Blind Seer: A Scalable Private DBMS

Vasilis Pappas*, Fernando Krell*, Binh Vo*,
Vladimir Kolesnikov†, Tal Malkin*, Seung Geol Choi‡, Wesley George§, Angelos Keromytis*, Steven Bellovin*
* Columbia University, {vpappas,binh,fkrell,smb,angelos,tal}@cs.columbia.edu
†Bell Labs, kolesnikov@research.bell-labs.com
‡US Naval Academy, choi@usna.edu
§University of Toronto, wgeorge@cs.toronto.edu

***Abstract*—*Query privacy* in secure DBMS is an important feature, although rarely formally considered outside the theoretical community. Because of the high overheads of guaranteeing privacy in complex queries, almost all previous works addressing practical applications consider limited queries (e.g., just keyword search), or provide a weak guarantee of privacy.**

In this work, we address a major open problem in private DB: *efficient* sublinear search for *arbitrary* Boolean queries. We consider scalable DBMS with provable security for all parties, including protection of the data from both server (who stores encrypted data) and client (who searches it), as well as *protection of the query*, and access control for the query.

We design, build, and evaluate the performance of a rich DBMS system, suitable for real-world deployment on today medium- to large-scale DBs. On a modern server, we are able to query a formula over 10TB, 100M-record DB, with 70 searchable index terms per DB row, in time comparable to (insecure) MySQL (many practical queries can be privately executed with work 1.2-3 times slower than MySQL, although some queries are costlier).

We support a rich query set, including searching on arbitrary boolean formulas on keywords and ranges, support for stemming, and free keyword searches over text fields.

We identify and permit a reasonable and controlled amount of leakage, proving that no further leakage is possible. In particular, we allow leakage of some search pattern information, but protect the query and data, provide a high level of privacy for individual terms in the executed search formula, and hide the difference between a query that returned no results and a query that returned a very small result set. We also support private and complex access policies, integrated in the search process so that a query with empty result set and a query that fails the policy are hard to tell apart.

I. Introduction

Motivation. Over the last two decades, the amount of data generated, collected, and stored has been steadily increasing. This growth is now reaching dramatic proportions and touching every aspect of our life, including social, political, commercial, scientific, medical, and legal contexts. With the rise in size, potential applications and utility of these data, privacy concerns become more acute. For example, the recent revelation of the U.S. Government's data collection programs reignited the privacy debate.

We address the issue of privacy for database management systems (DBMS), where the privacy of *both the data and the query* must be protected. As an example, consider the scenario where a law enforcement agency needs to search airline manifests for specific persons or patterns. Because of the classified nature of the query (and even of the existence of a matching record), the query cannot be revealed to the DB. With the absence of truly reliable and trusted third parties, today's solution, supported by legislation, is to simply require the manifests and any other permitted data to be furnished to the agency. However, a solution that allows the agency to ask for and receive only the data it is interested in (without revealing its interest), would serve two important goals:

- allay the negative popular sentiment associated with large personal data collection and management which is not publicly accounted.
- enhance agencies' ability to mine data, by obtaining permission to query a richer data set that could not be legally obtained in its entirety.

In particular, we implement external policy enforcement on queries, thus preventing many forms of abuse. Our system allows an independent *oblivious* controller to enforce that metadata queries satisfy the specificity requirement.

Other motivating scenarios are abundant, including private queries over census data, information sharing between law enforcement agencies (especially across jurisdictional and national boundaries) and electronic discovery in lawsuits, where parties have to turn over relevant documents, but don't want to share their entire corpus [33], [43]. Often in these scenarios the (private) query should be answered only if it satisfies a certain (secret) policy. A very recent motivating example [3] involves the intended use of data from automated license plate readers in order to solve crimes, and the concerns over its use for compromising privacy for the innocent.

While achieving full privacy for these scenarios is possible building on cryptographic tools such as SPIR [24], FHE [21], ORAM [27] or multiparty computation (MPC), those solutions either run in polynomial time, or have very expensive basic steps in the sublinear algorithms. For example, when ORAM is used to achieve sublinear secure computation between two parties [29], its basic step involves oblivious PRF evaluation. [29] reports that it takes about 1000 seconds to run a binary search on 2^{20} entries; subsequent works [22], [39] remain too expensive for our setting. On the other hand, for data sets of moderate or large sizes, even linear computation is prohibitive. This motivates the following.

Design goals. Build a secure and usable DBMS system,

with rich functionality, and performance very close to existing insecure implementations, so as to maintain the current *modus operandi* of potential users such as government agencies and commercial organizations. At the same time, we must provide *reasonable* and *provable* privacy guarantees for the system.

These are the hard design requirements which we achieve with Blind Seer (BLoom filter INDex SEarch of Encrypted Results). Our work can be seen as an example of applying cryptographic rigor to design and analysis of a large system. Privacy/efficiency trade-offs are inherent in many large systems. We believe that the analysis approach we take (identifying and permitting a controlled amount of leakage, and proving that there is no additional leakage) will be useful in future secure systems.

Significance. We solve a significant open problem in private DB: efficient *sublinear* search for arbitrary Boolean queries. While private keyword-search was achieved in some models, this did *not* extend to general Boolean formulas. Natural breaking of a formula to terms and individual keyword-searching of each leaks formula structure and encrypted results for each keyword, significantly compromising privacy of both query and data. Until our work, and the (very different) independent and concurrent works [11], [31], it was *not* known how to efficiently avoid this leakage. (See Section IX for extended discussion on related work.)

A. Our Setting

Traditionally, DB querying is seen as a two-player engagement: the client queries the server operated by the data owner, although delegation of the server operation to a third player is increasingly common.

Players. In our system, there are three main players: client C, server S, and index server IS (there is another logical entity, query checker QC, whose task of private query compliance checking is technically secondary, albeit practically important. For generality, we consider QC as a separate player, although its role is normally played by either S or IS). We split off IS from S mainly for performance reasons, as two-player private DBMS querying has trivial linear in DB size lower bounds[1], while three non-colluding players allow for far better privacy-performance trade-offs. We note also that our system can be generalized to handle multiple clients in several ways (presenting different trade-offs), but we focus our presentation on the single client setting.

Allowed leakage. The best possible privacy for us would guarantee that C learns only the result set, and IS and S learn nothing at all. However, achieving this would be quite costly, and almost certainly far too expensive as a replacement for any existing DBMS. Indeed, practically efficient equality checking of encrypted data would likely require the use deterministic encryption, which allows to identify and accumulate access patterns. Additionally, for certain conjunctive queries, sublinear search algorithms are currently unknown, even for insecure DBMS. Thus, unless we opt for a linear time for all conjunctive queries, the running time already inevitably reveals some information (see Section VI-B for more discussion).

As a result, we accept that certain *minimal* amount of leakage is unavoidable. In particular, we allow players C and IS to learn certain *search pattern* information, such as the pattern of returned results, and the traversal pattern of the encrypted search tree. We stress that we still formally prove security of the resulting system – our simulators of players' views are given the advice corresponding to the allowed leakage. We specify the allowed leakage in more detail in Section VI.

We note that this work was performed under the IARPA SPAR program [1]. Many of the privacy and functionality requirements we address are suggested by IARPA. In Section X we provide further motivation, examples and discussion of our setting and choices.

B. Our Contributions

We design, prove secure, implement and evaluate the first scalable privacy-preserving DBMS which simultaneously satisfies *all* the following features (see the following sections for a more complete description and comparison to previous works):

- Rich functionality: we support a rich set of queries including arbitrary Boolean formulas, ranges, stemming, and negations, while hiding search column names and including free keyword searches over text fields in the database. We note that there is no standard way in MySQL to obtain the latter.
- Practical scalability. Our performance (similarly to MySQL) is proportional to the number of terms in the query and to the result set size for the CNF term with the *smallest* number of results.
  For a DB of size 10TB containing 100M records with 70 searchable index terms per DB row, our system executes many types of queries that return few results in well under a second, which is comparable to MySQL.
- Provable security. We guarantee the privacy of the data from both IS and C, as well as the privacy of C's query from S and IS. We prove security with respect to well defined, reasonable, and controlled leakage. In particular, while certain information about search patterns and the size of the result set is leaked, we do provide some privacy of the result set size, suited for the case when identifying that there is one result as opposed to zero results is undesirable (Section V-B).
- Natural integration of private policy enforcement. We represent policies as Boolean circuits over the query, and can support any policy that depends only on the query, with performance that depends on the policy circuit size.
- Support for DB updates, deletions and insertions.

To our knowledge the combination of performance, features and provable security of our system has never been achieved, even without implementation, and represents a breakthrough in private data management. Indeed, previous solutions either require at least linear work, address a more limited type of

[1]This lower bound can be circumvented if we allow precomputation, as done for example in the ORAM based schemes mentioned above. However, the resulting solution is far too inefficient for practice, as even its online phase is several orders of magnitude slower than our solution.

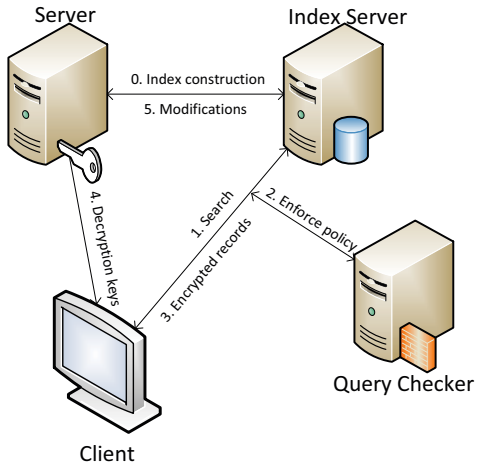


Figure 1. High-level overview of Blind Seer. There are three different operations depicted: preprocessing (step 0), database searching (step 1-4) and data modifications (step 5).

queries (e.g., just keyword search), or provide weaker privacy guarantees. The independent and concurrent work of [11], [31] (also performed under IARPA SPAR program) is the only system comparable to ours, in the sense that it too features a similar combination of rich functionality, practical scalability, provable security, and policy enforcement. However, the trade offs that they achieve among these requirements and their technical approach are quite different than ours.

Our scale captures moderate-to-large data, which encompasses datasets in the motivating scenarios above (such as the census data, on which we ran our evaluation), and represents a major step towards privacy for truly "big data". Our work achieves several orders of magnitude performance improvement as compared to the fully secure cryptographic solution, and much greater functionality and privacy as compared to practical single keyword search and heuristic solutions.

II. System Design Overview

Participants. Recall, our system consists of four participants: *server* S, *client* C, *index server* IS, and *query checker* QC. The server owns a database DB, and provides its encrypted searchable copy to IS, who obliviously services C's queries. QC, a logical player who can be co-located with and may often be an agent of S, privately enforces a policy over the query. This is needed to ensure control over hidden queries from C. Player interaction is depicted in Figure 1.

Our approach. We present a high-level overview of our approach and refer the reader to Section IV for technical details. We adhere to the following general approach of building large secure systems, in which full security is prohibitively costly: in a large problem, we identify small privacy-critical subproblems, and solve those securely (their outputs must be of low privacy consequence, and are handled in plaintext). Then we use the outputs of the subtasks (often only a small portion of them will need to be evaluated) to complete the overall task efficiently.

We solve the large problem (encrypted search on large DB) by traversing an encrypted search tree. This allows the subtasks of privately computing whether a tree node has a child matching the (arbitrarily complex) query to be designated as security-critical. Further, unlike the protected input and the internals of this subtask, its output, obtained in plaintext by IS, reveals little private information, but is critical in pruning the search tree and achieving efficient sublinear (logarithmic for some queries) search complexity. Putting it together, our search is performed by traversing the search tree, where each node decision is made via very efficient secure function evaluation (SFE).

We use Bloom filters (BF) to store collections of keywords in each tree node. Bloom filters serve this role well because they support small storage, constant time access, and invariance of access patterns with respect to different queries and match outputs. For SFE, we use state-of-the-art Yao's garbled circuits.

Because of SFE's privacy guarantee in each tree node, the overall leakage (i.e. additional information learned by the players) essentially amounts to the traversal pattern in the encrypted search tree.

We discuss technical details of these and other aspects of the system, such as encrypted search tree construction, data representation, policy checking, etc., in Section IV. We stress that many of these details are technically involved.

III. Preliminaries

We assume that readers are familiar with pseudorandom generators (PRG), pseudorandom functions (PRF), and semi-homomorphic encryption schemes with semantic security [28], e.g., ElGamal encryption [19].

Notations. Let $[n] = \{1, \ldots, n\}$. For ℓ-bit strings a and b, let $a \vee b$ (resp., $a \wedge b$ and $a \oplus b$) denote the bitwise-OR (resp. bitwise-AND and bitwise-XOR) of a and b. Let $S = (i_1, i_2, \ldots, i_\eta)$ be a sequence of integers. We define a projection of $a \in \{0,1\}^\ell$ on S as $a{\downarrow_S} = a_{i_1} a_{i_2} \cdots a_{i_\eta}$; for example, with $S = (2, 4)$, we have $0101{\downarrow_S} = 11$. We also define a filtering of $a = a_1 a_2 \ldots a_\ell$ by S as $a{\ddagger_S} = b_1 b_2 \ldots b_\ell$ where $b_j = a_j$ if $j \in S$, or $b_j = 0$ otherwise; for example, with $S = (2, 4)$, we have $1110{\ddagger_S} = 0100$. We define a shrinking function $\zeta_m : \mathbb{N}^\eta \to \mathbb{N}^\eta$ as $\zeta_m(i_1, i_2, \ldots, i_\eta) = (j_1, j_2, \ldots, j_\eta)$, where $j_k = (i_k - 1) \bmod (m + 1)$; for example, we have $\zeta_3(1, 3, 4) = (1, 3, 1)$.

Bloom filter (BF). A Bloom filter [8] is a well-known data structure that facilitates efficient search. The filter B is a string initialized with 0^ℓ and associated with a set of η different hash functions $\mathcal{H} = \{h_i : \{0,1\}^* \to [\ell]\}_{i=1}^{\eta}$. For a keyword $\alpha \in \{0,1\}^*$, let $\mathcal{H}(\alpha)$ the sequence of the hash results of α, i.e.,

$$\mathcal{H}(\alpha) = (h_1(\alpha), h_2(\alpha), \ldots, h_\eta(\alpha)).$$

To add a keyword α to the filter, the hash result $\mathcal{H}(\alpha)$ is added to it, that is, $B := B \vee (1^\ell{\ddagger_{\mathcal{H}(\alpha)}})$. To see if a keyword β is

in the filter, one needs to check if B contains $\mathcal{H}(\beta)$, that is, $B{\downarrow_{\mathcal{H}(\beta)}} \stackrel{?}{=} 1^{\eta}$. Bloom filters guarantee no false negatives, and allow the false positive rate to be tuned:

$$\mathrm{FP_{bf}} = \left(1 - \left(1 - \frac{1}{\ell}\right)^{\eta t}\right)^{\eta} \approx \left(1 - e^{-\frac{\eta t}{\ell}}\right)^{\eta},$$

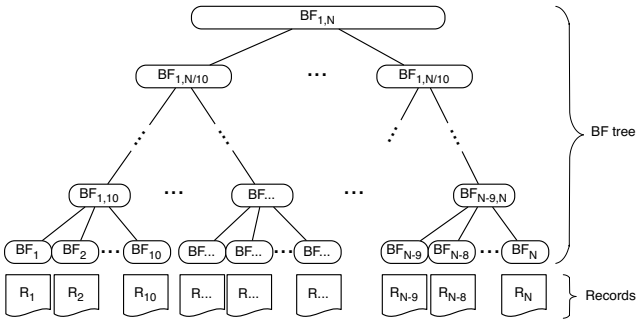
where t is the number of keywords in the Bloom filter. In our system, we choose $\eta = 20$ and $\ell = 28.86t$ to achieve $\mathrm{FP_{bf}} \approx 10^{-6}$.

A. Secure Computation Based on Yao's GC

Yao's garbled circuit (GC). Yao's garbled circuits allow circuits to be evaluated obliviously by one party on hidden inputs provided by another party. Let C be a Boolean circuit with n input wires, m gates, and one output wire; let $(1, \ldots, n)$ be the indices to the input wires and $q = n + m + 1$ be the index to the output wire. To generate a garbled circuit $\tilde{C}$, a pair of random keys w_i^0, w_i^1 are associated with each wire i in the circuit; key w_i^0 corresponds to the value '0' on wire i, while w_i^1 corresponds to the value '1'. Then, for each gate g in the circuit, with its input wires i, j and its output wire k, a garbled gate $\tilde{g}$ (consisting of four ciphertexts) is constructed so that it will enable one to recover $w_k^{g(b_i,b_j)}$ from $w_i^{b_i}$ and $w_j^{b_j}$ (refer to [14], [36], [40], [48] for more detail.) The garbled circuit $\tilde{C}$ is simply the collection of all the garbled gates. By recursively evaluating the garbled gates, one can compute the garbled key w_q^b given the keys $(w_1^{a_1}, \ldots, w_n^{a_n})$, where $b = C(a_1, \ldots, a_n)$. We will sometimes call wire keys corresponding to input/output *garbled input/output*, and denote them by $\tilde{a}$ and $\tilde{b}$, i.e., $\tilde{a} = (w_1^{a_1}, \ldots, w_n^{a_n}), \tilde{b} = w_q^b$. We will also use the notation of garbled evaluation $\tilde{b} = \tilde{C}(\tilde{a})$.

Oblivious transfer. An oblivious transfer (OT) [20], [46] is a two-party protocol supporting a sender that holds values (x_0, x_1) and a receiver that holds an index $r \in \{0, 1\}$. The receiver learns x_r, but neither the sender nor the receiver learns anything else, i.e., the receiver learns nothing about any other values held by the sender, and the sender learns nothing about the receiver's index. We use the Naor-Pinkas protocol [42] as a basis and optimize the performance using OT extension [30] and OT preprocessing [5].

Secure computation. It is known that Yao's garbled circuit, in combination with any oblivious-transfer protocol yields a constant-round protocol for secure two-party computation with semi-honest security [38], [52], [53]. In fact, due to the privacy guaranteed by Yao's GC [7], even if the circuit C is a private input from Alice along with x_A, Yao's GC can also hide the circuit C from Bob, revealing only the topology of C. We use GCs not only for search tree traversal but also for policy enforcement. Yao's GC is one of the most efficient algorithms known for secure computation of functions. For example, a recent work [51] demonstrated secure evaluation of AES (a circuit with 33880 gates) in 0.2 seconds. We use the standard techniques of Free-XOR [14], [36] and point-and-permute [40], [48] in constructing garbled circuits.



Let $(R_i, \ldots, R_n)$ be the overall database records. The Bloom filter $BF_{a,b}$ contains all the keywords of records $R_a, R_{a+1}, \ldots, R_b$.

Figure 2. Index structure: Bloom-filter-based search tree.

IV. Basic System Design

In this section, we will begin by describing the basic system design supporting only simple private query. In the next section, we will augment this basic design with more features.

A. BF Search Tree

Our key data structure enabling sublinear search is a BF search tree for the database records. We stress that there is only one global search tree for the entire database. Let n be the number of database records and T be a balanced b-ary tree of height $\log_b n$ (we assume $n = b^z$ from some positive integer z for simplicity). In our system, b is set to 10. In the search tree, each leaf is associated with each database record, and each node v is associated with a Bloom filter B_v. The filter B_v contains all the keywords from the (leaf) records that the node v have (as itself or as its descendants). For example, if a node contains a record that has `Jeff` in the `fname` field, a keyword $\alpha =$ `'fname:Jeff'` is inserted to B_v. The length ℓ_v of B_v is determined by the upper bound of the number of possible keywords, derived from DB schema, so that two nodes of the same level in the search tree have equal-length Bloom filters. The insertion of keywords is performed by shrinking the output of the hash functions $\zeta_{\ell_v}(\mathcal{H}(\alpha))$ to fit in the corresponding BF length ℓ_v. Here, $\mathcal{H}$ is the set of hash functions associated with the root node BF. See Figure 2.

Search using a BF search tree. Consider a simple recursive algorithm Search below. Let α and β be keywords and r the root of the search tree. Note that $\mathsf{Search}(\alpha \wedge \beta, r)$ will output all the leaves (i.e., record locations) containing both keywords α and β; any ancestor of a leaf has all the keywords that the leaf has, and therefore there should be a search path from the root to each leaf containing α and β. This algorithm can be easily extended to searching for any monotone Boolean formula of keywords.

$\mathsf{Search}(\alpha \wedge \beta, v)$:
  If the BF B_v contains α and β, then
    If v is a leaf, then output $\{v\}$.
    Otherwise, output $\bigcup_{c:}$ children of v $\mathsf{Search}(\alpha \wedge \beta, c)$.
  Otherwise, output $\emptyset$.

B. Preprocessing

Roughly speaking, in this phase, S gives an encrypted DB to IS. To be more specific, by executing the following protocols, the two parties encrypt and permute the records, create a search tree for the permuted records, and prepare record decryption keys.

Encrypting database index/records. In this step, the server first permutes its DB to hide information of the order of records in the DB and then creates BF-search tree on this permuted DB; these DB and search tree are encrypted and sent to the index server.

1) (Shuffle and encrypt the records.) The server generates a key pair (pk, sk) for a public-key semi-homomorphic (e.g., additively homomorphic) encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Given a database of n records, the server S randomly shuffles the records. Let $(R_1, \ldots, R_n)$ be the shuffled records. S then chooses a random string s_i, and computes $\tilde{s}_i \leftarrow \mathsf{Enc}_{pk}(s_i)$ and $\tilde{R}_i = G(s_i) \oplus R_i$, where G is a PRG.
2) (Encrypt the BF search tree.) S constructs a BF search tree T for the permuted records $(R_1, \ldots, R_n)$. It then chooses a key k at random for a PRF F. The Bloom filter B_v in each node v is encrypted as follows: $\tilde{B}_v = B_v \oplus F_k(v)$. (This encryption can be efficiently decrypted inside SFE evaluation by GC.)
3) (Share) Finally, the S sends the (permuted) encrypted records $(pk, (\tilde{s}_1, \tilde{R}_1), \ldots, (\tilde{s}_n, \tilde{R}_n))$ and the encrypted search tree $\{\tilde{B}_v : v \in T\}$ to the index server. The client will receive the PRF key k, and the hash functions $\mathcal{H} = \{h_i\}_{i=1}^{\eta}$ used in the Bloom filter generation.

Preparing record decryption keys. To save the decryption time in the on-line phase, the index server and the server precompute record decryption keys as follows:

(Blind the decryption keys) The index server IS chooses a random permutation $\psi : [n] \to [n]$. For each $i \in [n]$, it chooses r_i randomly and computes $\tilde{s}'_{\psi(i)} \leftarrow \tilde{s}_i \cdot \mathsf{Enc}_{pk}(r_i)$. Then, it sends $(\tilde{s}'_1, \ldots, \tilde{s}'_n)$ to S. Then, the server decrypts each $\tilde{s}'_i$ to obtain the blinded key s'_i. Note that it holds $s'_{\psi(i)} = s_i r_i$.

C. Search

Our system supports any SQL query that can be represented as a monotone Boolean formula where each variable corresponds to one of the following search conditions: *keyword match*, *range*, and *negation*. So, without loss of generality, we support non-monotone formulas as well, modulo possible performance overhead (see how we support negations below). See Figure 3 as an example.

Traversing the search tree privately. The search procedure starts with the client transforming the query into the corresponding Boolean circuit. Then, starting from the root of the search tree, the client and the index server will compute this circuit Q via secure computation. If the circuit Q outputs true, the parties visit all the children of the node, and again evaluate this circuit Q on those nodes recursively, until they reach leaf nodes; otherwise, the traversal at the node terminates. Note that evaluation of Q outputs a single bit denoting the search result at that node. It is fully secure, and reveals no information about individual keywords.

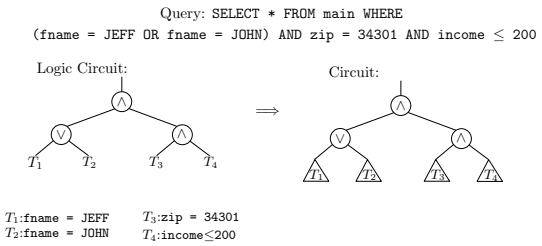


Figure 3. High level circuit representation of a query.

In order to use secure computation, we need to specify the query circuit and the inputs of the two parties to it. However, since the main technicalities lie in constructing circuits for the variables corresponding to search conditions, we will describe how to construct those sub-circuits only; the circuit for the Boolean formula on top of the variables is constructed in a standard manner.

Keyword match condition. We first consider a case where a variable corresponds to a keyword match condition. For example, in Figure 3 the variable T_1 indicates whether the Bloom filter B_v in a given node v contains the keyword α = `'fname:JEFF'`. Consider the Bloom filter hash values for the keyword α, and let Z denote the positions to be checked, i.e., $Z := \zeta_{\ell_v}(\mathcal{H}(\alpha))$. If the Bloom filter B_v contains the keyword α, the projected bits w.r.t. Z should be all set, that is, we need to check

$$B_v{\downarrow_Z} \stackrel{?}{=} 1^{\eta}. \tag{1}$$

Recall that the index server has an encrypted Bloom filter $\tilde{B}_v = B_v \oplus F_k(v)$, and the client the PRF key k and the hash functions $\mathcal{H} = \{h_i\}_{i=1}^{\eta}$. Therefore, the circuit to be computed should first decrypt and then check the equation (1). That is, the keyword match circuit looks as follows:

$$\mathsf{KM}((b_1, \ldots b_\eta), (r_1, \ldots, r_\eta)) = \bigwedge_{i=1}^{\eta} (b_i \oplus r_i).$$

Here, $(b_1, \ldots, b_\eta)$ is from the encrypted BF and $(r_1, \ldots, r_\eta)$ from the pseudorandom mask. That is, to this circuit KM, the index server will feed $\tilde{B}_v{\downarrow_Z}$ as the first part $(b_1, \ldots, b_\eta)$ of the input, and the client will feed $F_k(v){\downarrow_Z}$ as the second $(r_1, \ldots, r_\eta)$. In order that the two parties may execute secure computation, it is necessary that the client compute Z and send it (in plaintext) to the index server.

Range/negation condition. Consider the variable T_4 in Figure 3 for example. Using the technique from [47], we augment the BF to support inserting a number $x \in \mathbb{Z}_{2^n}$, say with $n = 32$, and checking if the BF contains a number in a given range.

To insert an integer a in a BF, *all* the canonical ranges *containing* a are added in the filter. A canonical range with level i is $[x2^i, (x+1)2^i)$ for some integer x, so for each level, there is only one canonical range containing the number a. In particular, for each $i \in \mathbb{Z}_n$, compute x_i such that $a \in [x_i 2^i, (x_i+1)2^i)$ and insert `'r:income:`i`:`x_i`'` to the Bloom filter.

Given a range query $[a, b)$, we check whether a canonical range *inside the given query* belongs to the BF. In particular, for each $i \in \mathbb{Z}_n$, find, if any, the minimum y_i such that $[y_i 2^i, (y_i+1)2^i) \in [a, b)$ and the maximum z_i such that $[z_i 2^i, (z_i+1)2^i) \in [a, b)$; then check if the BF contains a keyword `'r:income:`i`:`y_i`'` or `'r:income:`i`:`z_i`'`. If any of the checks succeeds for some i, then output yes; otherwise output no. Therefore, a circuit for a range query is essentially ORs of keyword match circuits.

For example, consider a range query with $\mathbb{Z}_{2^4}$. When inserting a number 9, the following canonical ranges are inserted: $[9, 10)$, $[8, 10)$, $[8, 12)$, $[8, 16)$. Given a range query $[7, 11)$, the following canonical ranges are checked: $[7, 8)$, $[10, 11)$, $[8, 10)$. We have a match $[8, 10)$.

Negation conditions can be easily changed to range conditions; for example, a condition `'NOT work hrs = 40'` is equivalent to `'work hrs` $\leq$ `39 OR work hrs` $\geq$ `41'`.

Overall procedure in a node. In summary, we describe the protocol that the client and the index server execute in a node of the search tree.

1) The client constructs a query circuit corresponding to the given SQL query. Then, it garbles the circuit and sends the garbled circuit, Yao keys for its input, and the necessary BF indices.
2) The client and the index server execute OT so that IS obtains Yao keys for its input (i.e., encrypted BF). Then, the index server evaluates the garbled circuit and sends the resulting output Yao key to the client.
3) The client decides whether to proceed based on the result.

Record Retrieval. When the client and the index server reach some of the leaf nodes in the tree, the client retrieves the associated records. In particular, if computing the query circuit on the ith leaf outputs success, the index server sends $(\psi(i), r_i, \tilde{R}_i)$ to the client. Then, the client sends $\psi(i)$ to S, and gets back $s'_{\psi(i)}$. Note that it holds $s'_{\psi(i)} := s_i r_i$. The client C decrypts $\tilde{R}_i$ using s_i and obtains the output record.

V. Advanced Features

In this section, we discuss how our system supports advanced features such as query policies, and one-case indistinguishability. We also overview insert/delete/update operations from the server.

A. Policy Enforcement

The policy enforcement is performed through a three-party protocol among the query checker QC (holding the policy), the client C (holding the query), and the index server IS. A policy is represented as a circuit that takes a query as input and outputs accept or reject. In our system, QC garbles this policy circuit, and IS evaluates the garbled policy circuit on the client's query. A key idea here is to have *the client and the query checker share the information of input/output wire key pairs in this garbled policy circuit;* then, the client can later construct a garbled query circuit (used in the search tree traversal) to be dependent on the output of the policy circuit. Assuming semi-honest security, this sharing of information can be easily achieved by having the client choose those key pairs (instead of QC) and send them to QC. The detailed procedure follows.

Before the tree search procedure described in the previous section begins, the client C, the query checker QC, and the index server IS execute the following protocol.

1) Let $\mathbf{q} = (q_1, \ldots, q_m) \in \{0,1\}^m$ be a string that encodes a query (we will discuss our encoding method in Appendix A). The client generates Yao key pairs $\mathbb{W}_{\mathbf{q}} = ((w_1^0, w_1^1), \ldots, (w_m^0, w_m^1))$ for the input wires of the policy circuit, and a key pair $\mathbb{W}_x = (t^0, t^1)$ for the output wire. The client sends the key pairs $\mathbb{W}_{\mathbf{q}}$ and $\mathbb{W}_x$ to query checker QC. It also sends the index server the garbled input $\tilde{\mathbf{q}} = (w_1^{q_1}, w_2^{q_2}, \ldots, w_m^{q_m})$.
2) Let P be the policy circuit. QC generates a garbled circuit $\tilde{P}$ using $\mathbb{W}_{\mathbf{q}}$ as input key pairs, and $\mathbb{W}_x$ as the output key pair (QC chooses the other key pairs of $\tilde{P}$ at random). Then, QC sends $\tilde{P}$ to the index server.
3) The index server evaluates the circuit $\tilde{P}$ on $\tilde{\mathbf{q}}$ obtaining the output wire key $\tilde{x} = \tilde{P}(\tilde{\mathbf{q}})$. Note that $\tilde{x} \in \mathbb{W}_x$.

After the execution of this protocol, the original search tree procedure starts as described before. However, the procedure is slightly changed when evaluating a leaf node as follows:

1) Let $Q'(\mathbf{b}, \mathbf{r}, x) = Q(\mathbf{b}, \mathbf{r}) \wedge x$ be an augmented circuit, where Q is the query circuit, $\mathbf{b}$ and $\mathbf{r}$ are the inputs from IS and C respectively, and x is a bit representing the output from the policy circuit. The client C generates a garbled query circuit $\tilde{Q}'$ using wire key pair $\mathbb{W}_x$ for the bit x. Then, it sends $(\tilde{Q}', \tilde{\mathbf{r}})$ to the index server, where $\tilde{\mathbf{r}}$ is the garbled input of $\mathbf{r}$.
2) After obtaining the input keys $\tilde{\mathbf{b}}$ for $\mathbf{b}$ from OT with C, the index server IS evaluates $\tilde{Q}'(\tilde{\mathbf{b}}, \tilde{\mathbf{r}}, \tilde{x})$ and sends the resulting output key to the client. Recall that it has already evaluated the garbled policy circuit $\tilde{P}(\tilde{\mathbf{q}})$ and obtained $\tilde{x}$.
3) The client checks the received key and decides to accept or reject.

Regarding privacy, the client learns nothing about the policy, since it never sees the garbled policy circuit. The index server obtains the topology of the policy circuit (from the garbled policy circuit).

Note that the garbled policy circuit is evaluated only once, before the search tree execution starts. Therefore, the policy checking mechanism introduces only a small overhead. It is also worth observing that, so far, we have not assumed any restriction on the policy to be evaluated. Since Yao-based

computation can compute any function represented as a circuit, in principle, we could enforce any policy computable in a reasonable time (as long as it depends only on the query). We describe our own implemented policy circuit in more detail in Appendix A.

B. One-case Indistinguishability

So far, in our system the index server learns how many records the client retrieved from the query. In many use cases, this leakage should be insignificant to the index server, in particular, when the number of returned results is expected to be, say, more than a hundred. However, there do exist some use cases in which this leakage is critical. For example, suppose that a government agent queries the passenger database of an airline company looking for persons of interest (POI). We assume that the probability that there is indeed a POI is small, and the airline or the index server discovering that a query resulted in a match may cause panic. Motivated from the above scenario, we consider a security notion which we call *one-case indistinguishability*.

Motivation. Consider a triple $(q, D_0, \mathbf{r})$ where q is a query, and D_0 is a database with the query q resulting in no record, but $\mathbf{r}$ satisfies q. Let D_1 be a database that is the same as D_0 except that a record is randomly chosen and replaced with $\mathbf{r}$. Let VIEW_0 (resp. VIEW_1) denote the view of IS when the client runs a query q with the database D_0 (resp., D_1).

A natural start would be to require that for any such $(q, D_0, \mathbf{r})$, the difference between the two distributions VIEW_0 and VIEW_1 should be small ϵ (in the computational sense), which we call ϵ zero-one indistinguishability. However, it does not seem possible to achieve negligible difference ϵ without suffering significant performance degradation (in fact, our system satisfies this notion for a tunable small constant ϵ). Unfortunately, this definition does not provide a good security guarantee when the difference ϵ is non-negligible, in particular, for the scenario of finding POIs. For example, let Π be a database system with perfect privacy and Π' be the same as Π except that when it is 1-case (i.e., a query with one result record), the client sends the index server the message "the 1-case occurred" with non-negligible probability. It is easy to see that Π' satisfies the definition with some non-negligible ϵ, but it is clearly a bad and dangerous system.

One-case indistinguishability. Observe that in the use case of finding POIs, we don't particularly worry about "the 0-case", that is, it is acceptable if the airline company sometimes knows that a query definitely resulted in no returned record. Motivated by this observation, this definition intuitively requires that if the a-priori probability of a 1-case is δ, then a-posteriori probability of a 1-case is at most $(1+\epsilon)\delta$. For example, for $\epsilon = 1$, the probability could grow from δ to 2δ, but never more than that, no matter what random choices were made. Moreover, if the a-priori probability was tiny, the a-posteriori probability remains tiny even if unlucky random choices were made. In particular, consider $(q, D_0, \mathbf{r})$ and D_1 as before. Now consider a distribution E that outputs (b, v) where $b \in \{0, 1\}$ chosen with $\Pr[b = 1] = \delta$, and v is the view of the index server when the query q is run on D_b. The system satisfies ϵ one-case indistinguishability if for any $(q, D_0, \mathbf{r})$, δ and v, it holds

$$\Pr_E[b = 1 | v] \leq (1 + \epsilon)\delta.$$

Augmenting the design. To achieve these indistinguishability notions, we change the design such that the client chooses a small random number of paths leading to randomly selected leaves. In particular, let $\mathcal{D}$ be the probability distribution on the number of random paths defined as follows:

$$\text{Distribution } \mathcal{D}: \quad \begin{array}{ll} \text{For } 1 \leq x \leq \alpha - 1, & \Pr_{\mathcal{D}}[x] = 1/\alpha. \\ \text{For } x \geq \alpha, & \Pr_{\mathcal{D}}[x] = (1/\alpha) \cdot 1/2^{x-\alpha+1}. \end{array}$$

Here, α is a tunable parameter. The client chooses $x \leftarrow \mathcal{D}$, and then it also chooses x random indices $(j_1, \ldots, j_x) \leftarrow [n]^x$. When handling the query, the client superimposes the basic search procedure above with these random paths. Our system is $1/\alpha$ zero-one indistinguishable and ϵ one-case indistinguishable with $\epsilon = 1$. Intuitively, the leakage to the index server is the tree traversal pattern, and these additional random paths make the 0-case look like 1-case with a reasonably good probability. We give more detail in Appendix B.

If we slightly relax the definition and ignore views taking place with a tiny probability, say 2^{-20}, we can even achieve both 1-case and 0-case indistinguishability at the same time; the probability of the number x of fake paths is now $1/2^{|x-\alpha|+2}$ with a parametrized center α, say $\alpha = 20$ (except when $x = 0$, i.e., $\Pr[x = 0] = 1/2^{\alpha+1}$).

Against the server. One-case indistinguishability against the server is easily achieved by generating a sufficient number of dummy record decryption keys in the preprocessing phase; the index server will let the client know the (permuted) positions of the dummy keys. When zero records are returned from a query, the client asks for a dummy decryption key from the server. For brevity, we omit the details here, and exclude this feature in the security analysis.

C. Delete, Insert, and Update from the Server

Our system supports a basic form of dynamic deletion, insertion, and update of a record which is only available to the server. If it would like to delete a record R_i, then the server sends i to the index server, which will mark the encrypted correspondent as deleted. For newly inserted (encrypted) records, the index server keeps a separate list for them with no permutation involved. In addition, it also keeps a temporary list of their Bloom filters. During search, the temporary list is also scanned linearly, after the tree. When the length of the temporary Bloom filter list reaches a certain threshold, all the current data is re-indexed and a new Bloom filter tree is constructed. The frequency of rebuilding the tree is of course related to the frequency of the modifications and also the threshold we choose for the temporary list's size. Our tree building takes one hour/100M records. Finally, update is simply handled by atomically issuing a delete and an insert command.

Functionality $\mathcal{F}_{db}$

Parameter: Leakage profile.

Init: Given input (D, P) from S, do the following:

1) Store the database records D and the policy P. Let n be the number records in D. Shuffle D and let $(R_1, \ldots, R_n)$ be the shuffled records. Choose a random permutation $\pi : [n] \to [n]$. Construct a BF-search tree for $(R_1, \ldots, R_n)$ using the hash functions $\mathcal{H}$.
2) To handle the client's queries, it chooses hash functions $\mathcal{H} = \{h_i : \{0,1\}^* \to [\ell]\}_{i=1}^{\eta}$ for Bloom filters with parameters (η, ℓ) to maintain false positive rate of 10^{-6}.
3) Finally, return a $\mathsf{DONE}_{\mathsf{init}}$ and the leakage to all parties.

Query: Given input $\mathbf{q}$ from C, do the following:

1) Check if $\mathbf{q}$ is allowed by P. If the check fails, then disallow the query by setting $y = \emptyset$. Otherwise, for each $i \in [n]$, let $B_i \in \{0,1\}^{\ell'}$ be the Bloom filter associated with the ith leaf in the BF tree. For $i = 1, \ldots, n$, check if the query passes according to the filter B_i (refer to Section II); if so, add (i, R_i) to the result set Y.
2) Return Y to C and return a $\mathsf{DONE}_{\mathsf{query}}$ message and leakage to all parties.

Figure 4. The Ideal Functionality $\mathcal{F}_{db}$

We note that updates is not our core contribution; we implement and report it here, but don't focus on its design and performance. A more scalable update system would use a BF tree rather than a list; its implementation is a simple modification to our system.

VI. Security Analysis

In this section, we present an overview of the security of our system. A full analysis with formal definitions and extensive proofs is completed and written separately.

We consider static security against a semi-honest adversary that controls at most one participant. We first describe an ideal functionality $\mathcal{F}_{db}$ parameterized with a leakage profile in Figure 4, and then show that our system securely realizes the functionality where the leakage is essentially the search tree traversal pattern and the pattern of accessed BF indices.

For the sake of simplicity, we only consider security where there are no insert/delete/update operations,[2]and unify the server and the query checker into one entity. We also assume that all the records have the same length.

We use the DDH assumption (for ElGamal encryption and Naor-Pinkas OT), and our protocols are in the random oracle model (for Naor-Pinkas OT and OT extension). We also use PRGs and PRFs, and those primitives are implemented with AES.

[2] As access patterns are revealed, additional information for inserted/deleted/updated records is leaked. For example, C or IS may learn whether a returned record was recently inserted; they also may get advantage in estimating whether the query matched a recently deleted record. We stress that this additional leakage can be removed by re-running the setup of the search structure.

A. Security of Our System

With empty leakage profile, the ideal functionality $\mathcal{F}_{db}$ in Figure 4 captures the privacy requirement of a database management system in which each query is handled deterministically. The client obtains only the query results, but nothing more. The index server and the server learn nothing. Realizing such a functionality incurs a performance hit. Our system realizes the functionality $\mathcal{F}_{db}$ with the leakage profile described below. The security of our system can be proved from the security of the secure computation component, and is deferred to the full version.

Leakage in Init. Since the server has all the input, the leakage to S is none. The leakage to C is n, that is, the total number of records. The leakage to IS is n and $|R_1|$.

Leakage to S in each query. We first consider the leakage to the server. The server is involved only when the record is retrieved. Let $((i_1, R_{i_1}), \ldots, (i_j, R_{i_j}))$ be the query results. Then, the leakage to the server is $(\pi(i_1), \pi(i_2), \ldots, \pi(i_j))$.

Leakage to C in each query. The leakage to the client is the BF-search tree traversal paths, that is, all the nodes v in which the query passes according to the filter B_v.

Leakage to IS in each query. The leakage to the index server is a little more than that to the client. In particular, the nodes in the faked paths that the client generates due to one-case indistinguishability are added to the tree search pattern. Also, the topology of the query circuit and of the policy circuit is leaked to IS as well. Finally, the BF indices are also revealed to IS (although not the BF content), but assuming that the hash functions are random, those indices reveal little information about the query. However, based on this, after observing multiple queries, IS can infer some correlations a C's queries' keywords.

B. Discussion

Leakage to the server. We could wholly remove the leakage to the server by modifying the protocol as follows:

> Remove the decryption key preparation (and blinded keys) in the preprocessing; instead, the client receives the secret key sk from the server. The client (as the receiver) and the index server (as the sender) execute oblivious transfer at each leaf of the search tree. The choice bit of the client is whether the output of the query circuit is success. The two messages of the index server is the encrypted record and a string of zeros.

However, we believe that it is important for the server to be able to upper-bound the number of retrieved records. Without such control, misconfiguration on the query checker side may allow overly general queries to be executed, causing too many rows to be returned to the client; in contrast, in our approach, S releases record decryption keys at the end, and therefore it is easy to enforce the sanity check of the total number of returned records. Moreover, if S has a commercial DB, it may

be convenient to implement payment mechanism in association with key release by S.

OR queries. For OR queries passing the policy, our system leaks extremely small information. In particular, the leakage to the client is minimal, as the tree traversal pattern can be reconstructed from the returned records. As a consequence, if the client retrieves only document ids, the client learns nothing about the results for individual terms in his query. The leakage to the index server is similar. We believe that the topology of the SQL formula and the policy circuit reveals small information about the query and the policy. If desired, we can even hide those information using universal circuits [37] with a circuit size blow-up of a logarithmic multiplicative factor.

AND queries. For AND queries, the tree traversal pattern consists of two kinds of paths. The first are, of course, the paths reaching the leaves (query results). The second stop at some internal nodes due to our BF approach[3]; although the leakage from this pattern reveals more information about which node don't contain a given keyword, we still believe this leakage is acceptable in many use cases.

We stress that the second leakage is related to the fact that a large linear running time seems to be *inherent* for some AND queries, irrespective of privacy, but depending only on the underlying database (see Section VIII-C for more detail). Therefore, if we aim at running most AND queries in sublinear time, the running time will inherently leak information on the underlying DB.

VII. Implementation

We built a prototype of the proposed system to evaluate its practicality in terms of performance. The prototype was developed from scratch in C++ (a more than a year effort, almost two years including designing) and consists of about 10KLOC. In this section, we describe several interesting parts of the implementation that are mostly related to the scalability of the system.

Crypto building blocks. We developed custom implementations for all the cryptographic building blocks that were previously described in Section II. More specifically, we used the GNU Multiple Precision (GMP) library to implement oblivious transfers, garbled circuits and the semi-homomorphic key management protocol. The choice of GMP was mostly based on thread-safety. As for AES-based PRF, we used the OpenSSL implementation because it takes advantage of the AES-NI hardware instructions, thus delivering better performance.

Parallelization. The current implementation of Blind Seer supports parallel preprocessing and per-query threading when searching. For all the multi-threading features we used Intel's Threading Building Blocks (TBB) library. To enable multi-threaded execution of the preprocessing phase we created a 3-stage pipeline. The first stage is single-threaded and it is responsible for reading the input data. The second stage handles record preprocessing. This stage is executed in parallel by a pool of threads. Finally, the last stage is again single-threaded and is responsible for handling the encrypted records. Concurrently supporting multiple queries was straightforward as all the data structures are read-only. To avoid accessing the Bloom filter tree while it is being updated by a modification command, we added a global writer lock (which does not block reads). Since we only currently support parallelization on a one-thread-per-query basis, it only benefits query throughput, not latency. However, long-running queries involve a large amount of interaction between querier and server that is independent and thus amenable to parallelization. The improvement we see in throughput is a good indicator for how much we could improve latency of slow queries by applying parallelization to these interactions.

Bloom filter tree. This is the main index structure of our system which grows by the number of records and the supported features (e.g., range). For this reason, the space efficiency of the Bloom filter tree is directly related to the scalability of the system. In the current version of our system we have implemented two space optimizations: one on the representation of the tree and another on the size of Bloom filter in each tree node.

Firstly, we avoided storing pointers for the tree representation, which would result in wasting almost 1G of memory for 100M records. This is achieved by using a flat array with fixed size allocations per record.

Secondly, we observed that naively calculating the number of items stored in the inner nodes by summing the items of their children is inefficient. For example, consider the case of storing the 'Sex' field in the database, which has only two possible values. Each Bloom filter in the bottom layer of the tree (leaves) will store either the value `sex:male` or `sex:female`. However, their parent nodes will keep space for 10 items, although the Sex field can have only two possible values. Thus, we estimate the number of items that need to be stored in a given level as the minimum between the cardinality of the field and the number of leaf-nodes of the current subtree. This optimization alone reduced the total space of the tree by more than 50% for the database we used in our evaluation.

Keyword search and stemming. Although we focus on supporting database search on structured data, our underlying system works with collections of keywords. Thus, it can trivially handle other forms of data, like keyword search over text documents, or even keyword search on text fields of a database. We actually do support the latter – in our system we provide this functionality using the special operator `CONTAINED_IN(column, keyword)`. Also, we support stemming over keyword search by using the Porter stemming algorithm [2].

VIII. Evaluation

In this section, we evaluate our system. We first evaluate our system as a comparison with MySQL as a baseline, to establish

[3] For example, consider a query q that looks for two keywords, say, $q = \alpha \wedge \beta$. Let v be some node and $c_1, \ldots, c_b$ be the children of v in the search tree. If c_1 contains only α, and c_2 contains only β, then v will contain both α and β, and so the node v will pass the query; however, neither c_1 nor c_2 would.

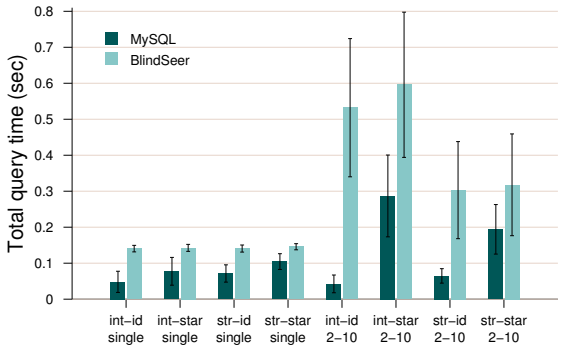


Figure 5. Comparison with MySQL for single-term queries that have a single result (first four bar groups) and 2 to 10 results (last four bar groups). The search terms are either strings (str) or integers (int) and the returned result is either the id or the whole record (star).

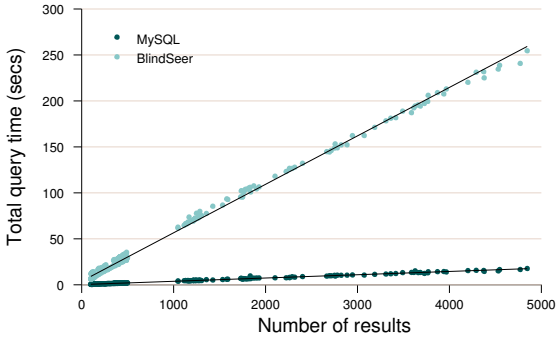


Figure 6. Comparison of the scaling factor with respect to the result set size, using single-term queries. Both MySQL and Blind Seer scale linearly, however, Blind Seer's constant factor is $15\times$ worse (mostly due to increased network communication).

what the performance cost of providing private search is. We then generalize the performance expectations of our system by performing a theoretical analysis based on the type of queries.

Dataset. The dataset we use in all of our tests for the first part of the evaluation is a generated dataset using learned probability distributions from the US census data and text excerpts from "The Call of the Wild", by Jack London. Each record in our generated database contains personal information generated with similar distributions to the census. It also contains a globally unique ID, four fields of random text excerpts ranging from $10 - 2000$ bytes from "The Call of the Wild", and a "fingerprint" payload of random data ranging from 50000 to 90000 bytes. The payload is neither searchable nor compressible, and is included to emulate reasonable data transfer costs for real-world database applications. The census data fields are used to enable various types of single-term queries such as term matching and range queries, and the text excerpts for keyword search queries.

Testbed. Our tests were run on a four-computer testbed that Lincoln Labs set up and programmed for the purpose of testing our system and comparing it to MySQL. Each server was configured with two Intel Xeon 2.66 Ghz X5650 processors, 96GB RAM (12x8 GB, 1066 MHz, Dual Ranked LV RDIMMs), and an embedded Broadcom 1GB Ethernet NICS with TOE. Two servers were equipped with a 50TB RAID5 array, and one with a 20TB array. These were used to run the owner and index server. MySQL was configured to build separate indices for each field. DB queries were not known in advance for MySQL or for our system.

A. Querying Performance

Single term queries with a small result set. Figure 5 shows a comparison of single term queries against MySQL. We expect the run time for both our system and MySQL to depend primarily on the number of results returned. The first four pairs show average and standard deviation for query time on queries with exactly one result in the entire database, and the latter four for queries with a few (2-10) results. Queries are further grouped into those which are run on integer fields (int) and string fields (str), and those which return only record ids (id) and those which return full record content (star). For each group, we executed 200 different queries to avoid caching effects in MySQL.

As we can see, for single result set queries, our system is very consistent. Unlike with MySQL, the type of query has no effect on performance, since all types are stored and queried the same way in the underlying Bloom filter representation. Also, the average time is dominated by the average number of results, which is slightly larger for integer terms. Unexpectedly, there is also no performance difference for returning record ids versus full records. This is likely because for a single record, the performance is dominated by other factors like circuit evaluation, tree traversal and key handling, rather than record transfer time. Overall, aside from some bad-case scenarios, we are generally less than $2\times$ slower.

Variation in performance of our system is much larger when returning a few results. This is because the amount of tree traversal that occurs depends on how much branching must occur. This differs from single result set queries, where each tree traversal is a single path. With the larger result sets, we can also begin to see increased query time for full records as opposed to record ids, although it remains a small portion of the overall run time.

Scaling with result set size. Figure 6 expands on both systems' performance scaling with the number of results returned. This experiment is also run with single term queries, but on a larger range of return result set sizes. As one would expect, the growth is fairly linear for both systems, although our constant factor is almost $15\times$ worse. This indicates that for queries with a small result set, the run time is dominated by additive constant factors like connection setup for which we are not much slower than MySQL. However, the multiplicative constant factors involved in our interactive protocol are much larger, and grow to dominate run time for longer running queries. This overhead is mostly due to increased network communication because of the interactiveness of the search

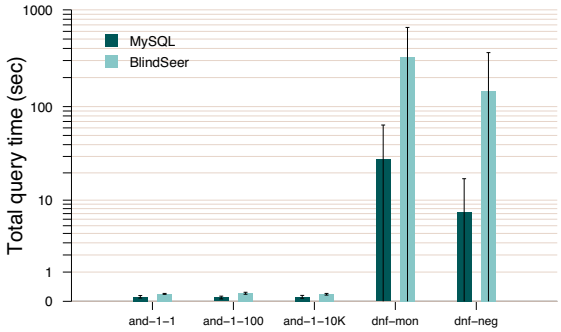


Figure 7. Boolean queries having a few results (< 10). The first three are two-term AND queries where one of the terms has a single result and the other varies from 1 to 10K results. The fourth group includes monotonic DNF queries with 4-9 terms, the last includes 5-term DNF queries with negations.

protocol. Although this is inherent, we believe that there is room for implementation optimizations that could lower this constant factor.

Boolean queries. Figure 7 shows our performance on various Boolean queries. The first three groups show average query time for 2-term AND queries. In each case, one term occurs only once in the database, resulting in the overall Boolean AND having only one match in the database. However, the second term increases up to 10000 results in the database. As we can see, our query performance does not suffer; as long as at least one term in a Boolean is infrequent we will perform well. The next two groups are more complex Boolean queries issued in disjunctive normal form, the latter including negations. The first one includes queries with 4-9 terms, and the second one, with 5 terms. These incur a larger cost, as the number of a results is larger and possibly a bigger part of the tree is explored. As we can see, MySQL incurs a proportionally similar cost.

We note that the relatively large variation shown in the graph is due to the different queries used in our test. Variation is much smaller when we run the same query multiple times.

Parallelization. We have implemented a basic form of parallelization in our system, which enables it to execute multiple queries concurrently. As there are no critical sections or concurrent modifications of shared data structures during querying, we saw the expected linear speedup when issuing many queries up to a point where the CPU might not be the bottleneck anymore. In our 16-core system, we achieved approximately factor 6x improvement due to this crude parallelization.

Discussion. We note several observations on our system, performance, bottlenecks, etc.

Firstly, we note that our experiments are run on a fast local network. A natural question is how this would be translated into the higher-latency lower bandwidth setting. Firstly, there will be performance degradation proportional to bandwidth reduction, with the following exception. We could use the slightly more computationally-expensive, but much less communication intensive GESS protocol of [34] or its recent extension sliced-GESS [35], instead of Yao's GC. In reduced-bandwidth settings, where bandwidth is the bottleneck, sliced-GESS is about 3x more efficient than most efficient Yao's GC. Further, we can easily scale up parallelization factor to mitigate latency increases. Looking at this in a contrapositive manner, improving network bandwidth and latency would make CPU the bottleneck.

All search structures in our system are RAM-resident. Only the record payloads are stored on disk. Thus, disk should not be a bottleneck in natural scenarios.

B. Other Operations

Although querying is the main operation of our system, we also include some results of other operations. First, we start with the performance of the setup phase (preprocessing). Blind Seer took roughly two days to index and encrypt the 10TB data. As mentioned before, this phase is executed in parallel and is computationally efficient enough to be IO-bounded in our testbed. We note that the corresponding setup of MySQL took even longer.

Policy enforcement was another feature for which we wanted to measure overhead. However, in our current implementation, it cannot be disabled (instead, we use a dummy policy). We experimentally measured the overhead of enforcing the dummy policy versus more complex ones, but there was no noticeable difference. We plan to add the functionality to totally disable policy enforcement – because it is an optional feature – and measure its true performance overhead. Our expectation is that it will be minimal.

Finally, we performed several measurements for the supported modification commands: insert, update and delete. All of them execute in constant time in the order of a few hundred microseconds. The more expensive part though is the periodic re-indexing of the data that merges the temporary Bloom filter list in the tree (see Section V-C). In our current prototype, we estimated this procedure to take around 17 minutes, while avoiding re-reading the entire database. This can be achieved by letting the server store some intermediate indexing data during the initial setup and reusing it later when constructing the Bloom filter tree.

C. Theoretical Performance Analysis

In this section, we discuss the system performance for various queries by analyzing the number of visited nodes in the search tree. Let $\alpha_1, \ldots, \alpha_k$ be k single term queries, and for each $i \in [k]$, let r_i be the number of returned records for the query α_i, and n be the total number of records.

OR queries. Our system shows great performance with OR queries. In particular, consider a query $\alpha_1 \vee \cdots \vee \alpha_k$. The number of visited nodes in the search tree is at most $r \log_{10} n$, where $r = r_1 + \ldots + r_k$ is the number of returned records. Therefore, performance scales with the size of the result set, just like single term queries.

AND queries. The performance depends on the best constituent term. For the AND query $\alpha_1 \wedge \cdots \wedge \alpha_k$, the number of

visited nodes in the search tree is at most $\min(r_1, \ldots, r_k) \cdot \log_{10} n$. Note that the actual number of returned records may be much smaller than r_is. In the worst case, it may even be 0; consider a database where a half of the records contain α (but not β) and the other half β (but not α). The running time for the query $\alpha \wedge \beta$ in this case will probably be linear in n. However, we stress that this seems to be *inherent*, even without any security. Indeed, without setting up an index, every algorithm currently known runs in linear time to process this query.

This can be partially addressed by setting up an index, in our case by using a BF. For example, for AND queries on two columns, for each record with value `a` for column `A`, and value `b` for column `B`, the following keywords are added: `A:a, B:b, AB:a.b`. With this approach, the indexed AND queries become equivalent to single term queries. However, this cannot be fully generalized, as space grows exponentially in the number of search columns.

Complex queries. The performance of CNF queries can be analyzed by viewing them as AND queries where each disjunct (i.e, OR query) is treated as a single term query. In general, any other complex Boolean query can be converted to CNF and then analyzed in a similar manner. In other words, performance scales with the number of results returned by the best disjunct when the query is represented in CNF. Note that we do *not* actually need to convert our queries to this form

(nor know anything about the data, in particular, which are high- or low-entropy terms) in order to achieve this performance (this aspect is even better than MySQL).

Computation and Communication. Both computational and communication resources required for our protocol are proportional to the query complexities described above.

False Positives. As our system is built on Bloom filters, false positives are possible. In our experiments, we set each BF false positive rate to 10^{-6}. Assuming the worst-case scenario for us, where the DB is such that many of the search paths do reach and query the BFs at the leaves, this gives 10^{-6} false positive probability for each term of the query. Of course, the false positive is a tunable parameter of our system.

IX. Related Work

The problem of private DBMS can be solved by general purpose secure computation schemes [26], [38], [52], [53]. These solutions, however, involve at least linear (often much more) work in the database size, hence cannot be used for practical applications with large data. Oblivious RAM (ORAM) [27] can be used to completely hide the client's query pattern, and can also be used as a tool to achieve sublinear amortized time for secure computation if we allow to leak the program running time [29], [39]. Nonetheless, computational costs are still prohibitively high, rendering these solutions impractical for the scale we are interested in.

Private Information Retrieval protocols (PIR) [16] consider a scenario where the client wishes to retrieve the ith record of the server's data, keeping the server oblivious of the index i. Symmetric PIR protocols [24] additionally require that client should not learn anything more than the requested record. While most PIR and SPIR protocols support record retrieval by index selection, Chor et al. [15] considered PIR by keyword. Although these protocols have sublinear communication complexity, their computation is polynomial in the number of records, and inefficient for practical uses.

Another approach would be to use fully homomorphic encryption (FHE). In 2009, Gentry [21] showed that FHE is theoretically possible. Despite this breakthrough and many follow up works, current constructions are too slow for practical use. For example, it is possible to homomorphically compute 720 AES blocks in two and a half days [23].

Little work has appeared on practical, private search on a large data. In order to achieve efficiency, weaker security (some small amount leakage) has been considered. The work of [44], [47] supports single keyword search and conjunctions. However, the solution does not scale well to databases with a large number of records (say millions); its running time is linear in the number of DB records. One of the interesting features of this work is the way they address range queries. Our system also uses their idea to achieve range queries, and extends it to support negations (since we use a sublinear underlying OR query, our range queries are also sublinear, in contrast to them). A more efficient solution towards this end was proposed in [18]. However, they only considered single keyword search.

Any single keyword search solution can be used to solve (insecurely) arbitrarily Boolean formulas; solve each keyword in the formula separately and then combine (insecurely). Obviously, however, this leaks much more information to the parties (and also has work proportional to the sum of the work for each term). Our system, in contrast, provides privacy of the overall query (and work proportional to just the smallest term).

There has been a long line of research on searchable symmetric encryption (SSE) [11]–[13], [17], [25], [41], [50]. Note that, although many of the techniques used in SSE schemes can be used in our scenario, the SSE setting focuses on data outsourcing rather than data sharing. That is, in SSE the data owner is the client, and so no privacy against the client is required. Additionally, SSE solutions often offer either a linear time search over the number of database records [12], [41], [50], or a restricted type of client's queries [17], [32], namely single keyword search or conjunctions. One exception is the recent SSE scheme of [11], which extended the approach of [17] to allow for any Boolean formula of the form $k_0 \wedge \phi(k_1, ..., k_{m-1})$, where $\phi(\cdot)$ is an arbitrarily Boolean formula. Their search time complexity is $O(m \times D(k_0))$, where $D(k_0)$ is the number of records containing keyword k_0. Note that an arbitrary formula could be represented this way, as k_0 can always be set to *true*, but then the complexity will be linear in the number of records. On the other hand, if the client can format the query so that k_0 is a term with very few matches, the complexity will go down accordingly. In contrast, our solution addresses arbitrary Boolean formulas,

with complexity proportional to the best term in the CNF representation of the formula. Searchable encryption has also been studied in the public key setting [4], [6], [9], [10], [49]. Here, many users can use the server public key to encrypt their own data and send it to the server.

The CryptDB system [45] addresses the problem of DB encryption from a completely different angle, and as such is largely incomparable to our work. CryptDB does not aim to address the issue of the privacy of the query (but it does achieve query privacy similar to the single-keyword search solution described above). Their threat scenario focuses on DB data confidentiality against the curious DB administrator, and they achieve this by using a non-private DBMS over what they call SQL-aware encrypted data. That is, the SQL query is pre-processed by a fully trusted proxy that encrypts the search terms of the query. The query is then executed by standard SQL, which combines the results of individual-term encrypted searches. Additionally, for free-text search, CryptDB uses the linear solution of [50].

The closest to our setting/work is a a very recent extension [31] of the SSE solution [11], which additionally (to the SSE requirements) addresses data privacy against the client (and hence, as we do, addresses private DB). We note that the work of [11], [31] is performed independently and concurrently to ours. [31] support the same class of functions as [11] (discussed above). In the worst case, such as when the client has little *a priori* information about the DB and chooses a sub-optimal term to appear first in the query term, the complexity of the [31] solution can be linear in the DB size. In contrast, our solution for general formulas does not depend on the client's knowledge of data distribution or representation choice (beyond the size of the formula). However, we note that for typical practical applications this is not a serious issue, as the client can represent his query as a conjunction, and moreover, can make a good guess for which term will have low frequency in the data and is a good choice as the first term. Thus, a large majority of practically useful queries can be evaluated by [31] with asymptotic complexity similar to ours. In terms of security, our guarantees vary: [31] achieves security against malicious client, which is much stronger than our semi-honest setting, and of particular importance for the policy enforcement. Our leakages vary and are incomparable. We and [31] leak different access pattern structures (search tree for us and index lookups for [31]). Because we use a more expensive basic step of SFE, our protection of query-related data, at least in some cases, is somewhat better. For example, depending on the DB data, we may hide everything about the individual terms of the query, while [31] leak to the client and (their counterpart of the) IS the support sizes for individual terms of the disjunctive queries (individual term supports are revealed to the client, but this is only an issue if the query does not ask for all the columns of the records).

At the same time, the concrete query performance of [31] is somewhat better than ours, due to their elegant non-interactive approach. The very expensive step of DB setup is faster for us, and the CPU load is lower, as we use mainly symmetric-key primitives. We also note that our interactive approach allows significant flexibility. For example, the 0-1 security (cf. Section V-B), is naturally and cheaply achievable in our system; it appears harder/more expensive to achieve in a non-interactive system, and in fact is not considered in [10]. The use of GC as the basic block similarly provides significant flexibility and opportunities for feature expansion. A strong point of [31] is easy scalability due to storing search structures on disk. This is achieved at the cost of significant additional system complexity and setup time. Finally, [31] naturally support multiple clients, while our natural extensions to multiple clients require that all clients share a secret key not known to IS.

Because of the different trade offs presented by our work and that of [31], each system is better suited for different applications/use cases. It is interesting to note that these two works, the first ones to address the major open problem of truly practical, provably secure, and very rich (including any formula) query DBMS, are based on very different technical approaches. We believe that this adds to the value and strength of each of these systems.[4]

X. Discussion and Motivation of Our Setting

Semi-honest model. Semi-honest model is often reasonable in practice, especially in the Government use scenarios. For example, C, S and index server may be Government agencies, whose systems are verified and trusted to execute the prescribed code. Further, regular audits will help enforce semi-honest behavior.

Security against malicious adversaries can be added by standard techniques, but this results in impractical performance. In follow up work we show how to amend our protocols to protect against one malicious player (C or IS) at a very small cost (ca. 10% increase). This is possible mainly because the underlying GC protocols are already secure against malicious evaluator.

Impact of the allowed leakage. Formally pinning down exact privacy loss is beyond the reach of state-of-the-art cryptography, even with no leakage beyond the output and amount of work (the field of differential privacy is working on this problem, with very moderate success). Therefore, understanding our leakage and its impact for specific applications is crucial to ascertain whether it's acceptable. We informally investigated the impact of leakage in several natural applications, such as population DBs and call-record DBs and query patterns (see example below); we believe that our protection is insufficient in some scenarios, while in many others it provides strong guarantees.

Rough leakage estimation for call-records DB. Consider a call-records DB, including columns (`Phone number`, `Callee phone number`, `time of call`). The client C is allowed to only ask queries of the form `select * where`

[4]We note that in an earlier stage there were two other performers on the IARPA SPAR program. However, we do not know the details of their approaches, and are not aware of published work presenting their solutions.

`phone number = xxx AND callee phone number = yyy AND time of call ∈ {interval}`.

For typical call patterns (e.g.,0-10 calls/person/day), the query leakage will almost always constitute a tree with branches either going to the leafs (returned records) or truncated one or two levels from the root. We believe that for many purposes this is acceptable leakage. Again, we stress that this is not a formal or detailed analysis (which is beyond the reach of today's state-of-the-art); it is included here to support our belief that our system gives good privacy protection in many reasonable scenarios.

Reliance on the third party. While a two-party solution is of course preferable, these state-of-the-art solutions are orders of magnitude slower than what is required for scalable DB access. Probably the most reasonable approach would be to use ORAM, which is set up either by a trusted party or as a (very expensive) 2-PC between data owner and the querier. Then the querier can query the ORAM held by the data owner. Due to privacy requirements, each ORAM step must be done over encrypted data, which triggers performance that is clearly unacceptable for the scale required in our application (cf. [29]).

Further, in Government use cases, employing third party is often seen as reasonable. For example, such a player can be run by a neutral agency. We emphasize that the third party is *not* trusted with the data or queries, but is trusted not to share information with the other parties.

XI. Conclusion

Guaranteeing complete search privacy for both the client and the server is expensive with today's state of the art. However, a weaker level of privacy is often acceptable in practice, especially as a trade-off for much greater efficiency. We designed, proved secure, built and evaluated a private DBMS, named Blind Seer, capable of scaling to tens of TB's of data. This breakthrough performance is achieved at the expense of leaking search tree traversal information to the players. Our performance evaluation results clearly demonstrate the practicality of our system, especially on queries that return a few results where the performance overhead over plaintext MySQL was from just $1.2\times$ to $3\times$ slowdown.

We note that the range from complete privacy to best performance is wide and our work only targets a specific point within it. We see it as a step towards exploring several other trade-offs in this space. Our goal for future work is to develop a highly tunable system which will be able to be configured and match many practical scenarios with different privacy and performance requirements.

Acknowledgments. 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. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

Fernando Krell was supported by BECAS CHILE, CONICYT, Gobierno de Chile.

This material is based upon work supported by (while author Keromytis was serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We thank MIT Lincoln Labs researchers for supporting this program from the beginning to the end and facilitating extensive testing of our code.

Finally, we thank our colleagues from other IARPA SPAR teams for great collaboration and exchange of ideas.

References

[1] IARPA Security and Privacy Assurance Research (SPAR) program. http://www.iarpa.gov/Programs/sso/SPAR/spar.html.

[2] The porter stemming algorithm. http://tartarus.org/martin/PorterStemmer/.

[3] Privacy groups file lawsuit over license plate scanners. http://www.therepublic.com/view/story/210d27e7585543a3941f5e577cf7f627/CA--License-Plate-Suit.

[4] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *J. Cryptol.*, 21(3):350–391, 2008.

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

[6] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *Proceedings of CRYPTO'07*, 2007.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, Oct. 2012.

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

[9] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of EUROCRYPT'04*, pages 506–522, 2004.

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

[11] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Aug. 2013.

[12] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, volume 3531, 2005.

[13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 577–594. Springer, Dec. 2010.

[14] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-XOR" technique. In R. Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Mar. 2012.

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

[16] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[17] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 06*, pages 79–88, 2006.

[18] E. De Cristofaro, Y. Lu, and G. Tsudik. Efficient techniques for privacy-preserving sharing of sensitive information. In *TRUST'11*, pages 239–253, 2011.

[19] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[20] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *CRYPTO'82*, pages 205–210. Plenum Press, New York, USA, 1982.

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

[22] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2013.

[23] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Aug. 2012.

[24] 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.

[25] E.-J. Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[26] 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.

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

[28] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[29] 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.

[30] 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.

[31] S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 875–888. ACM Press, Nov. 2013.

[32] S. Kamara and C. Papamanthou. Searching Dynamic Encrypted Data in Parallel. In *FC 2013*, 2013.

[33] D. M. Kays. Reasons to "friend" electronic discovery law. *Franchise Law Journal*, 32(1), 2012.

[34] V. Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In B. K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 136–155. Springer, Dec. 2005.

[35] V. Kolesnikov and R. Kumaresan. Improved secure two-party computation via information-theoretic garbled circuits. In I. Visconti and R. D. Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 205–221. Springer, Sept. 2012.

[36] 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 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, July 2008.

[37] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In G. Tsudik, editor, *FC 2008*, volume 5143 of *LNCS*, pages 83–97. Springer, Jan. 2008.

[38] 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.

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

[40] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[41] T. Moataz and A. Shikfa. Boolean symmetric searchable encryption. In *ASIACCS 2013: 8th ACM Symposium on Information, Computer and Communications Security*, 2013.

[42] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, Jan. 2005.

[43] J. E. Pace III. Testing the security blanket: An analysis of recent challenges to stipulated blanket protective orders. *Antitrust Magazine*, 19(3), 2005.

[44] V. Pappas, M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Private search in the real world. In *ACSAC '11*, pages 83–92, 2011.

[45] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP '11*, pages 85–100. ACM, 2011.

[46] M. O. Rabin. How to exchange secrets by oblivious transfer. In *Technical Report TR-81*. Aiken Computation Laboratory, Harvard University, 1981.

[47] M. Raykova, B. Vo, S. Bellovin, and T. Malkin. Secure anonymous database search. In *CCSW 2009.*, 2009.

[48] P. Rogaway. *The round complexity of secure protocols*. PhD thesis, Massachusetts Institute of Technology, 1991.

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

[50] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.

[51] J. K. Yan Huang, David Evans and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

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

[53] 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
REPRESENTING QUERY & POLICY

Encoding a query. In our system, a query is represented as a Bloom filter. This filter contains all the relevant columns and operations, and search terms and conditions. For example, consider the following query:

```
SELECT id WHERE fname = ALICE AND dob <= 1975-1-1
AND CONTAINED_IN(notes1, engineer)
```
(2)

The bloom filter will contain the following:

- `fname`, `fname:=`, `fname:ALICE`, `fname:=:ALICE`
- `dob`, `dob:<=`, `dob:1975-1-1`, `dob:<=:1975-1-1`
- `notes1`, `notes1:contained_in`, `notes1:engineer`, `notes1:contained_in:engineer`

Policy circuit. The current implementation provides a parser for any policy that can be represented as a monotone DNF where each variable indicates whether some policy condition (BF keyword) belongs to the input BF representing a query as described above; if the formula output is true, then the client's query is disallowed. For example, a policy may disallow a query if it contains an equality check on `fname` with value `ALICE` and a range in `dob`. In this case, the policy circuit is a simple formula V_1 AND V_2, where the variable V_1 is true if the input BF contains `fname:=:ALICE`, and V_2 is true if the filter contains `dob:<=`. Indeed, query (2) above will be disallowed.

We believe that this provides a wide coverage of policies. For example, our parser also supports a policy that allows only *range* operation on `fname`, indirectly. One technical issue is that we do not want to allow any false approval of a query that fails the policy (though a tunable small probability of false rejection of a good query is acceptable), but the Bloom filters allow no false negatives. We can fix this issue by adding keywords representing absence column, or column operators

to the BF. In the example above the system adds the following keywords:

- `NOT:fname:range, NOT:dob:=, NOT:notes1:stem, NOT:lname, NOT:zip, NOT:marital_status` ....

Now, the aforementioned policy is equivalent to one that disallows queries if the corresponding the BF contains `fname` and `NOT:fname:range`. If the check succeeds, then the query is disallowed. Likewise, a policy allowing only equality operation on `dob` will check if the filter has `dob` and `NOT:dob:=`. In addition, the policy can now disallow queries that do not contain an equality on `dob` column or that do not contain `lname`. More importantly, the policy can now enforce that the query must have `lname` value if `fname` was present.

APPENDIX B
ONE-CASE INDISTINGUISHABILITY

Here, we give a formal definition of one-case indistinguishability. Since our system realizes the ideal functionality $\mathcal{F}_{db}$, the definitions concern only input/output behavior and the leakage profile L.

The distribution E discussed in Section V-B with δ is defined as follows:

> Let $(D_0, q, \mathbf{r})$ be a database, a query and a record as specified in Section V-B. Choose a record in D_0 uniformly at random and replace it with $\mathbf{r}$. Let D_1 be the modified database. Choose a bit $b \in \{0, 1\}$ according to the following distribution:
>
> $$\Pr[b = 1] = \delta, \quad \Pr[b = 0] = 1 - \delta.$$
>
> Run $\mathcal{F}_{db}$, calling Init with (D_0, P), and Query with q. Let v be the leakage to the index server. Output (b, v).

We show that our system satisfies one-case indistinguishability. Note that the initial leakage is none, and therefore, we only need to consider the query leakage which is the query pattern and the tree search pattern. This implies that we only need to consider the tree search pattern since the same query is considered in the experiment. Observe that the newly introduced record $\mathbf{r}$ is equivalent to adding a random paths in terms of the tree search pattern. Therefore, it suffices to focus on the number of added random paths. In particular, let D^+ be defined as follows:

$$x \leftarrow \mathcal{D}; \quad \text{output } (x + 1).$$

Now, consider a following game X:

> Choose a bit $b \in \{0, 1\}$ such that $\Pr[b = 1] = \delta$ and $\Pr[b = 0] = 1 - \delta$. If $b = 0$, let $x \leftarrow \mathcal{D}$; otherwise let $x \leftarrow \mathcal{D}^+$. Output (b, x).

Now, we show that for any x, it holds that

$$\Pr_X[b = 1 |\ x] \le 2\delta.$$

We show this by using case analysis:

- When $x \le 1$, it never comes from $\mathcal{D}^+$, so the inequality trivially holds.
- When $2 \le x \le \alpha - 1$, it holds that
$$\Pr[b = 1 |\ x] = \frac{\Pr[X = (1, x)]}{\Pr[x]} = \frac{\delta/\alpha}{\delta/\alpha + (1 - \delta)/\alpha} = \delta.$$
- When $x \ge \alpha$, it holds that
$$\begin{aligned} \Pr[b = 1 |\ x] &= \frac{\Pr[X = (1, x)]}{\Pr[x]} \\ &= \frac{\delta \cdot (1/\alpha) \cdot 1/2^{x-\alpha}}{\delta \cdot (1/\alpha) \cdot 1/2^{x-\alpha} + (1 - \delta) \cdot (1/\alpha) \cdot 1/2^{x-\alpha+1}} \\ &= \frac{\delta}{\delta + (1 - \delta)/2} = \frac{2\delta}{1 + \delta} \le 2\delta. \end{aligned}$$