a privacy advantage of our solutions is that they don't disclose the searched attributes.

General boolean queries are supported by CryptDB [28], a work which garnered quite some media attention and represents the state-of-art from a systems perspective. Compared to our work, CryptDB protocols offer considerably less privacy - e.g., a query for a particular database column revelas the plaintext contents of the whole complete column - and provides reduced confidency due to absence of a rigorous proof. Furthermore, CryptDB's free-text search relies on a linear SSE scheme [30] as sub-system which hinders scalability. Note that all provided benchmarks in [28] were on databases which are several orders of magnitude smaller than ours and easily fit in RAM. Having small databases fitting into RAM allowed them to side-step the challenge of dealing with high-latency storage which is unavoidable for truly large databases. A related issue arises also with protocols based on the non-adaptive version of [14] which relies on a sequential linked-list implementation for the inverted index: the required inherently sequential list traversal will cause considerably latency on queries and limit scalability as soon as indices exceed the available RAM. Our implementation allows parallel list access which can hide these latency costs in overlapping network and computation costs to a mostly constant rather than linear factor.

Another line of work on searchable encryption addresses support for efficient updates of the encrypted database [31, 25], but so far only single-keyword search was considered in this setting. Our OXT protocol, and its boolean query capabilities, can be extended to support efficient database updates at the cost of increased EDB size and partial leakage that can help differentiating between old and new documents.

Searchable encryption has also been investigated in the public-key domain [9, 32, 1, 4, 10, 29] allowing parties other than the data owner to encrypt into the database, but the cost of public

key solutions makes the practical application of these techniques more limited (e.g., to a relatively small selection of tags or keywords). The case of conjunctive queries in the PK setting has been studied in [10].

F OXT Implementation and Experimental Results

This section reports the status of the OXT implementation and some results from using our prototype with three different data sets. The latency and scalability measurements included here should be viewed as providing empirical proof to the performance and scalability claims made earlier in the paper. The section starts with a brief description of the prototype that focuses on the TSet and XSet generation and their use in the query processing, which are the biggest challenges in implementing OXT. Next, the data sets are described followed by latency and scalability measurements. The section concludes with our plans for improving the protocol performance.

Prototype. The prototype consists of two main components, one for data set pre-processing and the other for query execution, in addition to a relatively simple client program used for testing and performance measurements. The three programs use the same cryptographic primitives, which leverage the OpenSSL 1.0.1 library. To realize the cryptographic primitives mentioned in Section 2, we use the NIST 224p elliptic curves for group operations, AES-FFX [5] for permutations and, depending on platform characteristics, AES-CMAC or HMAC for PRFs. Encryption optimized for small storage overhead is achieved with a stateful and position-based variant of AES in CTR mode. The overall C code, including test programs, measures roughly 16k lines.

The pre-processing generates the TSet and XSet data structures (or EDB) for the query component as flat files together with the meta-data describing the EDB. During initialization, the query-execution component reads the EDB meta-data, retrieves the RAM-resident structures from the flat files and, subsequently, answers queries from clients, one query at a time.

The design of the TSet and XSet data structures is driven by the scalability requirement, to data sets well beyond to RAM capacity of high-end servers or even of server clusters, and by our goal of minimizing query execution time. To achieve the latter, the design maximizes the use of the available RAM and leverages domain characteristics, such as tolerance to small false positive rates. In addition, the data structures had to be designed to properly handle the protocol-induced, strictly random disk I/O operations without introducing implementation-induced leakage.

For scalability, the TSet is realized as a disk-resident paged hash table. To maximize data retrieval per disk I/O operation, 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. Tuple blocks are grouped, based on their tags, in (hash table) pages, which occupy contiguous disk areas. The unused blocks in each page are filled with random bits to make them indistinguishable from the blocks used to store tuples. The used/unused type of a block remains unknown to the query execution component until the block's tag gets generated in a client query.

The page size is set to the stride size of the underlying RAID-5 array. Similarly, the (intermediate) ext4 file system is configured based on the RAID characteristics. The query execution component uses page-level direct I/O to prevent buffer cache pollution in the OS, as the hash table pages are inherently uncachable. In addition, this component parallelizes disk accesses using asynchronous I/O (`aio_*` system calls). As a result, each page access is handled by a single hard disk drive and each drive has several outstanding page requests, which achieves the maximum possible parallelism and I/O depth in the storage system. Note that due to the inherent (and intended) random nature of the TSet accesses, neither the underlying file system nor the storage subsystem can leverage standard acceleration techniques, such as pre-fetching, to improve system performance.

The much smaller XSet is realized as a RAM-resident Bloom filter [6], which enables the sizing the false positive rate to the lowest value that the server's RAM allows. In the the current prototype, we set the false positive rate to 2^{-20} and the resulting Bloom filter XSet still occupies only a small fraction of our server RAM. During query execution, disk accesses are overlapped with the xtag generation and Bloom filter query operations, which are both parallelized across all the available CPU cores.

Data Sets. To show the practical viability of our solution we run tests on three data sets: a 100,000 record (attribute-value) database synthesized from census data, where each record is treated as a separate document d, attribute values are atomic, and where each pair $(attribute_i, value)$ is treated as W_i, in an arbitrary ordering of attribute names; the Enron email data set [15] with more than 1.5 million documents (email messages and attachments) where all words, including attachments and envelope information, have been indexed resulting in about 1.2 million distinct words; and the ClueWeb09 [27] collection of crawled web-pages from which we extracted several databases of increasing size where the largest one was based on 0.4TB of HTML files with almost 3 billion of per-document-distinct words. 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 medium-size sets as the Enron case.

The pre-processing component accesses the three data sets as MySQL databases. The census data set is realized as a 100,000 record table, with one column for each attribute name. For the other two data sets, we wrote a set of Perl scripts to clean, stem, and convert the Enron emails and their attachments as well as the ClueWeb09 HTML files into separate MySQL tables. This transformation allow us to leverage the same pre-processing component for all data sets and it sets the foundation for handling more elaborate data sets such as XML document collections.

For the largest database derived from the ClueWeb09 data set based on 0.4 TB of HTML files, the sizes of the TSet hash table and XSet Bloom filter are 144.4 GB and 9.7 GB, respectively. The corresponding sizes for Enron and the census data are 12.4 GB and 0.6 GB, and 252 MB and 5 MB, respectively. For the multi-client configuration, in which each tuple includes the record decryption key encrypted with a key derived from the s-trap, the TSet size increases by 47%.

Experimental Results. The generation and retrieval of the DC (document cipher text) are straightforward and hence the measurements reported here cover only the protocols up to the discovery and decryption by the client of the rinds of the result set.

Figure 11 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 varies from 3 to 690,492 documents. As this query consists only of an s-term, the figure illustrates that its execution time is linear in the cardinality of the corresponding TSet. 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 OXTprotocol can be overlapped with disk and network I/O. It illustrates the for many surprising fact that even public-key cryptography, considered by many as expensive, can be surprisingly 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

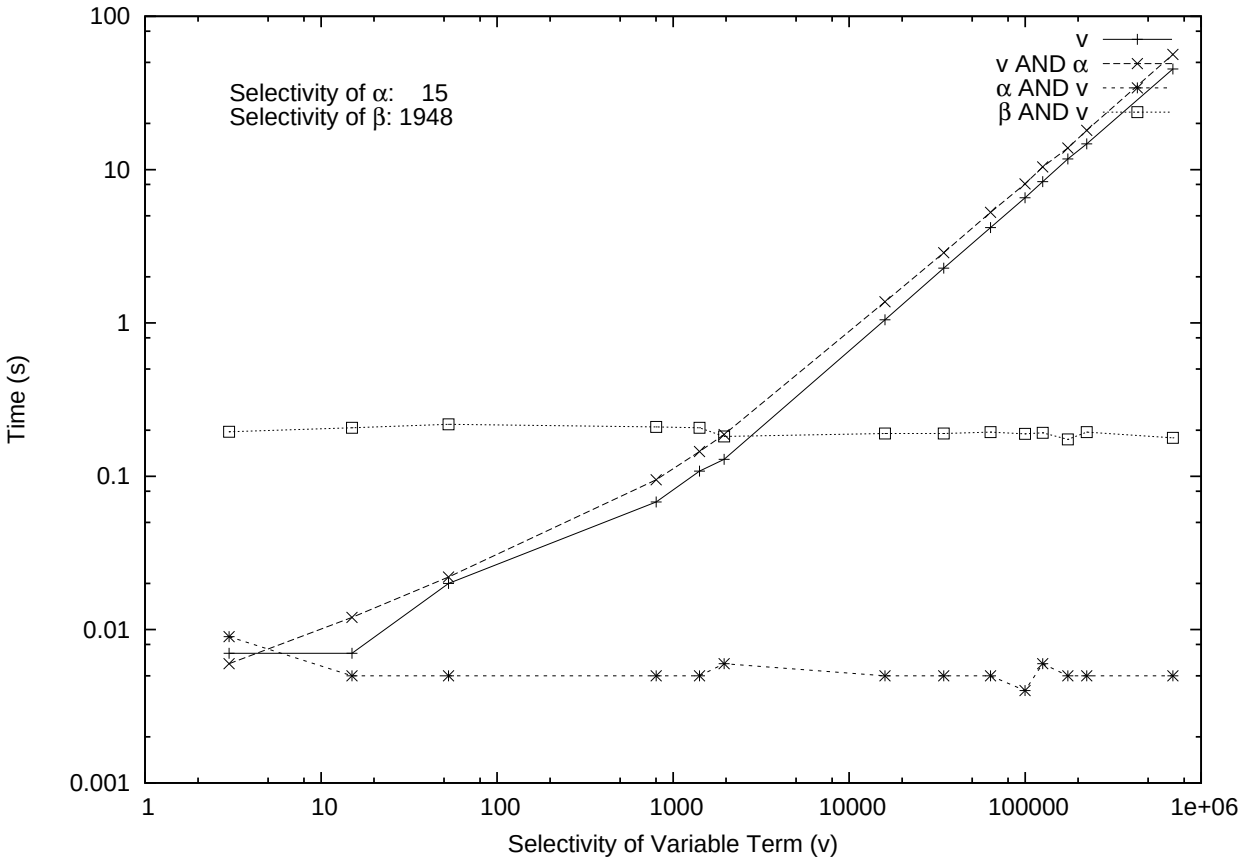

Figure 11: Enron Performance Measurement: Single Term & Conjunction

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 3.1.1. Experiments on the census data show similar characteristics. All experiments were run on IBM Blades HS22 attached to a commodity SAN system.

To further assess the scalability of our query engine, 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 $2,689,262,336$ per-document-distinct words. To make these databases comparable, we injected some artificial and non-conflicting query terms to randomly selected documents simulating words of various selectivity. We then queried these terms in the various databases. Figure 12 shows the results and 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.

To quantify the 'performance cost' of the query and data privacy that our system provides, we measured the execution times of the same queries when run by a MySQL server (release 5.5) using its full text search capability on the Enron data set. To get a fair comparison, we measure execution times using a "cold" MySQL server, where we restarted the server and flushed the file-system caches before each query. In this scenario, our implementation is very competitive: on

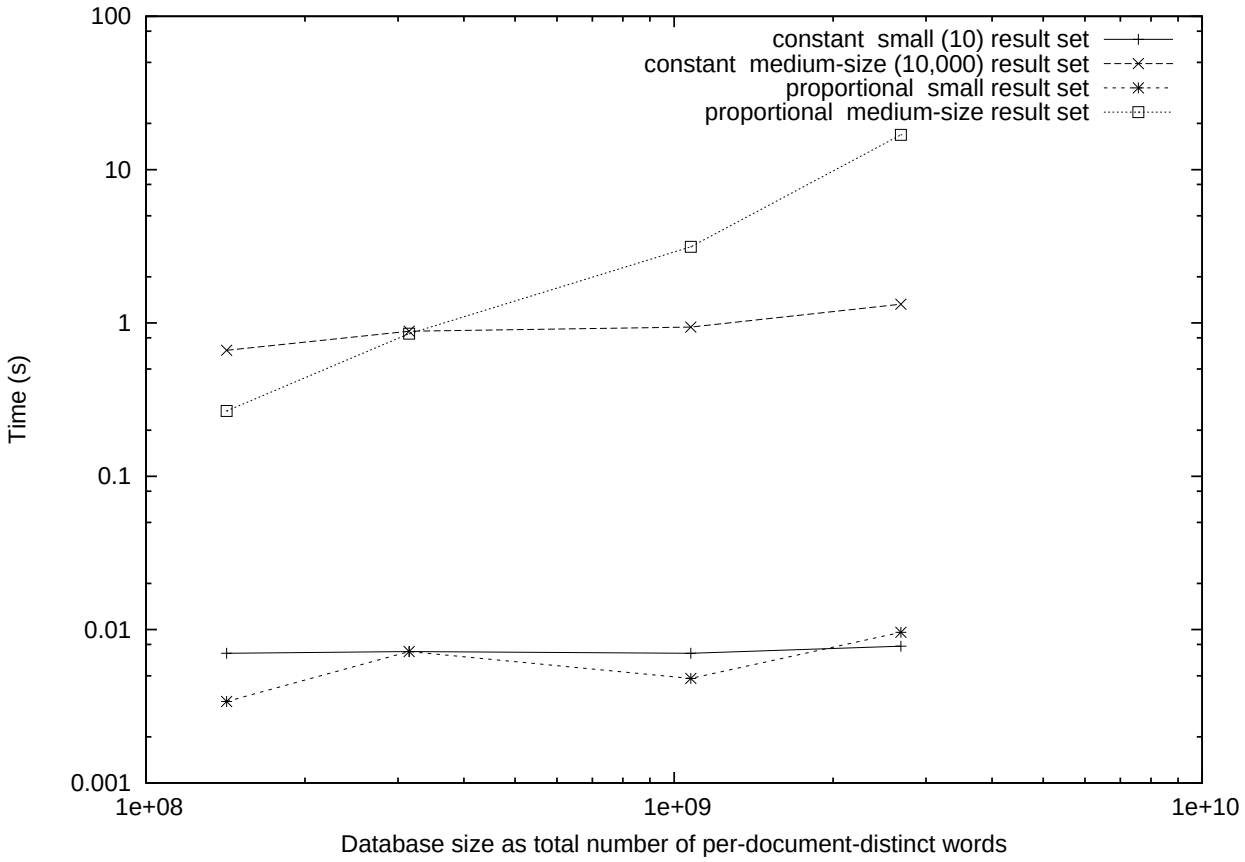

Figure 12: Clueweb09 Performance Measurement: Scaling Database Size

single term queries it is faster, sometimes significantly, for queries with small result sets and only slower (by a factor of 2) for terms with very poor selectivity, such as the very commom keyword 'enron'. For two-term conjunctions, the OXT prototype is slower than MySQL when the s-term is badly chosen. However, a good s-term selection results in consistently better performance than (cold) MySQL, in some cases by an order of magnitude.

As the Enron data set is much smaller than our server's RAM, a significant fraction of the data set and associated indexes get pre-fetched and cached in the MySQL and file system buffers after the first query. Not surprisingly, MySQL with warm caches is faster by one order of magnitude or more on most queries. We expect an alternative OXT prototype optimized for small data sets, namely using a RAM-resident TSet implementation, will be competitive with MySQL in warm cache scenarios. Finally, we need to emphasize that our prototype supports only a small subset of the query capabilities of a commercial database server, such as MySQL.

As our future work aims for even larger databases, we plan to investigate alternative strategies for realizing efficient disk-resident XSets.