# Malicious-Client Security in Blind Seer: A Scalable Private DBMS

Ben Fisch*, Binh Vo*, Fernando Krell*, Abishek Kumarasubramanian†, Vladimir Kolesnikov‡, Tal Malkin*, Steven M. Bellovin*,
* Columbia University, benafisch@gmail.com, {binh, fkrell, tal, smb}@cs.columbia.edu
† UCLA, abishekk@gmail.com
§Bell Labs, kolesnikov@research.bell-labs.com

***Abstract*—The Blind Seer system (Oakland 2014) is an efficient and scalable DBMS that affords both client query privacy and server data protection. It also provides the ability to enforce authorization policies on the system, restricting client's queries while maintaining the privacy of both query and policy. Blind Seer supports a rich query set, including arbitrary boolean formulas, and is provably secure with respect to a controlled amount of search pattern leakage. No other system to date achieves this tradeoff of performance, generality, and provable privacy.**

**A major shortcoming of Blind Seer is its reliance on semi-honest security, particularly for access control and data protection. A malicious client could easily cheat the query authorization policy and obtain any database records satisfying any query of its choice, thus violating basic security features of any standard DBMS. In sum, Blind Seer offers additional privacy to a client, but sacrifices a basic security tenet of DBMS.**

**In the present work, we completely resolve the issue of a malicious client. We show how to achieve robust access control and data protection in Blind Seer with *virtually no added cost* to performance or privacy. Our approach also involves a novel technique for a *semi-private function secure function evaluation* (SPF-SFE) that may have independent applications.**

**We fully implement our solution and report on its performance.**

## I. Introduction

With the exponential growth of electronically generated, transmitted and stored information comes the question of scalable authorized access to these databases. In many important scenarios there is an additional requirement of security and *privacy* of the data, the query, and the access control policies. These privacy requirements frequently arise in law enforcement and government scenarios, as well as in some business scenarios. For detailed motivation and examples of private DBMS applications, we refer the reader to, e.g., [14], [28].

Combining functionality, scalability and privacy has long been an elusive goal of the crypto and security research communities. Recently, two systems were concurrently proposed [14], [28], both supporting highly practical, sublinear, and privacy protected DB querying, with provable security with respect to a controlled amount of information leakage (e.g., search patterns across multiple queries). The two systems, called OSPIR-OXT and Blind Seer, offer varying features and relative advantages, making each system better suited for different application scenarios (see Section VII for a more detailed discussion). In terms of privacy, Blind Seer [28] offers somewhat stronger guarantees in that they formally ensure that the individual terms of the query formula are privacy-protected. (In contrast, OSPIR-OXT [14] leaks support sizes of the disjunctive formula terms.)

However, a major disadvantage of Blind Seer as compared to OSPIR-OXT is that it is only secure against semi-honest clients, namely clients who honestly follow the protocol specification. Even standard DBMS systems with no client privacy generally have robust access control, which Blind Seer does not provide against actively cheating clients. In fact, a very simple and undetectable deviation from the protocol enables a client to easily circumvent all access control in Blind Seer. This failure to meet a standard requirement and a common feature of database systems severely limits Blind Seer's viability and scope for practical deployment.

Lifting the Blind Seer protocol into the malicious setting using standard techniques is very expensive. For example, the cut-and-choose approach to malicious MPC (cf. [19], [20], [24]) carries the cost of at least 128-fold performance degradation for $2^{-128}$ security. These costs can be made somewhat better using very recent amortized garbled circuit techniques [12], [22]. Still, state-of-the-art generic or specialized techniques result in order(s) of magnitude cost overhead. The result of the present work, surprisingly, is that malicious security against the client can be obtained *for free*. That is, we show how to protect against a *malicious* client at virtually *no additional performance cost*, as well as no privacy or functionality degradation. Our result applies not only to Blind seer, but also to any setting where a potentially malicious party needs to evaluate a private function on a semi-honest party's private inputs. When both parties are semi-honest (or at least the party holding the private function is semi-honest), the *Yao Garbled Circuits* (Yao GC) protocol is a practical method that entirely preserves the privacy of the inputs and reveals no more than the private function's circuit topology. While it is well-known how to achieve this functionality (or even stronger privacy) for malicious players using general and expensive techniques, our technique is as efficient and achieves the same level of privacy as Yao GC.

We still assume that the server in our setting is semi-honest. In many natural applications, both in business and government, the trust in the server (e.g., Bank) is much higher than in the client. This is often because the server is operated by a business or an agency, and would risk a high legal penalty for actively compromising the privacy of a client. (Note, however, that we do *not* trust the server with private information).

### *Our Contributions*

– **Malicious-client security in Blind Seer:** We present the first design and implementation of a DBMS that features both fully robust access control and private arbitrary boolean querying, with performance about 2-3 times slower than (insecure and non-private) MySQL. The access control mechanism is highly expressive, and can implement any policy dependent on the client's query.

– **Novel SPF-SFE technique:** We give an extremely simple and efficient protocol for a *semi-private function secure function evaluation* which allows for secure function evaluation of any private function of known circuit topology that is held by the party who will receive the output.

– **Formal proofs:** We formally prove security against arbitrary malicious behavior of the client. We note that full cryptographic proofs are unusual for large systems such as ours. Our other privacy features and leakage profiles remain nearly the same as those of the original Blind Seer.

– **Implementation and Performance:** We implement our design of malicious-client secure Blind Seer, and discuss the details in Section VI-A. We compare the performances of the new design, the original Blind Seer design, and MySQL. We also demonstrate a greatly improved performance by implementing batching and parallelization within query processing. Since the original design did not support multi-threading, we ran the new system on a single thread when collecting the comparison data. When running our system with 16 threads on a 10TB, 100M-record DB, typical queries run in time comparable to MySQL, or up to only 3 times slower. This is more than a 5-fold improvement over the single threaded Blind Seer (while security is significantly better).

### *Organization*

In the next section (Section II) we provide an overview of the malicious-client vulnerability in Blind Seer and our solution. In Section III we present our main building block for malicious security—our new SPF-SFE protocol. We describe in detail our complete malicious-client secure system in Section IV, and formally analyze its security in Section V. We discuss implementation and performance features in Section VI and end with related work in Section VII.

## II. Overview

This section provides an overview of our solution for achieving malicious-client security in Blind Seer. We will review the basic architecture of Blind Seer [28], point out how the previous design was vulnerable to malicious-client cheating, and then describe how we address that vulnerability.

**Preliminaries** We use Bloom filters (BF), semantically secure encryption (both public key and symmetric key), Yao Garbled Circuits (GC), and Oblivious Transfer (OT). All these standard cryptographic primitives are described in Appendix A.

We use Yao GC to achieve *secure computation*, also called *secure function evaluation (SFE)*, which intuitively means that the two-party function is computed so that each of the parties learns no information except what follows from their own inputs and outputs. One may also consider *Private Function SFE (PF-SFE)*, where in addition the function that is being computed is itself the input of one of the parties, and remains hidden from the other party. A generalization of this is *Semi-Private Function SFE (SPF-SFE)*, where the function is known to belong to some class of functions, but beyond that remains hidden. Here, we will consider SPF-SFE where the function is known to have a certain *topology*. Explicitly, the structure of the gates in the circuit describing the function is known, but the operation of each gate (e.g., OR or AND) remains hidden. (See more details in Section III). Yao GC is an example of a protocol that achieves this property. Yao GC involves one party sending a "garbled circuit" to the other, and while the technique was not designed to hide the function, it turns out to hide the values of the gates.

In our setting, *OT preprocessing* [1] and *OT extension* [13] dramatically improve performance. We use the Naor-Pinkas protocol [25] for the "base" OTs that seed OT extension. We then use a version of the IKNP protocol for OT extension suggested by Nielsen [26] that is robust against a malicious receiver with small additional cost.

### *The Blind Seer DBMS*

We review here the basic features of the Blind Seer DBMS provided by [28]. We refer the interested reader to that paper for more details, as well as discussion and motivation for the setting and design choices (some of which are also discussed in [3], [14]). However, we stress that further details of the Blind Seer design beyond what is described here and in Section IV are not necessary for understanding the malicious-client vulnerability of the original design and the contributions of the present work.

**Participants.** The Blind Seer system consists of three main parties: a *server* $\mathsf{S}$, a *client* $\mathsf{C}$, and a third party server called the *index server* $\mathsf{IS}$. The server $\mathsf{S}$ owns a database DB. The client $\mathsf{C}$ submits queries and retrieves records satisfying those queries. $\mathsf{IS}$ holds an encryption of DB as well as an encrypted index to DB, and facilitates the private and efficient evaluation of the client's queries (without learning either the query or the data).

A fourth logical entity, called the query checker QC, is responsible for enforcing policy restrictions on the client's queries. For instance, a policy restriction might prohibit queries that ask for records associated with an important political figure. QC may be run by the server $\mathsf{S}$, but we will view it as a separate logical entity. As long as the separate parties do not collude, $\mathsf{C}$ only learns the results of queries that pass the policy, $\mathsf{C}$'s query is kept private from all other parties, the policy is kept private from $\mathsf{C}$, and the results of the policy check are unknown to any party.

**Architecture.** The basic architecture of Blind Seer is depicted in Figure 1. The server S, who holds the DB, will hand an encrypted copy of the DB to the third party server IS. In addition, S builds an encrypted Bloom filter (BF) tree index to the DB and sends it to IS.

The BF tree index is constructed as follows. The records of the DB are randomly permuted, and a $b$-ary tree is initialized with a one-to-one correspondence of tree leaves to records in the DB. A BF is placed at every node in the tree. Each leaf-node BF holds all the (indexed) keywords of its corresponding DB record. Each internal-node BF contains the union of all the keywords inserted into its children BFs. Keywords are inserted into a BF using $k$ cryptographic hash functions (see Appendix A for details on Bloom filters). Finally, each BF in the tree is encrypted with a one time pad generated from a keyed pseudorandom function, which (via the PRF key) is given to the client C.

The client submits all its queries to IS only. A query is expressed as a boolean formula over keyword terms (e.g., `fname:Jeffery` $\wedge$ `lname:Smith`). Each keyword terms tests the presence of a keyword in a Bloom filter, and so expands into a conjunction of $k$ predicates, each testing a single bit in the input BF. The query is interactively evaluated on each node down the BF tree. The query processing proceeds to children nodes only if it returns true on their parent node. When a query returns true on a leaf-node BF, the associated record is returned to the client. A Yao garbled circuit protocol variant is used to evaluate the query at each BF node. To avoid garbling a circuit that takes an entire Bloom filter as input, C reveals to IS the relevant BF indices that the query circuit examines ($k$ for each individual keyword term). Note that IS does not know the value of the BF at those indices because the BF is encrypted.

QC and C also separately engage in a computation of the policy circuit via the Yao GC technique, and with the help of IS. Specifically, C chooses key pairs for the input and output wires of the policy circuit, sends these pairs to QC, and sends only the keys corresponding to its query input to IS. In turn, QC garbles the secret policy circuit using the client's key pairs, and sends this garbled circuit to IS for oblivious evaluation on the client's inputs. The output of the policy circuit is integrated into the query circuit evaluation so that C learns only the AND of the two circuit outputs.

Finally, if the query evaluation (and policy approval) on any leaf record outputs success, IS sends the record and decryption information to C. C obtains any final decryption information from S.

**Privacy and Leakage.** Ideally, the client C would learn nothing except the result sets of its authorized queries, and the servers S and IS (as well as the query checker QC) would learn nothing at all. Blind Seer achieves privacy up to a controlled amount of leakage of *search patterns*. IS may correlate repeated queries, and both C and IS may observe traversal patterns through the tree index across multiple queries, possibly obtaining some correlation information between different queries. Additionally, since Yao's GC technique is used, IS

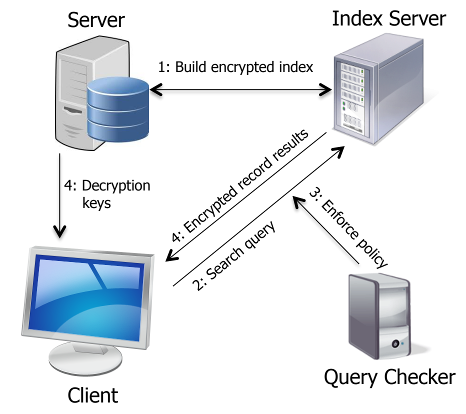

Figure 1. Basic architecture of Blind Seer.

does learn certain structural information about C's query and QC's policy, namely their circuit topologies. However, the individual search terms (i.e., keywords) and logical gates are hidden (as Yao GC provides SPF-SFE, leaking only the circuit's topology).

The privacy guarantees of Blind Seer have been formally proven with respect to *semi-honest* adversaries using the simulation paradigm. Semi-honest adversaries will not deviate from the prescribed protocol, but may attempt to learn arbitrary information from their view of the protocol. The controlled leakage was formally captured by including a leakage oracle in the ideal world functionality definition.

### *Malicious-Client Vulnerabilities in Blind Seer*

The Blind Seer policy enforcement collapses with a malicious client because there is no mechanism for evaluating an authorization policy directly on the client's private query. The client submits one query (represented as a garbled circuit) to be evaluated privately in the DB search protocol, and a second query (represented as input wires to the policy circuit) to be evaluated privately in the authorization policy protocol. An honest client will submit the same query to both protocols, but a malicious client could submit entirely separate query inputs to the search protocol and policy protocol, rendering the access control mechanism completely ineffective. Crucially, there is *no risk of detection* for C.

### *The Solution*

Our solution simultaneously cryptographically binds the client's inputs to *both* the query evaluation and policy check circuits, and encrypts the DB record results under a key that can only be obtained when *both* the outputs of the query and the policy are positive. Of course, this could be accomplished using any number of off-the-shelf expensive techniques, such as zero-knowledge or fully malicious-secure SFE, but our goal is to maintain the efficiency of Blind Seer.

Blind Seer uses Yao Garbled Circuits (GC) in both the search protocol and the policy protocol. Yao GC is one of the fastest secure two-party secure computation protocols for semi-honest players. In the Yao GC protocol, the two parties are distinguished as *generator* and *evaluator*. The generator selects the function to evaluate and "garbles" the function. The evaluator is able to obliviously evaluate the garbled function using both his own inputs and secret inputs that the generator supplies. While Yao GC satisfies only semi-honest security against the generator, it offers malicious security against the *evaluator*. Furthermore, Yao's GC is a special case of SPF-SFE in that it leaks only the boolean circuit topology of the function to the evaluator. The garbled circuit generator can even choose to cryptographically bind any of the evaluator's inputs by synchronizing the decryption key pairs on the corresponding input wires. The evaluator receives only one of the two keys through a single oblivious transfer, and is forced to reuse the same key for both wires.

The difficulty in using standard Yao GC to achieve malicious-client security in Blind Seer arises since the client is the generator, rather than the evaluator, of the query circuit in the search protocol.

**The crux of our solution** is a low-cost way of converting the client's query from a circuit to a circuit input, thereby swapping the role of the client from generator to evaluator. There are several challenges that we need to overcome in order to achieve this without sacrificing either efficiency or privacy:

1) How does IS play the role of garbled circuit generator without knowing the query circuit (that should remain hidden from IS)?
2) How do we link inputs to the query and policy that are related but formatted differently? For instance, where the query circuit takes a keyword `field` : `value`, the policy circuit might take only the field name `field`.
3) Recall how each DB index Bloom filter is encrypted with a randomly generated one-time pad. C and IS receive random shares of each BF from S, and both parties submit these bits as inputs to the query circuit. How do we prevent a cheating client from faking positive query results by flipping some of its BF input bits?

**Universal query circuit.** IS can play the role of generator in the query evaluation protocol without knowing the query by using a *universal circuit*. A universal circuit $UC_{\mathcal{F}}$ is a well-known construction that can simulate any circuit $C$ in the family $\mathcal{F}$. Specifically, it is a circuit that will take as input the description of any circuit $C \in \mathcal{F}$, any input $x$, and will output $C(x)$. There are many constructions of universal circuits $UC_k$ that can simulate any circuit of size $k$ [18], [31]. This is a very powerful tool for PF-SFE since it can be used to hide everything about the private function except its size. Indeed, a universal circuit provides exactly what we need in terms of swapping the role of the circuit generator to be input provider, while maintaining circuit privacy. Unfortunately, however, constructing a general universal circuit $UC_k$ results in a significant increase in circuit size, and thus a very high overhead in performance when evaluating it through the Yao GC technique.

Instead, we take advantage of the fact that we do not need full function privacy for PF-SFE, since the topology of the circuit is already leaked to the IS even in the previous solution. We construct a much simpler universal circuit $UC_{\mathcal{T}}$ that simulates any monotone circuit with topology $\mathcal{T}$, with virtually no overhead in its secure computation. We stress that considering only monotone circuits is not a limitation, since Blind Seer's tree traversal (and any natural DB tree index traversal, for that matter) only works for monotone query circuits (i.e., containing only `AND` and `OR` gates) over keyword terms, where negations are pushed to the variable level (something which can always be done efficiently). Also note that the keyword terms are computed by conjunctions of $k$ XOR gates, each taking one input from each party. However, since these circuits computing the keyword terms are fixed and known to both parties, they are not included in the universal query circuit.

Our construction of $UC_{\mathcal{T}}$ from $\mathcal{T}$ increases the number of gates by a factor two, but the extra gates are all XOR gates. Thus, when using the *free-XOR* technique [17], the cost of securely computing $UC_{\mathcal{T}}$ with Yao GC has roughly the same cost as securely computing any monotone circuit $C$ with topology $\mathcal{T}$.

In our new protocol, C sends the topology of its query circuit $Q$ to IS, and IS generates the corresponding universal query circuit $UQ$. IS garbles $UQ$ and sends it back to C for evaluation. Note that sending the topology of $Q$ does not increase leakage to IS because running Yao GC directly on $Q$ (with C as generator and IS as evaluator) also reveals its topology.

**Policy circuit.** QC garbles a policy circuit $PC$ and sends it to C for evaluation. The circuit outputs a key that reveals no information on its own, but is used to evaluated a garbled conjunction of the policy and query: $UQ \wedge PC$. The key difference from the previous design mechanism is that C will *not* submit any separate input to QC. Instead, the client commits to its query only once, and receives from IS all the keys it needs to evaluate both $UQ$ and $PC$.

QC and IS exchange information on the keys used in the garbling of $UQ$ and $PC$ in order to synchronize the keys used for common inputs to both circuits and so that IS can respond to the client's query with the appropriate keys. The new protocol also requires C to separately submit cryptographic hashes of all the field names and keywords used in its query. Keywords are inserted into the Bloom filter index in a way that binds each keyword hash to its corresponding field hash.

While the query circuit must be evaluated on every BF node encountered during the index tree traversal, it is only necessary to evaluate $PC$ once per query. The only inputs to $UQ$ that change as the BF node changes are the BF input bits to the keyword terms. The policy circuit is a fixed function of the query, and its output should be unaffected by the BF input. Thus, as long as the same query is executed on every node of the BF tree, the $PC$ input should remain the same at every node. By reusing the same keys for all invariant inputs, we guarantee this property. Likewise, the $PC$ is evaluated once, and its output key can be reused for every subsequent query-policy conjunction gate.

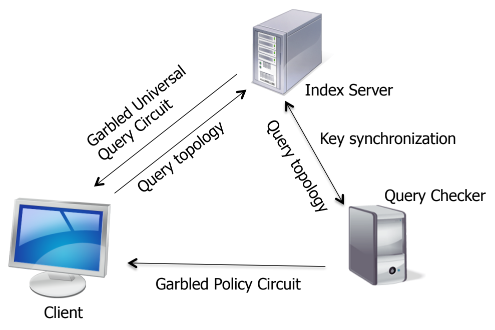

Figure 2. Malicious-client secure protocol overview.

QC does not learn anything about the query except its topology, which it uses for constructing the policy circuit. We may also avoid this leakage at some cost of efficiency by using universal circuits for the policy. In fact, since the policy circuit is only computed once per query, its size is not a critical point for performance.

**Supported policies.** The system supports a rich class of policies. The policy can be any function of the keywords, field names, and syntax (structure) of the query. An example of such a policy is: any conjunctive query that includes the keyword "lname = Obama" cannot include any keyword on the field "income." While our design essentially supports any policy that is dependent on the query, it doesn't support policies that might depend on the data as well. For instance, we would not support a policy that prohibits queries that have fewer than 20 matching records.

**Bloom filter FPR.** The final question we posed was how to prevent C from faking positive query results by flipping input bits from its BF shares. C's share is a pseudorandom mask, and reveals no information on which bits in the BF are 1 or 0. In order to fake positive results, C must set $k$ specific bits to 1. C can only do this by flipping bits, and it does not know if any given bit is initially a 0 or a 1. By setting parameters appropriately, we can choose a false positive rate (FPR) that makes these events equally likely. This way C only succeeds in setting any given bit with probability 1/2.

**DB security and policy privacy.** For each leaf node reached in the tree traversal, IS sends C a final garbled gate that computes the AND of the policy and universal query circuit outputs. IS sends the record $R$ associated with this leaf node to C, but encrypted under the output key $\text{OUT}_R^1$ of this final gate. Concretely, C needs to obtain the 1-key output of both the garbled $PC$ and $UQ$ in order to obtain $\text{OUT}_R^1$ and decrypt the record received.

While policy failure prevents the client from feasibly obtaining any records of the database, the tree traversal pattern still leaks partial information about the database. Alternatively, evaluating the conjunction of $PC$ and $UQ$ at the root of the search index prevents such leakage, but arguably affords less privacy to the policy. In fact, the policy conjunction can be evaluated anywhere inside the tree traversal, and this design decision is left open.

We formalize and prove security properties of our system in Section V. These properties are proved with respect to a malicious client adversary and a semi-honest index server adversary. The definitions are general enough so that open ended design decisions do not invalidate any of the proofs (e.g. where to evaluate the PC), and are intended to elucidate the tradeoffs of such decisions.

**Optimizations.** To optimize performance, the universal query circuit protocol is only run on leaf nodes of the Bloom filter tree index, where it matters the most. As a further optimization, the policy circuit will only be evaluated once, and its output key will be reused for each leaf node.

## III. Semi-Private Function SFE

*Private Function Secure Function Evaluation* (PF-SFE) is a two-party functionality in which a private function known only to one party (the *selector*) is evaluated on private inputs of both parties, and nothing but the outputs and function size is revealed. *Semi-private function SFE* (SPF-SFE) [29] is a generalization of PF-SFE in which the private function is chosen from any restricted class of functions known to both parties.

Since Yao's GC protocol reveals only the circuit topology to the evaluator, it can be viewed as a special case of SPF-SFE, where both parties may learn the topology $\mathcal{T}$ of the private function, but only the selector knows the identity of each gate in the circuit (AND, OR, XOR, etc). This requires the function selector to be the garbled circuit generator. Here, our goal is to construct such a protocol where the function selector is the garbled circuit evaluator. While it is known that this (and even PF-SFE) can be accomplished using universal circuits (UC), applying a general UC transformation would be expensive. Instead, in this section we present a simple protocol which is virtually as efficient as the standard Yao GC protocol, as long as the circuit topology is monotone, with all negations pushed to the input level (any circuit can be easily and efficiently converted to this form). To achieve this, we capitalize on the fact that the topology $\mathcal{T}$ is known to both parties, and take advantage of the free-XOR technique.

$UC_{\mathcal{T}}$ **construction.** First, a fan-in two (i.e. two wires per gate) circuit with topology $\mathcal{T}$ is constructed out of *universal gates*, or "blank gates" that do not have any pre-defined functionality. A third input wire is added to each universal gate, and represents the value of the gate (either AND or OR). Equivalently, each universal gate is a function $G(b, x, y)$ of three bits so that $G(0, x, y) = \text{OR}(x, y)$, and $G(1, x, y) = \text{AND}(x, y)$. Next, each fan-in 3 universal gate is replaced by the fan-in two cluster:

$$b \oplus ((x \oplus b) \ \vee \ (y \oplus b))$$

**Protocol 3.1: Yao Semi-Private Function SFE with Selector As Evaluator**

Party $P_1$ selects a monotone boolean circuit $C$. $P_1$ has input $\mathbf{x}$ and $P_2$ has input $\mathbf{y}$. The topology $\mathcal{T} = topo(C)$ is known to both parties.

1) $P_2$ constructs the universal circuit $UC_{\mathcal{T}}$ and generates its corresponding garbled circuit $\tilde{UC}_{\mathcal{T}}$ according to the free-XOR GC protocol. $P_2$ sends the tables of $\tilde{UC}_{\mathcal{T}}$ to $P_1$ along with the keys corresponding to the input bits $\mathbf{y}$.
2) $P_1$ runs OT with $P_2$ to receive the keys corresponding to its input bits $\mathbf{x}$ and its gate value bits $\mathbf{b}$.
3) $P_1$ evaluates the garbled circuit $\tilde{UC}_{\mathcal{T}}$ and obtains the output.

**Efficiency.** Each gate of the original circuit is replaced with a cluster of 3 XOR gates and 1 OR gate. Thus, the number of non-XOR gates remains constant. The cost of applying the free-XOR GC protocol to $UC_{\mathcal{T}}$ is roughly the same as applying it to the original circuit.

**Applications.** The capabilities of the generator and evaluator in Yao GC are not symmetric, particularly when dealing with malicious adversaries. For instance, Protocol 4.1 is useful for *repeated* SPF-SFE, where the same private function is to be evaluated more than once on different inputs, or possibly given as input to other functions. The selector could cheat as the garbler by using different functions for each set of inputs. But when the selector is the evaluator, the garbler can enforce consistency of the function. A special case is the problem that we address in Blind Seer, where the private function (client's query) must be evaluated on the DB index and also supplied as input to the policy check. A second capability of the generator is to encrypt a message under an output key from the garbled circuit so that the evaluator can only decrypt the message contingent on the output of the function.

## IV. System Protocol

This section details our new system protocol for Blind Seer. We describe the protocol for a single client C, a server S, and an index server IS. The outline of the protocol is as follows:

**Stage 1: Preprocessing**. S encrypts the database, builds an encrypted Bloom filter tree index, and sends these encrypted objects to IS.

**Stage 2: Client queries.** C builds a logical circuit Q representing its query, sends Q's topology to IS, together with hashes of all the field names and field values used in Q. IS uses Q's topology to construct a *universal query circuit* $UQ$ (Section III) equivalent to Q with output keys $\texttt{OUT}_u^{UQ}$, $u \in \{0,1\}$.

**Stage 3: Policy evaluation.** The query checker QC generates and sends C a garbled *policy circuit* (PC), C obtains the keys for evaluating PC from IS, and C computes the output key of the policy evaluation $\texttt{OUT}_p^{PC}$, $p \in \{0,1\}$.

**Stage 4: Tree traversal.** C and IS begin a multi-threaded traversal of the Bloom filter tree index, evaluating the query Q on each node processed using Yao GC with C as generator and IS as evaluator. Upon reaching a leaf node, the protocol proceeds to Stage 5.

**Stage 5: Leaf (record) nodes.** When a leaf node is reached, IS and C use the universal circuit UQ constructed in Stage 2 and the SPF-SFE protocol (Protocol 3.1) to evaluate Q on that node, and outputs $\texttt{OUT}_u^{UQ}$, $u \in \{0,1\}$. In addition, IS sends C a garbled $\texttt{AND}$ gate that takes $\texttt{OUT}_u^{UQ}$ and $\texttt{OUT}_p^{PC}$ as input, and gives a final output key $\texttt{OUT}_{u\cdot p}$. IS sends to C the encrypted record $\tilde{R}$ at this leaf node doubly encrypted as $\texttt{Enc}_{\texttt{OUT}_1}(\tilde{R})$.

**Stage 6: Record retrieval.** When a query is successful in satisfying both $UQ$ and $PC$, C uses the output keys $\texttt{OUT}_1^{UQ}$ and $\texttt{OUT}_1^{PC}$ to obtain $\texttt{OUT}_1$, and decrypts $\texttt{Enc}_{\texttt{OUT}_1}(\tilde{R})$. Finally, C uses the decryption keys obtained from the server S to decrypt $\tilde{R}$. Note that C always asks for the decryption information from S and always receives an encrypted record from IS, however, it only is able to successfully decrypt the record if both $UQ$ and $PC$ evaluated to true.

### *Stage 1: Preprocessing*

**Shuffle and Encrypt Records.** Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure homomorphic public key encryption, $(\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$ a semantically secure symmetric key scheme. S randomly permutes the database records, and encrypts each record $R_i$ as:

$$(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda), s_i \leftarrow \{0,1\}^\lambda,\ \tilde{s}_i \leftarrow \mathsf{Enc}_{pk}(s_i),\ \tilde{R}_i \leftarrow \mathsf{enc}_{s_i}(R_i).$$

S sends $(pk, \{(\tilde{s}_i, \tilde{R}_i)\}_{i=1}^n)$ to IS.

**Generate Encrypted Index.** S builds a balanced $b$-ary tree index $T$ of Bloom filters. Each leaf of the tree is associated with a unique database record, and a Bloom filter holding all of that record's indexed keywords. Each internal node filter $B_v$ holds all of the keywords of its children.

We introduce a subtle but significant change to the format of keyword insertions. For the purpose of efficient policy checking, C will separately submit hashes of both the field names and keywords in its query. In order to bind corresponding keyword hashes and field hashes, S inserts the concatenation of these hashes into the Bloom filters.

*Hash function generation.* S chooses random keys $k_c, k_s \leftarrow \{0,1\}^\lambda$. S sends $k_c$ to C and QC, and sends $k_s$ to IS. S then generates hash functions $\mathcal{H} = \{h_i : \{0,1\}^* \rightarrow [\ell]\}_{i=1}^\eta$. S can choose $H_1$, $H_2$ independently and set $h_i(x) = H_1(x) + i \cdot H_2(x)$ [16]. ($\ell$ is chosen to satisfy the desired false positive rate). We use the notation $\mathcal{H}(x) = \{h_i(x) : h_i \in \mathcal{H}\}$. Keyed hashes are derived naturally as $\mathcal{H}_k(x) = \mathcal{H}(k||x)$.

*Inserting keywords.* To insert the keyword $\texttt{field} : \texttt{value}$ in a filter $B_v$ of length $\ell_v$:

- Derive the set $I$ of BF index values by computing $I = \mathcal{H}_{k_s}(H_{k_c}(\texttt{field})||H_{k_c}(\texttt{field} : \texttt{value}))$
- $\forall i \in I$, set $B_v[i \mod \ell_v] = 1$

*BF mask:* Let $F$ denote a pseudorandom function (PRF). S chooses a new key $key \leftarrow \{0,1\}^\lambda$ for the PRF. Let $T$ denote the BF tree. The filter $B_v$ is masked as: $\tilde{B}_v := B_v \oplus F_{key}(v)$. S sends $\{\tilde{B}_v\}_{v \in T}$ to IS and $(key, \mathcal{H}, F)$ to C.

**Prepare Decryption Keys.** To reduce online query latency, the index server and the server precompute decryption keys. S holds the decryption keys $s_1, ..., s_n$ that it used to symmetrically encrypt the records. It also holds $\tilde{s}_i = \mathsf{Enc}_{pk}(s_i)$

for all $i$. S sends to IS these values $\tilde{s}_1, ..., \tilde{s}_n$. IS chooses a random permutation $\psi$, and random values $r_1, ..., r_n$. IS homomorphically computes $\tilde{s}_i + \mathsf{Enc}(r_i) = \mathsf{Enc}(s_i + r_i) = \tilde{s}'_{\psi(i)}$ for all $i$, and sends these value back to S. S decrypts and stores $s'_{\psi(1)}, ..., s'_{\psi(n)}$.

**Multiple clients.** Although the description of the system protocol is for a single client, we can easily support multiple clients without compromising security as long as the clients do not collude. Under this assumption, we can actually use the same keys for all clients. However, if all clients use the same key $k_c$ to encrypt their query keywords, then IS may correlate the queries of different clients. To prevent this, the server S can distribute separate keys $k_c$ to each client, and insert each keyword into the BF separately for each client encrypted under that client's key. The disadvantage of this additional security measure is that the size of the BF will scale not only with the size of the data but also with the number of clients.

*Stage 2: Client Queries*

Every query in the Blind Seer DBMS can be represented as a monotone boolean logical formula over atomic search terms. There are three types of atomic search terms: keywords, ranges, and negations. However, in Blind Seer, any range query from a field with value range $r$ is translated into a disjunction of $O(log\ r)$ keywords queries. A negation of a keyword $\alpha$ is translated into a disjunction of two range queries (i.e. $x < \alpha$ `OR` $x > \alpha$), which is again converted into disjunctions of keyword queries. Thus, the final representation of a query is a monotone logical formula over solely keyword terms (see [28] for details).

**Query circuits.** Queries are computed by *query circuits*. A query circuit transforms each atomic keyword term $\alpha$ into a small circuit computing the presence of $\alpha$ in an input Bloom filter. This is simply a conjunction of the bits at the $\eta$ BF hash indices of $\alpha$, but since C and IS hold separate shares of the input Bloom filter, it is actually a conjunction of $\eta$ XOR gates.

**Committing to Q.** For the purpose of malicious-security, we force C to effectively commit to its query circuit $Q$ as follows.

1) C sends $\mathtt{topo}(Q)$ to IS.
2) For each keyword of the form $\mathtt{field:value}$, C sends to IS the hashes: $H_{k_c}(\mathtt{field})||H_{k_c}(\mathtt{field:value})$.
3) IS generates a garbled universal circuit $UQ$ from $\mathtt{topo}(Q)$ as described in Section III. Each gate in the *query formula* layer of $Q$ is associated with a pair of keys in $UQ$. `OR` maps to a 0-key, and `AND` maps to a 1-key.
4) C does OT with IS to receive the keys corresponding to the gate values (i.e. `AND`/`OR`) of $UQ$.
5) IS computes the set of indices:
   $\mathcal{H}_{k_s}(H_{k_c}(\mathtt{field})||H_{k_c}(\mathtt{field:value}))$
   for each keyword $\mathtt{field : value}$, and sends these back to C.

*Stage 3: Policy evaluation*

At the end of Stage 2, IS has received $\mathtt{topo}(Q)$ and hashes of the form $H_{k_c}(\mathtt{field})||H_{k_c}(\mathtt{field:value})$ for each keyword, and has generated $UQ$. We denote by $G_Q = (g_1, \ldots, g_n)$ the gate value inputs to the garbled $UQ$, and by $K_Q = (\alpha_1, \ldots, \alpha_t)$ the query's keywords. Each keyword $\alpha_i$ is associated with a field $f_i$. We use the following notations: $F_Q = (f_1, \ldots, f_t)$, $H_{k_c}(K_Q) = (H_{k_c}(\alpha_1), \ldots, H_{k_c}(\alpha_t))$, and $H_{k_c}(F_Q) = (H_{k_c}(f_1), \ldots, H_{k_c}(f_t))$. Recall that QC receives $k_c$ from S during preprocessing.

**Policy functions.** A query policy is any function $\mathbf{p}$: $\mathcal{Q} \to \{0, 1\}$, where $\mathcal{Q}$ is the space of queries. Our system can implement as a policy any boolean function of the query keywords, fields, and syntax (with a tunable probability of error). More precisely, we can implement any function of $(Q, H_{k_c}(K_Q), H_{k_c}(F_Q))$, and so the error probability of simulating $\mathbf{p}$ is proportional to the collision probability of $H$.

**Policy circuit structure.** There are three types of inputs to policy circuits: gate values $G_Q$, keyword hashes $H_{k_c}(K_Q)$, and field hashes $H_{k_c}(F_Q)$. A policy circuit consists of an upper *logical layer* built over a bottom layer of *keyword check gates* and *field check gates*. A *keyword check gate* is associated with a blacklist/whitelist set of keywords, and evaluates whether its input hashed keyword $H_{k_c}(\alpha)$ is in the set. Likewise, each *field check gate* is associated with a subset of the database fields, and indicates whether its input hashed field $H_{k_c}(f)$ is a member of that set. The inputs $G_Q$ are fed directly into the logical layer along with the outputs of the keyword check gates and field check gates.

**Keyword check gates.** Let $\mathcal{L}$ denote the blacklist/whitelist of the gate, and let $\alpha$ denote the keyword submitted by the client. QC initializes a Bloom filter, and inserts into it $H_{k_c}(w)$ for all $w \in \mathcal{L}$. QC generates a mask for this filter and sends it to IS. Recall that IS holds $H_{k_c}(\alpha)$. QC and IS build a garbled circuit evaluating the presence of $H_{k_c}(\alpha)$ in the filter, and send both the garbled circuit and its input keys to C for evaluation.

**Field check gates.** Let $\mathcal{F} = \{f_1, \ldots, f_m\}$ denote the fields of the database schema. Let $\pi : [m] \to [m]$ be a random permutation. A field check gate consists of the following elements.

– *Field function.* A boolean function $b : \mathcal{F} \to \{0, 1\}$.
– *Permuted key table.* A table of keys $[k_{\pi(1)}, \ldots, k_{\pi(m)}]$
– *Output keys.* A pair of output keys $k_0^{\mathrm{OUT}}, k_1^{\mathrm{OUT}}$.
– *Garbled table.* A table of encrypted output keys:
$$[\mathtt{enc}_{k_{\pi(1)}}(k^{\mathrm{OUT}}_{b(f_{\pi(1)})}), \ldots, \mathtt{enc}_{k_{\pi(m)}}(k^{\mathrm{OUT}}_{b(f_{\pi(m)})})]$$

*Evaluation:* The private evaluation of a field check gate on a field $\mathtt{field}$ between QC, IS, and C is very simple. IS receives $H_{k_c}(\mathtt{field})$ from C. QC sends the permuted key table along with the mapping of field hashes into the table so that IS may locate the key corresponding to $H_{k_c}(\mathtt{field})$, and send it to C. QC sends the garbled table to C, who locates and decrypts the appropriate output key using the input key received from IS, exactly as in Yao.

**Policy False Positives and Negatives.** The use of Bloom filters for evaluating the keyword gates introduces a tunable false positive rate in the outcome of the gate. This contributes to either a false positive rate $FPR_p$ or false negative rate

$FNR_p$ of the overall policy, depending on how the keyword gates are used in the logical layer of the policy circuit (e.g. keyword blacklists may cause false rejects, and keyword whitelists may cause false approvals). Since the protocol only requires storing one relatively small policy circuit in RAM, we can afford to make the false positive rates of the keyword gate Bloom filters sufficiently small. For example, a policy that has 10 keyword gates, 10 keywords per gate filter, and overall error rate $2^{-256}$ would only require approximately 4.6 GB of space at most.

**Policy protocol.** IS initiates the policy evaluation.

1) IS sends $\mathtt{topo}(Q)$ to C along with key pairs for the gate value wires of the garbled $UQ$ (i.e. the key pairs for the inputs $G_Q$).
2) Given $\mathtt{topo}(Q)$, QC generates the policy circuit $PC$. The input to $PC$ is $(G_Q, H_{k_c}(K_Q), H_{k_c}(F_Q))$.
3) QC generates a garbled circuit from $PC$. It sets the key pairs for the inputs $G_Q$ using the key pairs received from $IS$, and it generates all other key pairs randomly (as in the usual Yao garbled circuit construction). It sends the garbled tables of $PC$ to C, and sends the key pairs for all the input wires to IS. Additionally, it sends the key pair $\{\textsc{out}_0^{PC}, \textsc{out}_1^{PC}\}$ for the policy output wire to IS.
4) Using the table of keys received from QC, IS identifies the input keys to $PC$ corresponding to inputs $H_{k_c}(K_Q)$ and $H_{k_c}(F_Q)$. IS sends these keys to the client. Note that the client has already received the keys for the inputs $G_Q$ via OT in Stage 2 (these are the same input keys that C will use for evaluating $UQ$ in Stage 5).
5) Finally, C uses the keys received from IS in (4) and the garbled tables received from QC in (3) to evaluate $PC$, and obtains the output policy key $\textsc{out}_p^{PC}$, $p \in \{0, 1\}$.

*Stage 4: Tree traversal*

C and IS begin a multi-threaded breadth first traversal of the Bloom filter index tree. They do not process leaf nodes at this stage. At any non-leaf node $v$ visited, C and IS evaluate $Q$ on the Bloom filter $B_v$ (IS's input bits are derived from $\tilde{B}_v$ and C's input bits are derived from the mask $F_k(v)$ at the hash indices computed in Stage 2). They use the following Yao GC variant:

1) C garbles $Q$ and sends the garbled circuit to IS. C also sends keys for its own input bits.
2) IS executes OT with C to obtain Yao keys for its input bits, evaluates the garbled circuit, and sends the output key back to C.
3) C learns the output value from the output key.

When the output value of Q is 1, C visits all of $v$'s children nodes. Otherwise, C terminates the path at $v$.

*Stage 5: Leaf nodes*

When C and IS reach a leaf node $v$ corresponding to record index $i$ in the search procedure of Stage 4, IS selects keys $(\textsc{out}_0, \textsc{out}_1)$, encrypts $\tilde{R}_i$ as $\mathbf{E}(\tilde{R}_i) = \mathtt{Enc}_{\textsc{out}_1}(\tilde{R}_i)$, and sends to C the tuple $(\psi(i), r_i, \mathbf{E}(\tilde{R}_i))$. In addition, IS sends to C the garbled $\mathtt{AND}$ table:

$$[\mathtt{Enc}_{\textsc{out}_i^{UQ}}(\mathtt{Enc}_{\textsc{out}_j^{PC}}(\textsc{out}_{i \wedge j} || i \wedge j))]$$

IS constructed $UQ$ in Stage 2 and C has already obtained the keys corresponding to its gate value inputs via OT. Additionally, both IS and C already have the sets of Bloom filter indices $\mathcal{I}_\alpha$ for each keyword term $\alpha$ in the query. IS has a masked Bloom filter $\tilde{B}_v$. C has a mask $F_k(v)$. Now:

1) IS generates a new garbled $UC$ using fresh key pairs for all wires except the gate value input wires.
2) IS sends to C the keys corresponding to its input bits $\{\tilde{B}_v[i]\}_{i \in \mathcal{I}_\alpha}$ for each $\alpha$.
3) C performs OT with IS to receive the keys corresponding to its input bits $\{F_k(v)[i]\}_{i \in \mathcal{I}_\alpha}$ for each $\alpha$.
4) Finally, C evaluates the garbled $UC$ and obtains an output key $\textsc{out}_u^{UC}$, $u \in \{0, 1\}$.

C has already obtained an output key $\textsc{out}_p^{PC}$, $p \in \{0, 1\}$, from $PC$ in Stage 3. If $p = u = 1$, then C can successfully obtain $\textsc{out}_1$ and decrypt $\mathbf{E}(\tilde{R}_i)$. Otherwise, C cannot feasibly deduce any information about $\tilde{R}_i$ other than its size.

*Stage 6: Record retrieval*

When C reaches a record $R_i$, IS will send it $\psi(i), r_i$, and $\mathbf{E}(\tilde{R}_i)$. C then sends $\psi(i)$ to S, who sends back $s'_\psi(i)$. C obtains $s_i = s'_{\psi(i)} - r_i$. If C successfully decrypted $\mathbf{E}(\tilde{R}_i)$ to obtain $\tilde{R}_i$ in Stage 5, then C now decrypts $\tilde{R}_i$ using $s_i$, and obtains the record $R_i$. Otherwise, C cannot feasibly deduce any information about $R_i$ other than its size.

## V. Security and Privacy Analysis

Privacy and security in the Blind Seer system was previously achieved with respect to semi-honest static adversaries [28]. An ideal functionality $\mathcal{F}_{db}$ was defined, and included a leakage profile describing the precise information that is leaked to each of the parties C, S, and IS. A standard simulation argument in the semi-honest static adversary model attests that Blind Seer securely realizes $\mathcal{F}_{db}$.

The main contribution of the present work is a mechanism that we claim strengthens the security of Blind Seer against a malicious client adversary in numerous respects including privacy, data protection, and access-control. The preceding sections presented the security benefits of our new mechanism in conceptual terms, and with a focus on how they address the vulnerabilities of Blind Seer's previous design. The goal of this section is to characterize these security properties more precisely in formal definitions, and to prove that our new Blind Seer protocol realizes these properties.

**Security properties.** We distinguish and analyze four properties of the system.

*Query indistinguishability* captures the inability of the server (or index server) to distinguish between queries that the client may submit. Blind Seer does not achieve perfect query indistinguishability. The query security of the original Blind Seer was analyzed using the simulation paradigm. It was shown that the client's queries reveal nothing to the semi-honest server and index server beyond a specified *leakage profile*. Essentially, the leakage profile for the server included record retrieval patterns, and the leakage profile for the index server included the search tree traversal pattern and deterministic hashes of the query keywords (i.e. BF indices). In terms of indistinguishability, this simulation security implies that the server (resp. index server) cannot distinguish between two queries that have the same (or indistinguishable) leakage profiles. Simulation security is actually a stronger notion than indistinguishability. The new system mostly preserves the privacy of the client's query that the original Blind Seer offered, but with some additions to the query leakage profiles. We do not include the full proof of simulation security in this paper, but we will describe the new query leakage profiles.

*Policy compliance indistinguishability* expresses that compliant queries with zero results and non-compliant queries are indistinguishable provided that they have identical index traversal patterns. *Policy soundness* is the extent to which the system prevents the release of information on non-compliant queries. *Query soundness* is the extent to which the system prevents the release of information from records that do not satisfy the query. We show that non-compliant queries reveal no information about the DB *payload*—the non-indexed primary data contained in the records of the DB. Similarly, we show that queries reveal no information from the payload of records outside their result set.

The strongest notions of policy and query soundness would require that the amount of information revealed is negligible. However, as previously discussed, Blind Seer's search mechanism may reveal some partial information about the indexed DB data, especially to a malicious client. Thus, the client may learn some meta information even on records that are not ultimately returned. The only way to prevent *all* leakage on non-compliant queries is to prevent the client from evaluating non-compliant queries on any part of the DB index. This would create an inherent asymmetry in the processing of compliant and non-compliant queries, detracting from policy privacy. We made a conscious design decision to compromise full policy security for policy privacy.

**Multiple clients.** Our analysis assumes a single client. However, the proofs naturally extend to multiple client parties as long as there is no collusion among the parties. If there is collusion between parties, then the proofs still apply to the combination of those parties as a single entity. We note that collusion between a client and the index server $\mathsf{IS}$ would compromise security for everyone.

**Indistinguishability vs. simulation.** Our analysis will use indistinguishability games rather than the simulation paradigm for a number of reasons. First, some of the primitives we implemented are not simulatable in malicious and/or concurrent settings, specifically Naor-Pinkas OT, which we use during the multi-threaded traversal of the Bloom filter. This technical issue could be resolved by implementing a UC-secure OT primitive such as PVW [30] instead of Naor-Pinkas [25] and using simulatable UC-secure OT extension protocols[1], or the server $\mathsf{S}$ could actually distribute random OTs during the preprocessing phase. Second, it is unclear how much we would gain from a simulation analysis. In fact, there are subtleties in understanding what is and is not protected from a malicious client. We favor a clear and direct approach that characterizes the various security properties and tradeoffs. A definition with a single ideal functionality and a leakage oracle is potentially more opaque.

**OT Extension Security.** As previously mentioned, our OT extension protocol does not satisfy a simulation-based definition of security in our setting, which involves both concurrency and a malicious receiver party (the client).

Nonetheless, we can prove that it is sufficiently secure for the malicious-client security properties that we ultimately want to prove. Informally, the property we need is that no malicious receiver can feasibly output both messages in any individual $\binom{1}{2}$-OT when the messages are independent and random.

Let $\text{OT-EXT}_m^{p(k)}$ denote an instance of our OT extension protocol producing $p(k)$ preprocessed $\binom{1}{2}$-OTs of length $m$ strings, where $k$ is a security parameter and $p$ is a polynomial. $\text{OT-EXT}_m^{p(k)}$ outputs $\{(e_i^0, e_i^1), (e_i^{r_i})\}_{i=0}^{p(k)}$, where $(e_i^0, e_i^1)$ is $\texttt{Sender}$'s output, $e_i^{r_i}$ is $\texttt{Receiver}$'s output, All $(e_i^0, e_i^1)$ are independent and random, and $\mathbf{r} \leftarrow \{0,1\}^{p(k)}$ is uniformly distributed. We prove the following Lemma in Appendix B.

**Lemma 1.** *No adversary corrupting* $\texttt{Receiver}$ *in a polynomial number of concurrent and independent $\text{OT-EXT}_m^{p(k)}$ sessions can output both $e_i^0$ and $e_i^1$ from any individual session with probability greater than* $\mathsf{negl}(k)$.

In what follows we will refer to the new system protocol as $\mathsf{mBS}$, for *malicious Blind Seer*.

*Client query privacy*

We describe the client query leakage profiles for the server $\mathsf{S}$, the index server $\mathsf{IS}$, and the query checker $\mathsf{QC}$. Only the leakage to $\mathsf{IS}$ and $\mathsf{QC}$ has changed from the original Blind Seer design. The new leakage to $\mathsf{IS}$ amounts to patterns in the keyword fields (e.g., columns) used in the query, and the new leakage to $\mathsf{QC}$ is the query topology. We note that it is straightforward to update the formal analysis of the original Blind Seer design to incorporate these new leakage profiles, but we will not include here the updated analysis.

**Leakage to $\mathsf{S}$ in each query.** Let $R_1, ..., R_n$ denote the records of the database and let $((i_1, R_{i_1}), \ldots, (i_j, R_{i_j}))$ denote the query results. Let $\pi : [n] \to [n]$ denote a random permutation (unknown to $\mathsf{S}$, but fixed for all queries). The leakage to the server is $(\pi(i_1), \pi(i_2), \ldots, \pi(i_j))$.

**Leakage to $\mathsf{IS}$ in each query.** The leakage to $\mathsf{IS}$ includes the BF-search tree traversal paths, the topology of the query $\texttt{topo}(Q)$, and the pairs of hashes $H_{k_c}(\texttt{field})||H_{k_c}(\texttt{field}:\texttt{value})$ for each of the client's keywords $\texttt{field}:\texttt{value}$ included in the query. This leakage reveals to $\mathsf{IS}$ when two different keywords in the query share

[1]There are various OT extension protocols offering full simulation security in the malicious adversary model that also have constant amortized cost [11], [27], but these protocols still have constant factor overheads that are relatively expensive.

the same field. (In contrast, the previous design only leaked hashes of the keywords in the query, and so IS only learned when full keywords repeated).

These leakage profiles imply that S cannot distinguish any two queries that access the same number of records and IS cannot distinguish any two queries that have the same number of repeated keywords/fields, indistinguishable traversal paths, and identical topologies. However, both S and IS may respectively accumulate record retrieval and query pattern information over many queries.

*Policy and query soundness*

We first define the notion of *payload indistinguishability*. Assume that each record $R$ of the DB has a *payload* $\mathcal{L}$, the main data associated with the record, and separate *metadata* $\mathcal{M}$, or the keywords that index the data. We write $R = (\mathcal{L}, \mathcal{M})$. Queries are evaluated on the metadata, so for any boolean query $\mathbf{q}$ we can write $\mathbf{q}(\mathcal{M}) \in \{0,1\}$.

**Definition 1.** *Let $\Sigma(D,\lambda)$ denote a DBMS mechanism executed on input database $D$ with security parameter $\lambda$. We define the following game $\mathsf{Game}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda)$ played with an adversary $\mathcal{A}$.*

$\mathsf{Game}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda)$*:*
– *$\mathcal{A}$ chooses $D_0 = (R_0^1, ..., R_0^n)$ and $D_1 = (R_1^1, ..., R_1^n)$ where $R_b^i = (\mathcal{L}_b^i, \mathcal{M}_b^i)$ and $\mathcal{M}_0^i = \mathcal{M}_1^i$ $\forall i$.*
– *Sample $b \in \{0,1\}$ uniformly at random.*
– *The protocol $\Sigma(D_b,\lambda)$ is executed with $\mathcal{A}$ in the querier's role. $\mathcal{A}$ outputs a decision bit $b'$.*
– *If $b' = b$, output 1. If $b' \neq b$, output 0.*

*Define* $\mathbf{Adv}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda) = |Pr[\mathsf{Game}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda) = 1] - \frac{1}{2}|$. *If* $\mathbf{Adv}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda) < \mathsf{negl}(\lambda)$ *for any poly time adversary $\mathcal{A}$, then the mechanism $\Sigma$ is **payload indistinguishable**.*

*More generally, we define $f$-**payload indistinguishability** by modifying the game so that $\mathcal{L}_0^i = \mathcal{L}_1^i$ for all i where $f(\mathcal{M}_b^i) = 0$, i.e. the payloads are only indistinguishable on records for which $f(\mathcal{M}_b^i) = 1$.*

**Theorem 1.** *An execution of* mBS *on any policy* $\mathbf{p}$ *and query* $\mathbf{q}$ *such that* $\mathbf{p}(\mathbf{q}) = 0$ *is payload indistinguishable.*

*Proof.* We prove the claim by reduction to the semantically secure encryption scheme $\Pi = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$ used by mBS.

Let us first unfold the meaning of executing mBS on policy $\mathbf{p}$ and query $\mathbf{q}$. The policy $\mathbf{p}$ is defined by an input to the semi-honest party QC. However, since C is now a malicious adversary $\mathcal{A}$, the query $\mathbf{q}$ defined by the actions of $\mathcal{A}$ may be unrelated to anything on C's initial input tape. $\mathcal{A}$ commits to the query circuit $Q$ that is ultimately evaluated on the DB records when it sends the circuit topology, sends the keyword and field hashes, and receives keys in $\{0,1\}^k$ representing the gate values of $Q$ using OTs generated by the subprotocol OT-EXT$_k^{p(k)}$. If $\mathcal{A}$ submits ill-formatted messages, then the protocol is aborted, and the query is considered empty. Thus, $Q$ is uniquely determined unless $\mathcal{A}$ is able to obtain more than one valid key for each OT, the probability of which is negligible in $k$ by Lemma 1.

The garbled policy circuit $PC$ and evaluation keys in $\{0,1\}^k$ that $\mathcal{A}$ receives are derived from the same information that determines $Q$. As long as $Q$ is uniquely determined, the following hold except with probability negligible in $k$: $\mathcal{A}$ only obtains the keys that allow it to compute the output policy key $\mathrm{OUT}_{\mathbf{p(Q)}}^{PC}$, and the security of Yao GC evaluation guarantees that $\mathcal{A}$ cannot obtain the key $\mathrm{OUT}_{1-\mathbf{p(Q)}}^{PC}$.

Any record payload $\mathcal{L}$ that $\mathcal{A}$ receives is encrypted using enc with the key $\mathrm{OUT}_1^{PC} = \mathsf{gen}(1^k)$. We have seen that $\mathcal{A}$ cannot compute $\mathrm{OUT}_1^{PC}$ when $\mathbf{p}(Q) = 0$ except with some negligible probability $\epsilon(k)$. Thus, it is easy to see that $\mathbf{Adv}_{\Sigma}^{\texttt{PAY-IND}}(\mathcal{A},\lambda) \leq \mathbf{Adv}_{\Pi}^{\texttt{IND}}(\mathcal{A},\lambda) + \epsilon(k) < \mathsf{negl}(k)$ by semantic security of $\Pi$ (cf. Appendix A). □

**Theorem 2.** *For any query* $\mathbf{q}$, *let* $\bar{\mathbf{q}}$ *denote its negation so that* $\bar{\mathbf{q}}(D)$ *is the set of records that fail* $\mathbf{q}$. *An execution of* mBS *on any query* $\mathbf{q}$ *is* $\bar{\mathbf{q}}$*-payload indistinguishable.*

*Proof.* For any record $R = (\mathcal{M}, \mathcal{L})$ that $\mathcal{A}$ receives, $\mathcal{L}$ is encrypted using enc with a key $\mathrm{OUT}_1^{UQ}$ that is output by the garbled $UQ$ circuit evaluated on $M$. As noted in the analysis of policy soundness (Theorem 1), the query $\mathbf{q}$ is uniquely determined by $\mathcal{A}$'s messages to IS except with probability negligible in $k$, the security parameter of $\Pi = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$.

We show that if $\mathbf{q}(\mathcal{M}) = 0$, i.e. $\bar{\mathbf{q}}(\mathcal{M}) = 1$, then $\mathcal{A}$ cannot obtain $\mathrm{OUT}_1^{UQ}$ with probability greater than $\mathsf{negl}(\lambda)$ where $\lambda$ is the minimum of $k$ and the Bloom filter FPR used at the leaf nodes of the tree index. In the $\bar{\mathbf{q}}$-payload indistinguishability game, the payloads $\mathcal{L}_0^i$ and $\mathcal{L}_1^i$ of records $R_0^i$ and $R_1^i$ are identical unless $\bar{\mathbf{q}}(\mathcal{M}_0^i) = \bar{\mathbf{q}}(\mathcal{M}_1^i) = 1$. Once we establish that $\mathcal{A}$'s probability of obtaining $\mathrm{OUT}_1^{UC}$ when $\bar{\mathbf{q}}(\mathcal{M}) = 0$ is negligible in $\lambda$, $\bar{\mathbf{q}}$-payload indistinguishability reduces to the semantic security of $\Pi$, as in Theorem 1.

Consider $\mathcal{A}$'s evaluation of $UQ$ on record $R$. As with the policy, $\mathcal{A}$ can only obtain one valid set of input keys, and can only compute one valid output key, except with probability $\mathsf{negl}(k)$. However, unlike the policy evaluation keys, the keys that $\mathcal{A}$ obtains to evaluate $UQ$ are not solely determined from the information that defines the query circuit $Q$. The reason is that $\mathcal{A}$ has some control over the DB inputs to $UQ$. Recall that for each BF node $B_v$, the client C holds a mask string $\mathsf{m}_v = F_k(v)$ and IS holds $\tilde{B}_v = B_v \oplus \mathsf{m}_v$. The circuit $UQ$ computes XORs of bits taken from $\mathtt{m}$ and $B_v$. Nothing prevents $\mathcal{A}$ from flipping bits in $\mathsf{m}_v$ when evaluating $UQ$.

We investigate $\mathcal{A}$'s success probability in flipping the output of $UQ$ from 0 to 1. Because $Q$ is monotone over its keyword predicates, $\mathcal{A}$ must succeed in flipping the output of at least one keyword predicate $P_\alpha$ from 0 to 1 in order to succeed. Suppose $\mathcal{A}$ receives $\eta$ indices $\alpha_1, ..., \alpha_\eta$ for a keyword $\alpha$ that has *not* been inserted into the filter $B_v$, i.e. $P_\alpha(B_v) = 0$. There is a unique vector of bits $\mathbf{z}_\alpha = (z_{\alpha_1}, ..., z_{\alpha_\eta})$ such that $\tilde{B}_v[\alpha_i] \oplus z_v[\alpha_i] = 1$ for each $i$. $\mathcal{A}$ must guess $\mathbf{z}_\alpha$ to succeed.

$\mathbf{z}_\alpha$ is actually a random variable over the randomness in the initialization of $B_v$. The bits of $B_v$ were set via insertions of keywords (distinct from $\alpha$) using $\mathcal{H}_{k_s}$, where $k_s$ is secret from $\mathcal{A}$, and $\mathcal{H}$ is a family of random oracles by assumption. Therefore, $\mathbf{z}_\alpha$ is distributed independently from $\mathcal{A}$'s view of the protocol. Recall that the BF parameters are set so that the FPR is $2^{-\eta}$. Together with the simplifying assumption that the bits $z_{\alpha_i}$ are $\eta$-wise independent, the FPR implies that $Pr[z_{\alpha_i} = 1] = 1/2$ for each $i$, and that the min entropy $\mathbf{z}_\alpha$ is $2^{-\eta}$.[2] □

*Policy compliance indistinguishability*

There are three possibilities when a query returns no results: the query was noncompliant, the query actually had an empty result set, or both. The client cannot ever tell with certainty which one of these is true. However, the search pattern may give the client heuristic reasons to believe one possibility over the other. For instance, if the policy is evaluated at the leaves and a query traversal reaches many leaf nodes in the DB index before failing on all, the client may reasonably infer that the query is failing the policy.

Thus, a definition that requires complete indistinguishability regardless of the database and query would be too strong for mBS to satisfy. Instead, we use a definition that incorporates a database equivalence relation $\mathbb{E}_\mathbf{q}$ parametrized by the query $\mathbf{q}$. Similar to a leakage profile, $\mathbb{E}_\mathbf{q}$ factors out instances where the adversary may defeat the indistinguishability game, for reasons either related or unrelated to distinguishing query non-compliance from empty results.

**Definition 2.** *We define the policy compliance indistinguishability game* $\mathsf{Game}_\Sigma^{\texttt{PC-IND}}(\mathcal{A}, \mathbb{E}_\mathbf{q}, \lambda)$ *as follows.*

$\mathsf{Game}_\Sigma^{\texttt{PC-IND}}(\mathcal{A}, \mathbb{E}_\mathbf{q}, \lambda)$*:*

– $\mathcal{A}$ *chooses a query* $\mathbf{q}$*, databases* $(D_0, D_1) \in \mathbb{E}_\mathbf{q}$*, and policies* $\mathbf{p_0}$*,* $\mathbf{p_1}$ *such that* $\mathbf{p_0}(\mathbf{q}) = 1$*,* $\mathbf{q}(D_0) = 0$*, and* $\mathbf{p_1}(\mathbf{q}) = 0$*.*

– *A bit* $b$ *is sampled uniformly at random.*

– *The protocol* $\Sigma(D_b, \mathbf{q}, \mathbf{p_b}, \lambda)$ *is executed with* $\mathcal{A}$ *in the querier's role.* $\mathcal{A}$ *outputs a decision bit* $b'$*.*

– *If* $b' = b$*, output 1. If* $b' \neq b$*, output* 0*.*

$\mathbf{Adv}_\Sigma^{\texttt{PC-IND}}(\mathcal{A}, \mathbb{E}_\mathbf{q}, \lambda) = |Pr[\mathsf{Game}_\Sigma^{\texttt{PC-IND}}(\mathcal{A}, \mathbb{E}_\mathbf{q}, \lambda) = 1] - \frac{1}{2}|$. *If* $\mathbf{Adv}_\Sigma^{\texttt{PC-IND}}(\mathcal{A}, \mathbb{E}_\mathbf{q}, \lambda) < \mathsf{negl}(\lambda)$ *for any poly time adversary* $\mathcal{A}$*, then* $\Sigma$ *satisfies* ***policy compliance indistinguishability*** *with respect to* $\mathbb{E}_\mathbf{q}$*.*

We formally define an equivalence relation $\equiv_\mathbf{q}$ for mBS. First, the relation must express traversal pattern equivalence. Otherwise, $\mathcal{A}$ could distinguish executions of $\mathbf{q}$ on $D_0$ and $D_1$ whenever the traversal patterns differ. In certain instances the traversal pattern may actually reveal policy failure (e.g., a search for a single name that returns true at the root and fails on every single record must be non-compliant). This instance is eliminated from the game by requiring that the traversal patterns are identical for both the compliant and non-compliant scenarios. Formally, let $I_\pi(\cdot)$ denote the DB index construction function of mBS using the record permutation $\pi$. Let $TP(\mathbf{q}, I_\pi, D)$ denote the distribution of traversal patterns induced by $\mathbf{q}$ on $I_\pi(D)$ for randomly sampled $\pi$. If $D_0 \equiv_\mathbf{q} D_1$, then $TP(\mathbf{q}, I_\pi, D_0) \approx TP(\mathbf{q}, I_\pi, D_1)$. Further, since the client generates the query circuit for the internal nodes, a malicious client can compute an arbitrary single bit of the BF inputs at each node. Thus, $\equiv_\mathbf{q}$ must express the stronger condition that corresponding Bloom filter nodes in the two databases must be identical on all indices the query touches. This does not include the last layer, where policy and $UQ$ circuits are evaluated.

**Theorem 3.** mBS *satisfies policy compliance indistinguishability with respect to* $\equiv_\mathbf{q}$*.*

*Proof.* Given the strict definition of $\equiv_\mathbf{q}$, $\mathcal{A}$'s views of the executions $\mathsf{mBS}(D_0, \mathbf{q}, \mathbf{p_0}, \lambda)$ and $\mathsf{mBS}(D_0, \mathbf{q}, \mathbf{p_0}, \lambda)$ are identical up until the failure nodes where $UQ \wedge PC$ is evaluated with Yao GC. If $\mathcal{A}$ can distinguish the intermediary outputs of $UQ$ and $PC$, it would contradict the security of Yao. □

## VI. Implementation and Evaluation

We implemented a prototype of our design in C++. In this section, we describe some interesting choices we made during the development of the prototype, and show experimental results.

**OT pool.** One important component of our system is the OT pool. This pool contains preprocessed Oblivious Transfers that are used in garbled circuit evaluations. The OT pool is filled regularly using the OT extension protocol of [13], [26]. We use the Naor-Pinkas [25] OT protocol as a base for OT extension.

**Parallelism within queries.** Our most important efficiency improvement comes from query parallelization. While the original Blind Seer implementation supported parallelization between queries to improve throughput, our new system supports multi-threaded evaluation of the tree structures within a single query to improve latency. Instead of having one global OT pool for all threads, each thread has its own OT pool. This significantly decreases the standby time of filling a single OT pool, as well as bypassing synchronization issues. We used the Intel Threading Building Block library to implement most of the parallelization used in the system.

Figure 3 shows query latency time plotted against the number of threads for the query

```
first:DIANE AND last:CASTRO
```

This query traverses a large fraction of the DB index, and hence, clearly illustrates the benefits from parallelization. We see full utilization and improvement all the way up to 15 threads.

**Cryptographic Tools.** Our system requires pseudorandom bits, used primarily for garbling circuits. We avoid expensive system calls to /dev/urandom by implementing a PRG using AES as a building block. All AES operations were performed

[2]It is inaccurate to assume that all BF bits are mutually independent. However, $\eta$-wise independence of the $\{z_{\alpha_i}\}_{i=1}^{\eta}$ is easily obtained. With an FPR of $2^{-\eta}$, the fraction of 1s in the BF cannot be more than 1/2 after all word insertions. If smaller, we can randomly set BF bits until exactly 1/2 are 1. The distribution of the final subset of 1s is uniform, independent of $\mathcal{H}(\alpha)$. Thus, the $\{z_{\alpha_i}\}_{i=1}^{\eta}$ are independent if $\eta < \ell/2$, where $\ell$ is the BF length.

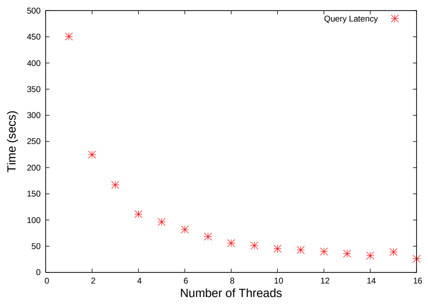

Figure 3. Query Latency versus number of threads on a $10^7$ record database. Run on a Boolean query with individually frequent terms but with sparse aggregate results.

using 128-bit key length. We used SHA256 for hashing, and implemented the keyed hash for keyword ingestion as an HMAC using SHA256 as the underlying hash function. Since OpenSSL uses the AES-NI instruction, we used this library to implement our basic cryptographic primitives. Our system also depends on operation on a group to which DDH intractability applies. We used the standard quadratic residues subgroup of $\mathbb{Z}_p$, where $p$ is a 2048-bit strong prime. The group operations were implemented using the GNU Multiple Precision library.

**Tree Traversal.** During tree traversal, each internal node evaluation requires 4 rounds of communication. The client first submits the node identifier to be evaluated and the garbled circuit to be evaluated, then OT is performed (2 rounds), and finally the index server sends back the output key. The roles are reversed at the leaf nodes. Since these rounds involve small packets, network latency becomes the bottleneck. In order to reduce standby time, we batched the evaluation of all sibling nodes. This reduced the number of rounds by 10 rounds. This batching technique is mostly helpful for queries with low branching traversal patterns, e.g., a single root to record path in the tree index.

## A. Experimental Results

Our system was tested in a similar way to Blind Seer. A synthetic *100-million* record database mimicking the US census data was constructed. Each database record contained global unique ID, personal information taken from randomized census, text fields for free-text search, XML data, and a payload of 100KB of random bytes used to simulate the cost of transferring data in real-life practical applications.

Our system was compared against MySQL (having separate indexes for each field) using a special-purpose testbed implemented by Lincoln Labs specifically for this purpose. Each party ran on a separate machine equipped with two Intel Xeon X5650 processors of 2.66Ghz and 12M cache, 96GB RAM at 1066 MHz, and Broadcom 1GB Ethernet NICS with TOE each. Index Server and Server each had a 20TB raid array attached also. All machines ran 64-bit Ubuntu 12.04LTS as base OS.

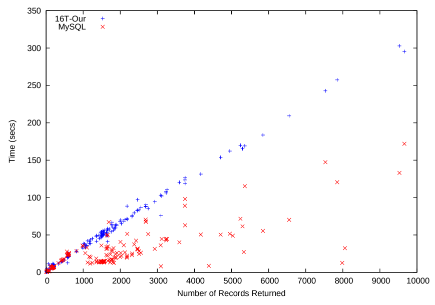

Figure 4. Single-term SELECT-id performance against number of records returned for our system and MySQL.

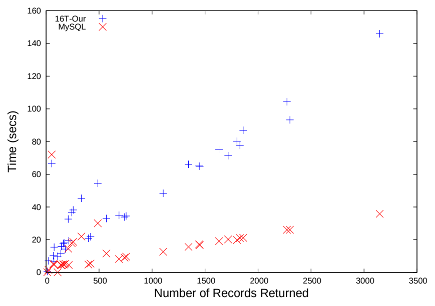

Figure 5. 3-terms conjunction SELECT-id queries performance against number of records returned for our system and MySQL.

We show next the efficiency of our system (running using 16 threads on full 100-million record DB) in terms of query latency time for a number of representative queries: single-term and size-3 conjunctive and disjunctive SELECT-id queries. We also show that when using SELECT-* queries on 100KB records, and thus incorporating a payload with our interactions, the associated overhead which is constant for all systems begins to dominate the costs of our system.

**Boolean SELECT-id Queries.** We compare the performance of our system against MySQL. Figures Fig. 4, Fig. 5, and Fig. 6 show the query latency for single-term, 3-term conjunctions (with two low-entropy terms and one medium-entropy term), and 3-term disjunctions as SELECT-id queries plotted against the number of records satisfying them. While original Blind Seer was 15 times slower than MySQL [28], our implementation manages to be only 2-3 times slower than MySQL for the case of single terms queries and 3-6 times slower for conjunctive and disjunctive queries on 3 terms. This saving is due to parallelization and low overhead of our new construction. Note that these queries are SELECT-id, and thus delivering a small payload. The relative overhead of our system would decrease with larger payloads since this cost is constant for both systems.

**Single-term SELECT-* Queries.** We measured query latency of our system and MySQL for queries returning 100KB

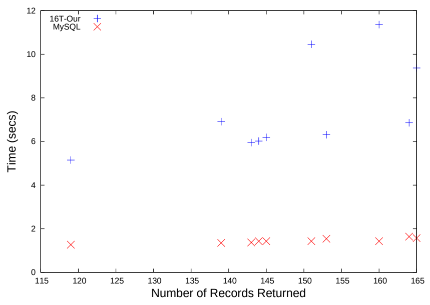

Figure 6. 3-terms disjunction SELECT-id queries performance against number of records returned for our system and MySQL.

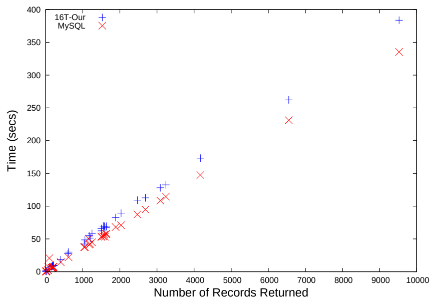

Figure 7. Single-term SELECT-* performance against number of results returned for our system and MySQL.

records. Our system, as well as original Blind Seer, performs better (compared to MySQL) when the records retrieved are bigger. In this case, the standby time required to submit records' data dominate the network usage. We can observe from the results shown in figure Fig. 7 that our system is only 10% slower than MySQL in this setting.

**Overhead of malicious-client security.** As a validation of the costs of our malicious-client algorithms, we also compare the performance of our malicious-client secure Blind Seer to that of the original Blind Seer. To eliminate extraneous implementation performance variables, we compare the two designs by running our current implementation of Blind Seer with and without the design changes introduced in this work. In particular, the design differences potentially affecting performance include two additional hash function calls per keyword, the new semi-private SFE using a universal circuit, an additional Oblivious Transfer per gate of the query circuit, and one additional symmetric encryption and decryption per record returned to C. The experiment consisted of running single-threaded SELECT-id queries for single-term, 3-term boolean queries containing a conjunction and a disjunction, and range queries (which are each translated to a 5-7 term disjunction [28]) over a *1-million* record database. The results are shown in figures 8, 9, 10. As expected, the design changes for achieving malicious-client security incur no significant overhead. The costs of the design changes for malicious-client security are proportional to the query size and number of leaf nodes reached in the Bloom filter tree index traversal, and not the total number of tree nodes processed. Thus, the performance gap will be even smaller on large databases when the number of leaf nodes reached is small compared to the number of internal nodes processed, which we expect to often be the case in practice.

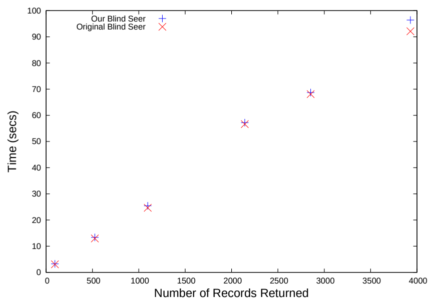

Figure 8. Performance of our prototype against original Blind Seer for single-term queries.

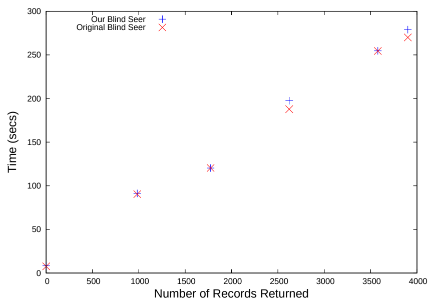

Figure 9. Performance of our prototype against original Blind Seer for 3-term boolean queries.

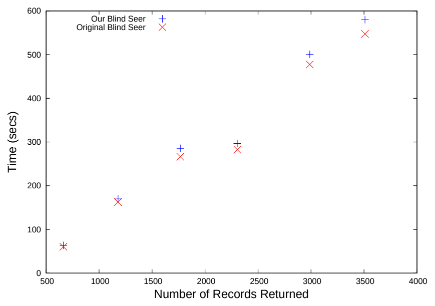

Figure 10. Performance of our prototype against original Blind Seer for range queries.

## VII. Related Work

Full privacy in a DBMS is possible to achieve using cryptographic tools such as SPIR [4], [6], FHE [5], ORAM [9], [10], [23], and MPC [8], [21], [32], [33]. However, all such solutions are prohibitively expensive to be used in many scenarios of interest where the amount data involved is large.

More efficient private search solutions have been proposed for more restricted scenarios. Allowing leakage of client access patterns is a common relaxation, since it is required in order to avoid the polylogarithmic overhead of ORAMs. Until recently, these solutions only offered very limited search capabilities. For a survey of different systems, we refer the reader to the related work sections of [3], [14], [28].

The Blind Seer [28] and OSPIR-OXT [14] projects are two recent systems aiming towards a favorable trade off between performance, functionality, security and privacy. The core philosophy of these system is to allow a *reasonable* and limited privacy leakage in order to achieve a performance and rich functionality close to an industry-standard baseline.

These two solutions are very different in terms of design and offer different tradeoffs between privacy, functionality and performance. At a high level, Blind Seer's core data structure is a search tree index that is traversed node by node, requiring high interaction between the client and server. OSPIR-OXT's index is an inverted index that doesn't require much interaction, and as a result, offers better performance than Blind Seer in this respect.

In terms of functionality, both systems offer rich set of queries. OSPIR-OXT functionality is a bit poorer since queries must be of the form $t_1$ AND $\phi(t_2, .., t_n)$ (where $t_i$ are single terms), with performance that depends on the number of records satisfying the first term $t_1$ (so to avoid linear time computation, the client should identify a representation of the query in this form, where $t_1$ does not have too many matches). In many applications, however, relevant queries are conjunctions, and it is reasonable to assume that the client has some a priori information of the database allowing it to guess which term is of low frequency. Blind Seer, on the other hand, supports arbitrary Boolean queries in time that depends on the number of records satisfying the best (minimal) term, even if that term is not first, not known by the client, or not even appearing explicitly in the query representation. A big strength of OSPIR-OXT is its support for multiple clients. Blind Seer can support non-colluding clients, but achieving more robust support for multiple clients is still an open problem. In particular, IS colluding with a client could compromise the privacy of other clients accessing the same data.

With respect to privacy, OSPIR-OXT reveals to the index server the number of records satisfying $t_1$ and number of records satisfying each subterm $t_1$ AND $t_i$, while Blind Seer hides this information completely. Both systems leak index traversal patterns. Both rely on semi-honest servers for security, but (until the current work) only OSPIR-OXT was secure against a malicious client.

## VIII. Acknowledgements

This work was supported in part by the Intelligence Advanced Research Project Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract Number D11PC20194. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. 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. Fernando Krell was additionally supported in part by BECAS CHILE, CONICYT, Gobierno de Chile. We thank MIT Lincoln Labs researchers for supporting this program from the beginning to the end and facilitating extensive testing of our code. We thank George Argyros for helpful discussions and for contributions to the implementation of this project. Finally, we thank our colleagues from other IARPA SPAR teams for great collaboration and exchange of ideas.

## References

[1] D. Beaver. Precomputing oblivious transfer. In D. Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, Aug. 1995.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, 2013.

[4] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR-CS0917, Dept. of Computer Science, Technion, 1997.

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

[6] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences*, 60(3):592–629, 2000.

[7] O. Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

[8] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43:431–473, 1996.

[10] 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 *ACM CCS 12*, pages 513–524, 2012.

[11] D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-combiners via secure computation. In R. Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 393–411. Springer, Mar. 2008.

[12] Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 458–475, 2014.

[13] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Aug. 2003.

[14] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 875–888, New York, NY, USA, 2013. ACM.

[15] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[16] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[17] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[18] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In G. Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.

[19] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, Aug. 2013.

[20] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In M. Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, May 2007.

[21] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, Apr. 2009.

[22] Y. Lindell and B. Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 476–494, 2014.

[23] S. Lu and R. Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, pages 377–396, 2013.

[24] P. Mohassel and M. Franklin. Efficient polynomial operations in the shared-coefficients setting. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 44–57. Springer, Apr. 2006.

[25] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In S. R. Kosaraju, editor, *12th SODA*, pages 448–457. ACM-SIAM, Jan. 2001.

[26] J. B. Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive*, 2007:215, 2007.

[27] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Aug. 2012.

[28] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, S. Bellovin, and A. Keromytis. Blind seer: A private scalable DBMS. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2014.

[29] A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS 09*, volume 5536 of *LNCS*, pages 89–106. Springer, June 2009.

[30] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In D. Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Aug. 2008.

[31] L. G. Valiant. Universal circuits (preliminary report). In A. K. Chandra, D. Wotschke, E. P. Friedman, and M. A. Harrison, editors, *STOC*, pages 196–203. ACM, 1976.

[32] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, Nov. 1982.

[33] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, Oct. 1986.

## Appendix

### A. *Preliminary tools*

**Bloom filters.** A Bloom filter [2] is a well-known data structure that efficiently stores arbitrary items for membership checking The filter's underlying data structure is an array of $\ell$ bits that are assigned using $k$ hash functions $h_1, ..., h_k$ with range $\{1, ..., \ell\}$. If $B$ denotes a Bloom filter, then we will use the notation $B[i]$ to denote the $i$th bit of $B$.

The operations on a Bloom filter are as follows. Initialization: all $\ell$ bits of $B$ are set to 0. Insertion: to insert keyword $\alpha$, for $1 \leq i \leq k$ set $B[h_i(\alpha)] = 1$. Lookup: to see if $B$ contains a keyword $\alpha$, check that $B[h_i(\alpha)] = 1$ for all $1 \leq i \leq k$.

A Bloom filter guarantees no false negatives and allows a tunable rate of false positives:

$$FPR = \left(1 - \left(1 - \frac{1}{\ell}\right)^{kt}\right)^k \approx \left(1 - e^{-\frac{kt}{\ell}}\right)^k,$$

where $t$ is the number of search keywords inserted into the filter. In our system, we set $k = 20$ and $\ell = 28.86t$, which gives $FPR = 2^{-20} \approx 10^{-6}$.

**Semantic security.** *Semantic security* and *ciphertext indistinguishability* are equivalent concepts for symmetric encryption [7], [15]. Let $\Pi = (\mathsf{gen}, \mathsf{enc}, \mathsf{dec})$ denote a symmetric encryption scheme. The game $\mathsf{Game}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \lambda)$ is played with an "eavesdropping" adversary $\mathcal{A}$ as follows. $\mathcal{A}$ chooses equal length messages $m_0, m_1$. The game derives $sk \leftarrow \mathsf{gen}(\lambda)$, samples $b \leftarrow \{0, 1\}$, and sends $\mathcal{A}$ the ciphertext $\mathsf{enc}_{sk}(m_b)$. $\mathcal{A}$ outputs a decision bit $b'$. If $b' = b$, the game outputs 1, and otherwise 0. Let $\mathbf{Adv}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \lambda) = |Prob[\mathsf{Game}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \lambda) = 1] - \frac{1}{2}|$. $\Pi$ is semantically secure if and only if $\mathbf{Adv}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \lambda) < \mathsf{negl}(\lambda)$.

Semantic security for a public key scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is defined similarly, only $\mathcal{A}$ is given the public key $pk$. The resulting definition is equivalent to *indistinguishability under chosen-plaintext attack* (IND-CPA).

**Yao garbled circuits (GC).** Yao's GC [33] is a protocol for two-party secure computation. One party generates a "garbled circuit" on which a second party can obliviously evaluate hidden inputs.

The garbling procedure is as follows. Let $C$ be a Boolean circuit. Each $i$th wire of $C$ is given a pair of random keys $w_0^i, w_1^i$, where key $w_0^i$ corresponds to the value '0', and $w_1^i$ corresponds to the value '1'. For each gate $g$ with input wires labeled $i, j$ and output wire $k$, the corresponding garbled gate $\tilde{g}$ consists of four ciphertexts from which the evaluator can uniquely recover the output key $w_{g(b_i,b_j)}^k$ given input keys $w_{b_i}^i$ and $w_{b_j}^j$.

When using the *free-XOR technique* of [17], XOR gates can be garbled and evaluated "for free" because they do not require extra communication or cryptographic operations. The technique uses a global random value $\Delta$ and sets all input wire keys so that $w_1^i = w_0^i \oplus \Delta$ for all $i$. The output wire $k$ of an XOR gate with input wires $i, j$ is given keys $w_0^k = w_0^i \oplus w_0^j$ and $w_1^k = w_0^i \oplus w_0^j \oplus \Delta$. Thus, the output wire $w_{b_i \oplus b_j}^k = w_{b_i}^i \oplus w_{b_j}^j$ can be computed arithmetically from the input wires.

**Oblivious Transfer.** A 1-out-2 oblivious transfer ($\binom{1}{2}$-OT) protocol is a two-party functionality in which a sender holds input $(x_0, x_1)$, a receiver holds a choice $r \in \{0, 1\}$, the receiver obtains $x_r$, and neither the sender nor the receiver learn anything else.

*OT preprocessing* is the technique of running random OTs offline, and later using them to efficiently run online OTs on real inputs. *OT extension* is a method that produces $N$ oblivious transfers from an initial pool of only $k << N$ "base" OTs of random $N$-bit strings, where $k$ is a fixed security parameter. The IKNP protocol [13] for semi-honest parties has a constant amortized cost per OT.

## B. Proof of Lemma 1

*No adversary corrupting* `Receiver` *in a polynomial number of concurrent and independent OT-EXT$_m^{p(k)}$ sessions can output both $e_i^0$ and $e_i^1$ from any individual session with probability greater than* $\mathsf{negl}(k)$.

*Proof.* Note that the protocol OT-EXT$_m^{p(k)}$ is a randomized functionality that does not take any external input. Internally, the `Receiver` samples $\mathbf{r} \leftarrow \{0,1\}^{p(k)}$ and $\mathbf{T} \leftarrow \{0,1\}^{k \times k}$, and `Sender` samples $\mathtt{s} \leftarrow \{0,1\}^k$. An adversary $\mathcal{A}$ can therefore simulate any session of OT-EXT$_m^{p(k)}$, producing a view that is statistically indistinguishable from its view of the real session. If $\mathcal{A}$ is active and substitutes arbitrary input $(\mathbf{r}', \mathbf{T}')$ in place of the `Receiver`'s randomly sampled input, it can still produce a statistically indistinguishable view of the session by using the same $\mathbf{r}', \mathbf{T}'$ and randomly sampling $\mathtt{s}$ for `Sender`'s input.[3]

By the hypothesis that the `Sender` behaves independently in all sessions, $\mathcal{A}$ cannot distinguish between an experiment in which it engages with all real sessions versus an experiment where it only engages with *one* real session and simulates all others. Therefore, we restrict our attention to the security of a single session $sid$.

Recall that our OT-EXT$_m^{p(k)}$ is Nielsen's OT extension protocol for malicious receivers [26], using Naor-Pinkas to instantiate the base OT protocol. Nielsen gave a concrete analysis showing that $\mathcal{A}$ cannot output both $e_i^0$ and $e_i^1$ for any $i$ with probability greater than $\mathsf{negl}(k)$ when the base OT is instantiated with an ideal OT box. However, the starting point of Nielsen's analysis only assumes that the output of the base OT is correct, and that $\mathtt{s}$ remains uniformly random in the view of $\mathcal{A}$. $\mathtt{s}$ is the choice vector of the `Sender` acting as receiver in the base OT protocol. Sequential or parallel Naor-Pinkas OT guarantees both correctness and the property that $\mathcal{A}$'s views of the protocol (as sender) executed with different choice vectors of the receiver are statistically indistinguishable [25]. Equivalently, conditioned on $\mathcal{A}$'s view, $\mathtt{s}$ remains statistically indistinguishable from a uniformly random vector. It follows that the adversary cannot gain a non-negligible advantage when the ideal OT box is replaced with Naor-Pinkas in Nielsen's OT extension (as otherwise this protocol could be used to distinguish $\mathtt{s}$ from random). $\square$

[3]Note that this is different than *simulatability*, where the simulator in an ideal world secure execution of the protocol is required to simulate the attacks of any adversary in the real world with the same results. There we consider the views of both parties, or a third environment party observing the system, while here we are only concerned with a single party's view.