# Low-Leakage Secure Search for Boolean Expressions

Fernando Krell[1], Gabriela Ciocarlie[2], Ashish Gehani[2], and Mariana Raykova[3]

[1] Dreamlab Technologies[⋆⋆⋆] fernando.krell@dreamlab.net
[2] SRI International {gabriela,gehani}@csl.sri.com
[3] Yale University[†] mariana.raykova@yale.edu

**Abstract.** Schemes for encrypted search face inherent trade-offs between efficiency and privacy guarantees. Whereas search in plaintext can leverage efficient structures to achieve sublinear query time in the data size, similar performance is harder to achieve for secure search. Oblivious RAM (ORAM) techniques can provide the desired efficiency for simple look-ups, but do not address the needs of complex search protocols. Several recent works achieve efficiency at the price of revealing the access pattern. We propose a new encrypted search scheme that reduces the leakage of current Boolean queries solutions, while introducing limited overhead and preserving the sublinear efficiency properties for the search protocol in the semi-honest model. Our scheme achieves a privacy-efficiency trade-off that lies between highly optimized systems such as Blind Seer [18] and OXT-OSPIR [15], which exhibit significant access pattern leakage, and the secure search solution of Gentry et al. [8], which has no leakage, but a much higher efficiency cost.
Our solution is based on a hybrid approach, which integrates ORAM techniques with the efficient search index structure of the Blind Seer system. We reduce the leakage to the server to only the number of nodes visited in the search tree during query execution. Queries that execute in sublinear time in Blind Seer execute also in sublinear time in our scheme. To enable delegated queries, we develop a new protocol for oblivious PRF sum evaluation and perform secure Boolean queries in a Bloom filter that reveals only the match result. We also enable oblivious-search token generation to hide the specifics of the delegated query from the data owner issuing the search tokens.
We evaluated our system by implementing a prototype and testing it on a 100,000-record database. Our results indicate that the index can be traversed at a rate of a few seconds per matching record for both conjunction and small Disjunctive Normal Form queries.
**Keywords:** Private Search, Boolean Queries, Bloom Filters, ORAM

## 1 Introduction

The ability to search over encrypted data provides critical capabilities for database systems that need to guarantee privacy protection for data and queries; exam-

[⋆⋆⋆] Work described here was carried out while this author was at SRI International and partially at Columbia University.
[†] Work described here was mainly carried out while this author was at SRI International.

ples include information sharing between law enforcement agencies; electronic discovery in private databases such as log files, bank records, during lawsuits, and private queries to census data; police investigations using data from automated license plate readers [4, 18]. Such functionality enables data outsourcing, where a client stores its data on a remote server and later sends queries that the server executes without learning them. An extension to this functionality is the setting of delegated search, where the data owner can generate query tokens for third parties that enable them to execute only the authorized query requests with the storage server without learning any other information about the outsourced data. An immediate application of this extension is the ability to audit cloud applications and allow auditors only issue to authorized queries.

However, there are two main relevant questions of privacy and efficiency for encrypted search schemes, whose answers are also related. The privacy question considers how much the storage server learns about the queries it executes. While encryption techniques can help hide the content stored on the server, they do not protect the client's access pattern which becomes a privacy leakage to the server. A recent work [14] has demonstrated that this leakage can be substantial even in the simplest search scenario of exact match such as keyword search. For more complex types of queries, such as Boolean queries, this leakage can have even more serious security implications. The second question is related to efficiency. Search algorithms in plaintext usually have sublinear efficiency in the size of the database, and this efficiency guarantee is crucial for the usability of an algorithm. The sublinear efficiency of a search algorithm implies that it does not access all data. At the same time, the ability of the server to know what data has been accessed translates into access pattern leakage. Thus, an inherent trade-off between the privacy and efficiency questions for secure search emerges.

There have been several approaches to secure search in previous works that achieve various trade-offs between the efficiency and the privacy guarantees of their schemes. Searchable encryption [22, 3, 6] provides the capability to encrypt data items such as keywords, issue search tokens for particular items using the private parameters for the scheme, and support matching functionality to check whether a ciphertext contains the same item as a search token. Using this primitive, one can implement a search protocol with sublinear efficiency, which completely reveals the access pattern into the database for each query. Private information retrieval (PIR) [5] and symmetric PIR [9] techniques allow a client to retrieve a record stored on a server without revealing what record is being retrieved or allowing the client to learn anything more about the data. Such techniques can be employed for secure search, but they require computation proportional to the size of the database.

While earlier work on secure search considered mostly single keyword search queries [22, 3, 6, 21, 19, 17], recent approaches address more complex queries over databases, such as Boolean and range queries, and model queries over databases of records, which are described by several attributes and a main payload [4, 15,

18, 7, 13][4]. These last solutions provide surprisingly good efficiency at the price of access pattern leakage. Unfortunately, the access pattern is not uniquely defined. In fact, each of these solutions reveals an access pattern that is specific to the structure of the scheme's data storage and goes beyond the records that match each query. Hence, it is difficult to analyze this leakage and to precisely define what is protected, making it impossible to compare the leakage of different schemes that employ different underlying encrypted search structures.
A different line of work [20] proposes a solution for encrypted search that can handle SQL queries. However, it considers a different adversarial model that aims to protect the data against curious database administrators, but assumes fully trusted proxy that encrypts the queries. This model does not match the guarantees that we want to achieve. This solution also reveals the access patterns for the query terms.
A completely different approach for the secure search problem is to employ secure computation techniques to implement the search functionality. While generic secure computation techniques require computation time at least linear in the size of its inputs, the works of Gordon et al. [12] and Afshar et al. [1] manage to achieve sublinear (amortized) time for sublinear RAM computations in the semi-honest and malicious setting respectively. These approaches leverage the random access machine computation (RAM) model together with a special structure for memory storage called oblivious RAM (ORAM) [11], which provides access patterns hiding with only polylogarithmic overhead for memory accesses. This approach was further pursued in the work of Gentry et al. [8] focusing on the database query functionality and employing somewhat-homomorphic encryption to implement the small secure computation steps. This work handles keyword database queries and limited conjunction queries.

Our work is motivated by the lack of a good grasp on analyzing leakage in the Boolean search protocols mentioned earlier. We propose a construction that adopts the approach of combining ORAM together with small secure computation steps. We focus on functionality for Boolean search queries and we develop tailored solutions for that. Our solution for secure search enables the same functionality for Boolean queries as Blind Seer [18, 7], but it diminishes the access pattern leakage, while preserving the sublinear efficiency overhead for queries that are executed in sublinear time in these protocols. As expected, when compared with these solutions that reveal complete access patterns, the concrete efficiency overhead for our protocol increases. Although the direct comparison with the work of Gentry et al. [8] is hard, since their work implements much simpler queries compared to our protocols, our protocols achieve a much better efficiency for comparable functionality.

### 1.1 Setting

The setting we are interested in is called Outsourced Symmetric Private Information Retrieval (OSPIR) [15]. It captures the scenario in which the data owner

[4] Interesting is the single-keyword range-query solution of [13] which provide a tunable privacy-efficiency trade-off

outsources the data to a server, and gives search capabilities to clients. Such a scheme can be defined by two phases $\mathsf{OSPIRSetup}$ and $\mathsf{OSPIRSearch}$.

In the $\mathsf{OSPIRSetup}$ phase, the data owner ($\mathsf{Owner}$) on input $\mathsf{DB}$, does some preliminary computation on the data and produces an *encrypted* database $\mathsf{EDB}$ and access parameters $\mathsf{params}$. $\mathsf{EDB}$ is then given to the server ($\mathsf{Server}$). In the $\mathsf{OSPIRSearch}$ phase, a client ($\mathsf{Client}$) inputs a query $\mathbf{q}$, $\mathsf{Owner}$ inputs $\mathsf{params}$, and $\mathsf{Server}$ inputs $\mathsf{EDB}$. After protocol execution, $\mathsf{Client}$ obtains database records satisfying its query. We provide a formal definition next, allowing a tunable false positive rate on the records returned to $\mathsf{Client}$.

**Definition 1.** *We define an* OSPIR Scheme *as a pair of interactive algorithms*

- $(\mathsf{params}, \mathsf{EDB}) \leftarrow \mathsf{OSPIRSetup}(1^\lambda, \mathsf{DB}, \mathsf{fp})$. $\mathsf{Owner}$ *inputs database* $\mathsf{DB} = \{(D_i, W_i)\}_{i=1}^{D}$, *and gets back* $\mathsf{params}$. $\mathsf{Server}$ *gets* $\mathsf{EDB}$.
- $(\mathsf{records}) \leftarrow \mathsf{OSPIRSearch}(\mathsf{params}, \mathsf{EDB}, \mathbf{q})$. $\mathsf{Owner}$ *inputs* $\mathsf{params}$, $\mathsf{Server}$ *inputs* $\mathsf{EDB}$ *and* $\mathsf{Client}$ *inputs* $\mathbf{q}$. $\mathsf{Client}$ *gets* $\mathsf{records}$.

*such that for all* $\lambda$ *and for all* $\mathsf{DB}$, $\mathbf{q}$, *if* $(\mathsf{params}, \mathsf{EDB}) \leftarrow \mathsf{OSPIRSetup}(1^\lambda, \mathsf{DB}, \mathsf{fp})$, *and* $(\mathsf{records}) \leftarrow \mathsf{OSPIRSearch}(\mathsf{params}, \mathsf{EDB}, \mathbf{q})$, *then* $\mathsf{DB}_{\mathsf{fp}}(q) = \mathsf{records}$, *where* $\mathsf{DB}_{\mathsf{fp}}(q)$ *denotes the records of* $\mathsf{DB}$ *satisfying the query* $\mathbf{q}$ *plus each* $\mathsf{DB}$ *record with probability* $\mathsf{fp}$.

The OSPIR scheme described in this work assumes a semi-honest behavior of the participants. That is, we assume that every participant honestly follows the description of the protocol, and we define and prove security in such setting.

## 1.2 Related Work

As mentioned earlier, the problem of privately searching a database can be solved by generic secure computation schemes [23, 10]. However, these generic protocols require a computation time that is at least linear in the size of the participant's inputs. A scenario closer to out setting is the one of PIR and SPIR protocols, where a client obliviously selects an item from the server's database. Although PIR-like protocols provide sublinear communication, they do require linear-time computation. Ad hoc solutions [12, 8] provide sublinear computation time for look-ups and single keyword search in the private DB setting.

A more practical approach is taken in the searchable encryption schemes [22, 3, 6, 4] and OSPIR protocols [15, 18, 7]. These schemes achieve an efficiency close to plaintext solutions, but at the cost of revealing access patterns to the database records and the underlying search structure. The OSPIR solution of [13] provides a tunable efficiency-privacy trade-off solution. They achieve efficiency comparable to [15, 18, 7] with virtually no access pattern leakage for a tunable number of queries. After this threshold is reached, the index need to be rebuild to avoid incurring access pattern leakage.

Among the works just mentioned, the HE-over-ORAM approach [8], and the Blind Seer and OXT-OSPIR[15] are of particular interest. First, these schemes focus on the delegated query scenario. Secondly, while HE-over-ORAM aims for a secure asymptotically sublinear solution for single keyword search, the Blind

Seer and OXT-OSPIR systems focus on practicality: they both support a rich set of queries and their efficiency is close to the plaintext database case. Our goal is to build a system that lies in-between these systems in terms of the privacy vs. efficiency trade-off. Hence, we borrow techniques from all these solutions.

**OSPIR-OXT and Blind Seer.** The first solution for the OSPIR setting was proposed by Jarecki et al. [15] and Pappas et al. [18]. Although they solve the same problem, they provide very different approaches. OSPIR-OXT [15] is an extension of the OXT searchable encryption scheme [4]. This solution allows for Boolean queries in Searchable Normal Form $(t_1 \wedge \phi(t_2, ..., t_n))$, and runs in time proportional to the number of records satisfying the term $t_1$. The solution is based on an inverted index approach, which is used to search information about the leading term $t_1$. This information is used then to search for the records satisfying the sub-queries $t_1 \wedge t_i$. A completely different approach was taken in Blind Seer [18, 7]. Instead of using an inverted index, Blind Seer builds a Bloom filter tree on the searchable keywords of the database. Each leaf of the tree is associated with a record in the database, and each internal node corresponds to a *masked* Bloom filter containing the searchable keywords of the records in its subtree. Hence, a Boolean formula is answered by following root-to-leaves paths, where the nodes' Bloom filter satisfy the query. To evaluate each node, the server and the client engage in a secure two-party computation protocol (implemented using Yao's protocol [23]), where the server inputs masked Bloom filter bits, and the client inputs the mask. The big advantage of OSPIR-OXT over Blind Seer is efficiency. This is due to the interactive nature of Blind Seer. In terms of leakage, these systems are incomparable since their underlying data structures are completely different. Blind Seer, though, has the advantage that the search procedure does not reveal the partial evaluation results. In addition, Blind Seer can answer *any* Boolean query in sublinear time.

**HE-over-ORAM Database Search.** Gentry et al. [8] recently proposed a private DB system with no leakage based on ORAM and Somewhat Homomorphic Encryption Scheme. ORAM is used to protect the client's access patterns and the owner's data from the server. To protect the database information from the client, data is also encrypted using a variation of a Somewhat Homomorphic Encryption Scheme that enables Equal-to-Zero and Comparison operations. These operations enable the client to blindly perform ORAM operations until the requested value is found. Although this work shows the feasibility of the HE-over-ORAM approach, it has significant limitations in efficiency and functionality. In terms of efficiency, their experimental results shows that it requires 30 minutes to execute a single keyword query on a $2^{22}$ record database. In terms of functionality, the system only allows single keyword queries, and conjunction may be enabled by a trivial addition of the keywords into the database index.

### 1.3 Approach

Our approach is to use the Bloom filter Search Tree of Blind Seer as our search structure, while storing the encrypted data and its index in ORAM structures at the server. We give the ORAM access parameters to the client, as done in the HE-over-ORAM scheme. To avoid the case where the client learns more information than necessary, the actual data held by the ORAM should be encrypted in a special way. While this is done using Somewhat Homomorphic Encryption by Gentry et al. [8], we provide a new encoding scheme that allows parties to securely evaluate an index node, revealing to the client only the necessary information to continue the search procedure. We accomplish this with a novel protocol for conjunctive query evaluation on specially encrypted Bloom filters. This protocol is then extended to handle queries in Disjunctive Normal Form.

The use of ORAM eliminates all important leakage to the index server of Blind Seer [18]. ORAM protocols, however, do leak the number of queries performed by the client; hence, our solution reveals the amount of work done by the client (which is unavoidable if we require sublinear time). In particular, the server can infer the number of records retrieved by the client. It also learns the relation between the amount of work in the index and the amount of records retrieved. Nevertheless, the server is unable to link the work done in the index and the specific record retrieved.

## 2 Preliminaries

**Bloom Filter.** A Bloom filter [2] is a data structure that allows for set membership queries. The structure is composed by a bit array $B[1..n]$ and a set of hash functions $\mathcal{H} = \{H^{(i)} : \{0,1\}^* \to [n]\}_{i=1}^{h}$. An element is inserted by turning on the bits at the positions indicated by the hash values of the element. Hence, if an element $e$ is in the filter, then $B[H^{(i)}(e)] = 1$ for all $i \in \{1..h\}$. A Bloom filter is parametrized by a *false positive* rate since elements not in the set may hash to positions that are all set in $B$. Given a false positive probability $\mathsf{fp}$ and the number of elements $N$ in the filter the optimal length $n$ of $B$ and the optimal number of hash functions $h$ to use can be approximately computed as $n = \lceil \frac{N \ln \mathsf{fp}}{\ln \frac{1}{2^{\ln 2}}} \rceil$, $h = \lceil \ln 2 \frac{n}{N} \rfloor \approx \log_2(1/\mathsf{fp})$.

**Bloom Filter Search Tree.** A Bloom Filter Search Tree ($\mathsf{BFT}$) is an index for a database $(D_i, W_i)_{i=1}^{D}$, where $D_i$ is an arbitrary document and $W_i$ is $D_i$'s associated set of searchable keywords. Given a parameter $\mathsf{fp}$ (false positive), a Bloom filter tree is constructed by building a $b$-ary tree of $D$ leaves. Each leaf of this tree is associated with a document $D_i$ and holds a Bloom filter with the corresponding set of keywords $W_i$. Each internal node contains a Bloom filter having inserted all keywords held at its children nodes. A search procedure for documents containing a keyword $w$ starts by querying the Bloom filter at the root node. If the keyword is present, we continue recursively querying its children until reaching the leaf nodes whose associated document contains $w$.

**Oblivious RAM.** An Oblivious RAM protocol [11] is a two-party protocol that allows a client to outsource its data and completely hide the access pattern of future queries. It is composed by an algorithm $\mathsf{OSetup}$ and by a protocol $\mathsf{OAccess}$. $\mathsf{OSetup}$ is run by the client (or data owner) and outputs parameters $\mathsf{param}$ (including an initial state $\mathsf{state}$), and data structure $\mathsf{struct}$. $\mathsf{OAccess}$ is a two-party protocol in which the client inputs an operation $\mathsf{op} \in \{\mathsf{ORead}, \mathsf{OWrite}\}$, an index $i$, data $D^\star$ (if $\mathsf{op}$ is $\mathsf{OWrite}$), and parameters $\mathsf{param}$. The server inputs $\mathsf{struct}$. At the end of the protocol execution, the server obtains an updated structure,

$\mathsf{struct}'$, while the client obtains the updated state $\mathsf{state}'$, and data $D_i$ (if $\mathsf{op}$ was $\mathsf{ORead}$). It is well known that any ORAM holding $n$ elements and simulating $m$ RAM accesses requires $\Omega(m \log n)$ accesses [11].

**Participants.** The system supports three actors: Owner, Server, and Client. The Owner knows the database $(D_i, W_i)_{i=1}^{D}$, builds an index and outsources the list of documents $(D_i)$ and the index to the Server. The Client has a *query* $\mathbf{q} = \phi(W)$ composed by a Boolean formula $\phi(\cdot)$ over a set of keywords $W$. The Client gets the set of documents $\{D_i : W_i \text{ satisfies } \phi(W)\}$.

**Notation.** We use $\lambda$ to denote a security parameter, and $\mathsf{fp}$ a false positive rate. The set $\{1, 2, ..., i\}$ will be denoted as $[i]$. Let $\mathcal{G}$ be a group of generator $g$ and prime order $p$, where the Decisional Diffie-Hellman (DDH) assumption holds. We use multiplicative notation for the group operations. Let $H : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a keyed hash function (or MAC) having keys in $\{0,1\}^\lambda$, in which $H(k,w)$ is denoted as $H_k(w)$. Similarly, let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathcal{G}$ be a pseudo random function (PRF) indexed by keys in $\{0,1\}^\lambda$, having domain in $\{0,1\}^\lambda$, and image in $\mathcal{G}$. We denote $F(k,r)$ as $F_k(r)$. Let $\mathcal{E} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec} \rangle$ be a semantically secure encryption scheme. For a query $\mathbf{q}$ corresponding to a DNF formula $\phi(\cdot)$, we let $|\mathbf{q}| = |\phi|$ be the number of conjunctive clauses in $\phi$. For a clause $C_i \in \phi$, we let $|C_i|$ be the number of terms in $C$. The *topology* of $\mathbf{q}$, denoted as $\mathsf{topo}(\mathbf{q})$ (or $\mathsf{topo}(\phi)$), correspond to $|\mathbf{q}|$ and $|C_i|$ for each $i \in [|\mathbf{q}|]$ We denote by $x \overset{\$}{\leftarrow} S$ the process of sampling a uniformly random element $x$ from set $S$. For a tree node $v$, we let $\mathsf{Children}(v)$ be the set of children nodes of $v$. We let $\mathsf{BFBuild}(S, \mathsf{fp})$ denote the process of building a Bloom filter for set $S$ with false positive rate $\mathsf{fp}$, and $\mathsf{BFMatch}(\mathsf{BF}, w)$ denotes the process of matching a keyword $w$ in Bloom filter $\mathsf{BF}$. For a set of hash functions $\mathcal{H}$, we let $\mathcal{H}(w)$ denote the set $\{H(w) : H \in \mathcal{H}\}$. Finally, we abuse ORAM notation and let $D_i \leftarrow \mathsf{ORead}(i, \mathsf{struct})$ denote a read ORAM access on address $i$ at ORAM structure $\mathsf{struct}$ held by the server. That is, we omit in the notation the client's parameters and the updated structure given to the server.

## 3 Cryptographic Primitives

In this section, we introduce the necessary cryptographic primitives for the construction of our private search scheme (Section 4). These primitives are presented in a modular way, and can be of independent interest.

**Oblivious PRF.** First, our solution uses an Oblivious Pseudorandom Function (OPRF). It involves two parties, $\mathcal{C}$ having input $m$ and $\mathcal{S}$ having input $k$, who jointly evaluate a pseudorandom function $F_k(m)$, keeping $k, m$ private to the respective party. A simple construction proposed by Jarecki and Liu [16] uses the Hashed Diffie-Hellman PRF ($F_k(m) = \mathsf{Hash}(m)^k$). The protocol is described in Figure 1. $\mathcal{C}$ starts by sampling a uniformly random invertible exponent $\alpha$ and sends $X = \mathsf{Hash}(m)^\alpha$ to $\mathcal{S}$. $\mathcal{S}$ responds with $Y = X^k$. Finally, $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}} = \mathsf{Hash}(m)^k$.

**MUL-OPRF.** In a simple variation of the above primitive, $C$ inputs a set $\{m_1, ..., m_n\}$, $S$ inputs the secret key $k$, and $\mathcal{C}$ receives as output $\prod_i F_k(m_i)$. We call this new primitive MUL-OPRF. We obtain a secure protocol for this primitive by using $\prod_i \mathsf{Hash}(m_i)$, as the random hash function in the protocol of Figure 1.

**Masked MOPRF.** For the purpose of the construction in Section 4, we require a slight modification on the above MUL-OPRF functionality. We call this new primitive a Masked MOPRF. In this primitive, $\mathcal{C}$ gets the result of the MUL-OPRF protocol masked with a random value $R$, while $\mathcal{S}$ obtains the mask $R$. This simple modification is achieved by adding one extra message in the protocol (Figure 2). The server starts by sampling a uniformly random exponent $\beta$, and sending $W = g^{\beta}$ back to $\mathcal{C}$. $\mathcal{C}$ responds with $X = (W \cdot \prod_i \mathsf{Hash}(m_i))^{\alpha}$ for the uniformly random invertible exponent $\alpha$. $\mathcal{S}$ replies with $Y = X^k$, and outputs $R = g^{\beta \cdot k}$. $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}} = R \cdot \prod_i \mathsf{Hash}(m_i)^k$.

**Two-party Protocol** OPRF

**Parameters.** A random hash function $\mathsf{Hash} : \{0,1\}^* \to \mathcal{G}$.
**Inputs.** $\mathcal{C}$: $w \in \{0,1\}^*$. $\mathcal{S}$: $k$.

1. $\mathcal{C}$ samples $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$, and sends back $X = \mathsf{Hash}(w)^{\alpha}$.
2. $\mathcal{S}$ replies with $Y = X^k$.
3. $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}}$.

Fig. 1: The Two-Party Protocol OPRF

**Two-party Protocol** Masked-MOPRF

**Parameter:** A random hash function $\mathsf{Hash} : \{0,1\}^* \to \mathcal{G}$.
**Input** $\mathcal{C}$: $\{m_i\}_{i \in [n]}$. $\mathcal{S}$: $k$.

1. $\mathcal{S}$ samples $\beta \xleftarrow{\$} \mathbb{Z}_p$ and sends $W = g^{\beta}$.
2. $\mathcal{C}$ samples $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$, and sends back $X = (W \cdot \prod_{i=1}^{n} (\mathsf{Hash}(m_i))^{\alpha}$.
3. $\mathcal{S}$ replies with $Y = X^k$.
4. $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}}$.
5. $\mathcal{S}$ outputs $R = g^{\beta \cdot k}$.

Fig. 2: The Two-Party Protocol Masked-MOPRF

**Security.** The security of the MUL-OPRF protocol follows directly form the security of the Hashed DH Oblivious PRF protocol [16] by using $\prod \mathsf{Hash}(\cdot)$ as the random function in the random oracle model. The security and correctness of the Masked MUL-OPRF protocol follows directly from DDH assumption since it implies that the value $g^{\beta \cdot k} \times \prod_i F_k(m_i)$ is pseudo-random even given $g^{\beta}$ (and even if the adversary somehow knows $\prod_i F_k(m_i)$).

## 4 Scheme

In this section, we present our private search scheme. Our ultimate goal is a secure search functionality that enables oblivious delegated queries on outsourced data to a server (Server), where the data owner (Owner) can obliviously issue a search token to a client (Client) for a query that remains hidden from the owner.

Given this search token, Client should only learn the data matching the query, while minimizing the information that the server (Server) learns about the issued queries (we analyze what Server learns formally in the full version of this work).

Recall from Section 1.3 that our search structure is a Bloom filter tree in which documents are associated with the leaves of the tree and each node contains a Bloom filter holding the searchable keywords of the documents associated with the leaves of its subtree. In the simple two-party setting, where Owner is the querier (or client), Owner can build a plaintext Bloom filter tree storing it as an ORAM at the server. Then, for each query, Owner traverses the Bloom filter tree (via ORAM accesses), to find the documents that satisfy its query (which it retrieves and decrypts also via ORAM accesses).

In the delegated queries scenario (i.e., where Client is not the database owner), if complete ORAM access is allowed to Client, information beyond what is strictly necessary is revealed. First, since each ORAM access may retrieve several elements, Client gets bits of the index that do not correspond to its query. Equally important, Client learns partial evaluation information, such as which keywords of the formula are satisfied at each node, and which Bloom filter bits are set. Ideally, Client should only learn if the complete query is satisfied by the index node being evaluated. These two problems are addressed by *specially* encrypting the index bits and introducing an oblivious protocol that allows Client to only learn whether the formula is satisfied by an index node, but nothing more.

In Section 4.1, we introduce techniques that allow for secure delegated queries leveraging Bloom filter tree and ORAM approaches. We first show how to generate query tokens without revealing the client's query to either party. We then describe how to securely evaluate single term queries, conjunctions and DNF queries on each Bloom filter, allowing Client to traverse the tree and find the documents satisfying its query. Finally, we describe how Client can decrypt the retrieved documents without any party knowing the identifiers of these documents. In Section 4.2, we present the complete construction of our private search scheme.

## 4.1 Building Block Techniques

**Obliviously Generating Search Tokens.** Before Client can evaluate its query, it needs to be able to compute Bloom filter indices corresponding to the terms in the query for each Bloom filter in the tree. These indices need to be derived from a PRF, whose key, $s_{\mathsf{bf}}$, is held by the database owner. For this purpose, each term is mapped through the use of this PRF to a *search token*, which is then hashed to get the Bloom filter indices. Similarly to Jarecki et al. [15], we use the Hashed Diffie-Hellman PRF $F_{s_{\mathsf{bf}}}(w) = \mathsf{Hash}(w)^{s_{\mathsf{bf}}}$ as our PRF to compute search tokens for each term. This PRF can be obliviously computed via the protocol in Figure 1.

**Single Term Queries.** For single keyword queries, $\mathbf{q} = \phi(w) = w$, the client needs to learn if all the bits queried in a Bloom filter are set. For this purpose, we leverage the Masked-MOPRF protocol making use of the underlying PRF to encrypt each bit. We encode a bit to an arbitrary element in the range group of the PRF $F$, and use $F$ to encrypt the bit. Let $g$ be a group generator (that we keep secret from the client); we map a bit $b_i$ to $g^{b_i}$ and encrypt it as $\langle g^{b_i} \cdot F_k(r_i), r_i \rangle$ for position $i$ in the Bloom filter[5]. The client and server use the Masked-MOPRF primitive described in Figure 2 to evaluate a Bloom filter query that reveals no additional information to the client as follows. They execute the Masked-MOPRF protocol with inputs a set of $\{r_i\}_{i \in S}$, for the client, and a PRF key $k$ for the server, where $S$ is the set of BF indices corresponding to the query. At the end of the protocol, the client obtains $R \cdot \prod_{i \in S} F_k(r_i)$, while the server obtains a random value $R$. Next, the client computes $\prod_{i \in S} \left( g^{b_i} \cdot F_k(r_i) \right)$, and, using its output from the Masked-MOPRF protocol, obtains

$$\prod_{i \in S} \left( g^{b_i} \cdot F_k(r_i) \right) \left( R \cdot \prod_{i \in S} F_k(r_i) \right)^{-1} = R^{-1} \cdot g^{\sum_{i \in S} b_i}$$

The server now provides $H(R^{-1} \cdot g^h)$ so that the client can do the matching evaluating $H(R^{-1} \cdot g^{\sum_{i \in S} b_i})$ and the comparison. The random element $R$ binds together the values from all BF indices corresponding to a query, and does not allow the client to learn any information about subsets of the BF bits in the corresponding positions. The hash over the server-side matching key $R^{-1} \cdot g^h$ hides $R$ from the client. Hence, the protocol completely hides the value $\sum_{i \in S} b_i$ for a mismatching query.

**Conjunction Queries** The method described above can be trivially extended to conjunctions since the single term case is in fact a conjunction on the corresponding Bloom filter bits. We can treat a conjunction as a bigger single term query. Let $C$ be a conjunction, and let $|C|$ denote the number of terms in $C$, then the number of bits to be checked is $h \times |C|$.

**Disjunctive Normal Form Queries.** In the case of single term queries (and conjunctions), a match requires that all the bits at the query indices of the Bloom filter be set to one. Therefore, it suffices that the server provides the hash of a single "randomized matching key" $H(R \cdot g^h)$ to the client. In the case of disjunctions, on the other hand, there are many settings for the bit values of the query BF indices that can satisfy the query; hence, there are many possible matching keys. In fact, there can be as many as $|C| \cdot 2^{h \cdot (|C|-1)}$ different satisfying bit value assignments for the BF query indices. However, in our construction, we consider the expression $g^{\sum_{i \in S} b_i}$ for each term in the conjunction, which has $h$ different possible values which depend only on the number of ones in the set of bits. Hence, there are only $|C| \cdot h^{|C|-1}$ possible matching evaluation values for the client formula. With this observation in mind, we construct the following protocol:

**1.** For each conjunctive clause $C$ the client and the server execute the protocol for the single query matching (without the final stage where server reveals the hashed matching key), and the client learns the value $R_C \cdot g^{\sum_{i \in S_C} b_i}$, where $S_C$ denotes the set of Bloom filter positions to be checked for clause $C$.

**2.** Each of the resulting values is blinded by a public random exponent $L_C$, and the final matching evaluation key is computed as $\prod_{C \in \phi} (R_C \cdot g^{\sum_{i \in S_C} b_i})^{L_C}$.

**3.** The server computes the set Matching of all the possible matching values,

[5] The values $r_i$ across different Bloom filters are independent

and the client *obliviously* does the matching. There are several ways to do the matching. One possibility is to hash and permute all the matching keys, before sending them to the client. Another approach is through a Bloom filter.

The purpose of the exponent $L_C$ is to separate the space of possible values of each clause evaluation, such that there are no overlaps that could (with high probability) make a set of unsatisfying clauses evaluate to a matching key.

**Record Decryption.** After finding the list of identifiers of records satisfying the query, Client can actually retrieve them by querying the ORAM that contains the records. However, as mentioned earlier, in the case of the index ORAM, each ORAM access can potentially reveal records that do not satisfy Client's query. Hence, each document should be encrypted under a key unknown to Client. However, the client should be allowed to decrypt the satisfying records. For this purpose, Owner samples a secret key $s_{\mathsf{r}}$ and, using again the Hashed Diffie-Hellman PRF, it derives for each document $D_i$ an encryption key $k_i \leftarrow F_{s_{\mathsf{r}}} = \mathsf{Hash}(i)^{s_{\mathsf{r}}}$. For each document identifier obtained by Client, Owner and Client execute the OPRFprotocol in Figure 1 to derive the decryption keys.

### 4.2 Final Scheme

**Preprocessing.** The procedure is parametrized by a false positive rate $\mathsf{fp}$ and a security parameter $\lambda$. The database owner starts by choosing a key $s_{\mathsf{bf}}$ for the PRF $F$ and keys $s_{\mathsf{k}}$, $s_{\mathsf{r}}$ for the keyed hash function $H$. It then proceeds by building a Bloom filter Search Tree with false positive rate $\mathsf{fp}$ for the database $\mathsf{DB} = (D_i, W_i)_{i=1}^{D}$, where each keyword $w \in W_i$ is mapped to $H_{s_{\mathsf{k}}}(w)$ forming set $\tilde{W}_i$. Each record $D_i$ is encrypted using the derived key $k_i \leftarrow H_{s_{\mathsf{r}}}(i)$,$\tilde{D}_i = \mathsf{Enc}_{k_i}(D_i)$. The Bloom filter tree is then *encrypted* by encoding each Bloom filter bit $b$ as $g^b$ and encrypting it as $\mathsf{bEnc}_{s_{\mathsf{bf}}}(g^b) = \langle g^b \cdot F_{s_{\mathsf{bf}}}(r), r \rangle$, where $r$ is sampled uniformly random from $\{0,1\}^{\lambda}$. The owner continues by preparing an ORAM structure $(\mathsf{param}_I, \mathsf{struct}_I)$ holding the encrypted index, and the ORAM structure $(\mathsf{param}_D, \mathsf{struct}_D)$ holding the records. In principle, each encrypted Bloom filter bit can be an ORAM block. However, this can be optimized to pack several bits in the same ORAM block to reduce the number of ORAM lookups. We can choose, for example, to hold an entire Bloom filter in one ORAM block, or to pack together bits in the same position across sibling Bloom filters.

We describe next the basic procedures used by the setup phase:

◦ $\mathsf{BFTBuild}(\{\tilde{W}_i\}_{i=1}^{D}, \mathsf{fp}, d)$**:** Let $\mathsf{BFT}$ be a balanced $d$-ary tree of $D$ leafs. Let $L = \lceil \log_d D \rceil$ be the height of the tree. We build the tree level by level, starting from the bottom level $L$. We then proceed recursively until reaching the root of the tree. Let $N_L = \max |W_i|$. Using $N_L$ and $\mathsf{fp}$, compute Bloom filters length $n_L$ and number of Bloom filter hash function $h_L$. Then, we sample $h_L$ independent hash function $\mathcal{H}_L = \{H^{(1)}, ..., H^{(h_L)}\}$ with image $\{0, 1, ..., n_L - 1\}$. For each $i \in [D]$, we build a Bloom filter $B_i$ (using $\mathcal{H}_L$) inserting the elements of $\tilde{W}_i$. We maintain each Bloom filter in a unique leaf of $\mathsf{BFT}$. The internal nodes of the tree are built recursively as follows: we associate each node at level $\ell$ with the keywords held in its children. That is, for each internal node, we build a Bloom filter that contains the elements from all its $d$ children. Return $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, ..., \mathcal{H}_L\}$ and tree $\mathsf{BFT}$. We force the sets of hash functions to be of the same size $h = h_L = |\mathcal{H}_L|$, such that, for each query, the number of lookups in every node is the same. This will prevent the server from learning the level in the tree of the nodes being evaluated.

$\circ$ $\mathsf{BFTEncrypt}(\mathsf{BFT}, 1^\lambda)$**:** Sample a uniformly random key $s_{\mathsf{bf}}$ for PRF $F$. Build a tree $\mathsf{EBFT}$ by: a) encoding each bit $b$ of $\mathsf{BFT}$ as $g^b$, b) encrypting $g^b$ as $\mathsf{bEnc}_{s_{\mathsf{bf}}}(g^b) = \langle g^b \cdot F_{s_{\mathsf{bf}}}(r), r\rangle$, where $r$ is uniformly random in $\{0,1\}^\lambda$. Return key $s_{\mathsf{bf}}$ and tree $\mathsf{EBFT}$.

**Search.** Client inputs a DNF formula $\mathbf{q} = \phi(\boldsymbol{W}) = C_1 \vee C_2 \vee \cdots \vee C_{|\mathbf{q}|}$ on keywords in $\boldsymbol{W}$. The client reveals the query *topology* (number of clauses and size of each clause) to Server. Client and Owner then execute the protocol in Figure 1 to obtain search tokens for each keyword in each clause. For each clause $C$ in the query, Client (or Server) uniformly samples $L_C$ from $[|\mathbf{q}|]$ and sends it to Server (Client). Client and Server then start the tree traversal protocol. For each node being evaluated, both parties proceed as follows:

1. For each clause $C$ of the query, the client computes the Bloom filter positions of the clause's hashed keywords for the node being evaluated, and performs the ORAM queries to get the corresponding encrypted bits $\langle g^{b_i} \cdot F_{s_{\mathsf{bf}}}(r_i), r_i\rangle$.

2. To get each clause evaluation key, Client and Server engage in the Masked-MOPRF protocol, where Client inputs the encryption randomness $r_i$ of each encrypted bit, and Server inputs the PRF secret key $s_{\mathsf{bf}}$. Client obtains $\pi_C = R_C \cdot \prod_{t \in S_C} F_{s_{\mathsf{bf}}}(r_i)$, and Server obtains the random mask $R_C$. Client computes each clause $C$ evaluation key as $\prod_{i \in S_C} (g^{b_i} \cdot F_{s_{\mathsf{bf}}}(r_i)) \cdot (\pi_C)^{-1}$. The key obtained is $\zeta_C = R^{-1} \cdot g^{\sum b_i}$.

3. The client computes each clause evaluation key $K_C = \zeta_C^{L_C}$, and multiplies all keys together to obtain the final evaluation key FinalKey: $\prod_{C \in \phi} K_C = \prod_{C \in \phi} (R_C^{-1} \cdot g^{\sum_{i \in S_C} b_i})^{L_C}$

4. The server computes all possible matching keys. That is, for each clause $C$, Server computes the set $\mathsf{Matching}_C = \left\{ (R_C \cdot g^{|C| \cdot h})^{L_C} \cdot \prod_{C' \neq C} (R_{C'} \cdot g^{\nu_{C'}})^{L_C} \right\}$, where each $\nu_{C'} \in \{0, \ldots, |C'| \cdot h\}$.

6. Each node evaluation finishes by checking if Client's FinalKey belongs to the set $\mathsf{Matching} = \bigcup_C \mathsf{Matching}_C$. This can be done securely by computing a Bloom filter with all matching keys and sending the filter to the client, or by sending a permutation of all hashed keys.

After the tree traversal, Client gets the indices of all documents satisfying the query. It can obtain the documents by querying the documents ORAM structure. To obtain the document decryption keys, Client and Owner execute protocol OPRF, where Owner inputs key $s_{\mathsf{r}}$ and Client inputs the document identifiers. A formal description of the protocol is presented in the full version of this work.

## 5 Evaluation

In this section, we quantify the performance of the encrypted index traversal of our OSPIR protocol by both showing the results of running our prototype on datasets of 1K, 10K, and 100K records, and providing an asymptotic analysis of performance.

**Experimental Setup** Motivated by the audit log application on cloud services, we collected provenance data from an Ubuntu 14.04 system running Apache. From this data, we built a single table database containing on each row a node from the provenance graph and its annotation. We set up two Intel Xeon E5-2430 2.2Ghz (2 cores of 12 threads), 100GB RAM machines with Broadcom 1GB Ethernet. Server and Owner run on the same machine. Our system parameters were set so that the index for the 100K records database fits in 100GB of RAM. Specifically, we fixed the degree of the tree to 10, the Bloom filter false positive to $10^{-5}$, and the number of searchable keywords per record to 4.

**Queries.** We ran SELECT-id queries that match a single record. The performance of the queries that return one result provides the worst-case latency per record, since queries returning several records do not need to inspect already-evaluated nodes. Additionally, by returning just the record identifier, we can evaluate exactly the cost of the search procedure. The types of queries covered were single term, conjunctions, disjunctions and 3-DNFs.

**Conjunctions vs. Disjunctions**. Figure 3 shows, in $\log_{10}$ scale, the latency time for conjunctions and disjunctions of sizes 1, 2, 3, 4, and 5 on a 100 K records database. We observe that while conjunctive queries run in a few seconds, disjunctive queries are exponentially more expensive. It is interesting to note that the number of ORAM queries performed by both types of queries is exactly the same; hence, the latency time is dominated by the cryptographic operations and the data transfer of the matching keys set. In the case of disjunctive queries, we also note that the use of multiple cores does not reduce the latency significantly (at most a factor of two for 24 cores). In the case of conjunctions, the evaluation is entirely sequential and the use of multiple cores has no effect.

**Varying Database Size.** Figure 4 shows the latency for different DNF queries across databases of sizes 1K, 10K, and 100K. The difference between running a query on databases of varying sizes is captured in the number of nodes to be evaluated and the potentially larger ORAM size for larger databases. Observe the sub-linearity of our system's running time: an increase in the database size by a factor of 10 increases the running time by a comparatively small amount, which is due to a single extra evaluation node and a larger ORAM structure.

**ORAM vs. node evaluation.** In Table 1, the second and third columns illustrate the time our prototype spent in ORAM read queries and node evaluation, once ORAM queries have been performed. Since same-size queries require the same number of ORAM operations, the ORAM time is identical for same-size queries. Disjunctive queries, however, exhibit a much more expensive node evaluation execution, since they involve an exponentially large number of possible matching keys, which Server has to compute and hash individually. Moreover, the fourth column indicates that the network usage increases significantly with bigger disjunctions. The reason is that Server also needs to send the set of possible matching keys to Client. In particular, for size-4 3-DNF, the network usage raises to 1GB, and we can infer that for these queries the index traversal will dominate the running time for queries that also return the records' payload.

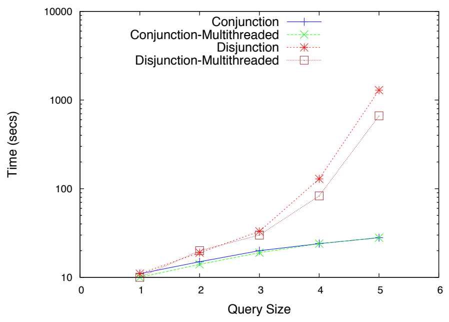

Fig. 3: Latency of conjunctions and disjunctions of sizes 1, 2, 3, 4 and 5 for 100K records DB.

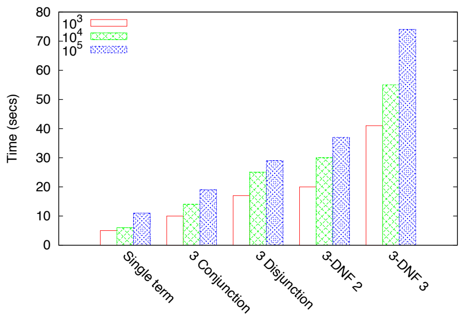

Fig. 4: Query latency time for different-size DNF queries for databases of sizes 1K, 10K and 100K records.

| Query | ORAM | Eval | Network | Query | ORAM | Eval | Network |
|---|---|---|---|---|---|---|---|
| Single Term | 4 | 6 | 26 MB | | | | |
| 2-Conjunction | 9 | 6 | 52 MB | 4-Conjunction | 18 | 6 | 105 MB |
| 2-Disjunction | 9 | 10 | 52 MB | 4-Disjunction | 18 | 90 | 140 MB |
| 3-Conjunction | 14 | 6 | 78 MB | 5-Conjunction | 18 | 6 | 131 MB |
| 3-Disjunction | 14 | 20 | 80 MB | 5-Disjunction | 18 | 90 | 932 MB |
| | | | | Size 2 3-DNF | 25 | 11 | 158 MB |
| | | | | Size 3 3-DNF | 35 | 35 | 249 MB |
| | | | | Size 4 3-DNF | 50 | 680 | 1173 MB |

Table 1: Latency in seconds of tasks in protocol and network usage per query on a 100K records DB.

**Index Size.** One of the drawbacks of our solution is the space utilization of the index. Each bit of a plaintext version of our index is encoded using 140 bytes. Moreover, the index is stored as is in an ORAM structure, which multiplies the space by a non-small constant factor. In our evaluation, each record was associated with 4 searchable keywords. Consequently, for our 100K records dataset, the encrypted index uses 75GB of RAM.

## 6 Conclusions

We proposed an private search scheme that supports Boolean queries and delegated queries. Our system diminishes the leakage of existent solutions, while preserving sublinear search efficiency. Our construction integrates ORAM techniques with efficient search index structures, and leaks to the server only the number of nodes visited in the search tree during the execution of a query. We proposed a new protocol for oblivious PRF evaluation that allows to securely evaluate Bloom filters. This enables the delegated-query feature by disclosing only the match result. Finally, protect the client's queries from the data owner. We implemented our system prototype and ran it on a 100,000-record database. We showed that our system can handle conjunctive queries and small DNF formulas in 10-30 seconds. The sublinearity of our solution, also experimentally illustrated in Figure 4, allows us to extrapolate that queries on much larger databases ($10^6$, $10^7$, and $10^8$ records) will run in a few minutes. The cost of eliminating leakage is substantial; the Blind Seer and OXT-OSPIR systems manage to return records in less than a second for databases of size $10^8$ records with a much larger number of searchable keywords. On the other hand, our system outperforms the secure single-keyword search of the HE-over-ORAM solution whose experimental results showed that their system answers a query in 30 minutes for $4 \times 10^6$ record databases. Therefore, our scheme provides a new tradeoff mark between privacy and efficiency.

## Acknowledgments

This work was funded by the US Department of Homeland Security (DHS) Science and Technology (S&T) Directorate under contract no. HSHQDC-10-C-00144. The views and conclusions contained herein are the authors' and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DHS or the US government.

While at Columbia University, Fernando Krell was supported by NSF awards #CNS-1445424 and #CCG-1423306.

Mariana Raykova is supported by NSF grants CNS-1633282, 1562888, 1565208, and DARPA W911NF-15-C-0236, W911NF-16-1-0389.

## References

1. A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate RAM programs with malicious security. In *EUROCRYPT*, 2015.
2. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.
3. D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of EUROCRYPT'04*, pages 506–522, 2004.
4. D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO'13*, pages 353–373, 2013.
5. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6), 1998.
6. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, 2006.
7. B. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: A scalable private dbms. Cryptology ePrint Archive, Report 2014/963, 2014. `http://eprint.iacr.org/`.
8. C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *ACNS*, 2015.
9. Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. *J. Comput. Syst. Sci.*, 60(3):592–629, 2000.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.
11. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

12. S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.
13. Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, pages 90–107, 2016.
14. M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
15. S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *CCS*, 2013.
16. S. Jarecki and X. Liu. Fast secure computation of set intersection. In *SCN*, pages 418–435, 2010.
17. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.
18. V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *IEEE S&P*, 2014.
19. V. Pappas, M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Private search in the real world. In *ACSAC '11*, pages 83–92, 2011.
20. R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.
21. M. Raykova, B. Vo, S. Bellovin, and T. Malkin. Secure anonymous database search. In *CCSW 2009.*, 2009.
22. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P*, 2000.
23. A. C. Yao. Protocols for secure computations. In *FOCS*, 1982.