Secure Computation Towards Practical Applications

Fernando Krell Loy

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

Secure Computation Towards Practical Applications

Fernando Krell Loy

Secure multi-party computation (MPC) is a central area of research in cryptography. Its goal is to allow a set of players to jointly compute a function on their inputs while protecting and preserving the privacy of each player's input.

Motivated by the huge growth of data available and the rise of global privacy concerns of entities using this data, we study the feasibility of using secure computation techniques on large scale data sets to address these concerns.

An important limitation of generic secure computation protocols is that they require at least linear time complexity. This seems to rule out applications involving big amounts of data. On the other hand, specific applications may have particular properties that allow for ad-hoc secure protocols overcoming the linear time barrier. In addition, in some settings the full level of security guaranteed by MPC protocols may not be required, and some controlled amount of privacy *leakage* can be acceptable.

Towards this end, we first take a theoretical point of view, and study whether sublinear time RAM programs can be computed securely with sublinear time complexity in the two party setting. We then take a more practical approach, and study the specific scenario of private database querying, where both the server's data and the client's query need to be protected. In this last setting we provide two private database management systems achieving different levels of *efficiency*, *functionality*, and *security*. These three results provide an overview of this three-dimensional trade-off space.

For the above systems, we describe formal security definitions and establish mathematical proofs of security. We also take a practical approach providing an implementation of the systems and experimental analysis of their efficiency.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work would have not been possible without the helpful guidance of my adviser Professor Tal G. Malkin. I'm extremely grateful to her in trusting and admitting me as one of her students at Columbia University, and for being part of her research group and projects.

I would also like to thank

- the members of the committee Allison Bishop, Moti Yung, Vlad Kolesnikov, and Dov Gordon for their feedback and time.
- my coauthors Steven M. Bellovin, Seung G. Choi, Gabriela Ciocarlie, Ben Fisch, Ashish Gehani, Wesley George, Dov Gordon, Jonathan Katz, Angelos Keromytis, Vlad Kolesnikov, Abishek Kumarasubramanian, Tal Malkin, Vasilis Pappas, Mariana Raykova, Yevgeniy Vahlis, Binh Vo, for the hard work done in the research projects presented in this document.
- Professors Xi Chen, Rocco Servedio, Allison Bishop and Mihalis Yannakakis for their great courses I was lucky to take at Columbia University.
- all PhD students and friends I met during my studies. Special thanks to my office-mates Igor Oliveira and Clément Cannone for their endless patience and unconditional support in times of stress.
- CS department staff members Jessica Rosa and Remi Moss for their help in solving administrative problems.
- my undergrad adviser Professor Alejandro Hevia at University of Chile for introducing and enlightening me with the world of cryptography, and pointing me in the right direction.

- Finally, I thanks my parents Irene and Edgardo, my brothers Rodrigo and Javier, and specially my wife Valentina and daughter Amanda for their unrestricted support and guidance.

External Support.

The first four years of my doctoral studies were mainly supported by CONICYT-Becas Chile, Chilean Government scholarship program.

The work of Chapters 4 and 5 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.

Chapter 1

Introduction

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 have become more acute. An example is the revelation of the U.S. Government's data collection programs, which rekindle the privacy debate.

The privacy issue can be mitigated by the use of standard encryption of the data generated, at the cost of hindering the applications' functionality. The ability of searching over encrypted data provides critical capabilities for database management systems that need to guarantee privacy protection for data and queries: examples include information sharing between law enforcement agencies and electronic discovery in private databases, e.g., log files, bank records, during lawsuits; private queries to census data, police investigations using data from automated license plate readers [LAp].

Searching over encrypted data can be thought of a special case of *secure multi-party computation protocols.* A secure computation protocol allows a set of parties to evaluate a (possibly randomized) functionality, while keeping each party's input private from the rest of the parties. In the case of searching over private data, one party would input its private data, while the other party would input the token or query to be searched.

An important aspect of secure multi-party computation protocols is that their running time *must* be at least linear in the size of the parties' inputs. In particular, each party must at least touch every single bit of its input. This lower bound is in fact necessary for full security since otherwise a participant may infer some information about other parties' input by looking which of its inputs bits have been accessed.[1] Consider, for example, a database management system (DBMS) where a client perform queries on a server administered database. The database nodes or records untouched by the server during the evaluation of a query reveal, for example, that those nodes or records do not satisfy the query, disclosing non-trivial information about the client's query to the server.

The above linear lower bound seems to rule out many applications of interest. Indeed, as mentioned above, today's applications involve high volumes of data, and even a simple linear scan on the data may be prohibitively expensive.

The purpose of this thesis is to overcome the above limitation by taking advantage of some key points common to many applications:

- On simple problems such as binary search, graph queries, table look-ups, and many others, the exact same operation is computed repeatedly over the same preprocessed data.
- Data does not vary significantly after a few operations. This is the case in many database scenarios.
- In some settings, full security is not worth the cost of expensive secure computation. However, a somewhat weaker notion of security is still necessary. By allowing a limited amount of *privacy leakage*, applications may gain significantly in terms of efficiency and functionality.
- A third party, such as a cloud provider, is available (not necessarily considered trusted).

Having in mind the above insights, this work aims at answering the following questions: Is it possible to take advantage of the above mentioned assumptions and available cryptographic tools in order to circumvent the linear time barrier of secure computation? How

[1] We are assuming that the parties are computing a non-trivial function where every input bit has non-zero influence over the output of the function.

efficient can we be? What kind of functionalities can we support? How much do we need to leak?

In this work we take a practical approach to address the above questions, and provide three specific protocols that shed light on the matter. These three different approaches give new trade-off marks by varying the level of *privacy*, *functionality*, and *efficiency* supported.

The strongest result (security-wise) presented in this work takes advantage of the fact that in many client-server applications, the server's data is mostly static, and the client repeats similar queries on small-size inputs chosen anew each time. We present a protocol for secure two-party computation that is *sublinear* (in an amortized sense) in the size of the input for functionalities that can be computed in sublinear time (insecurely) over a Random Access Machine. In addition to the algorithmic construction, we provide an implementation of our approach. Our experimental results show that our asymptotic sublinear running time outperforms the state-of-the-art generic secure two-party computation schemes for relatively small data sizes for today's standard. The results of the work in this direction is presented in Chapter 3.

In spite of the novelty and theoretical interest of the above result, the protocol does not achieve the level of efficiency needed in most practical applications. This is due to large constants and degree of polylogarithmic factors behind the "big O" notation in terms of computation and communication complexity. In our next result, we move towards efficiency in the trade-off space. Our goal is to build a new scheme for the more specific application of private search on databases that is highly efficient, rich in functionality, and at the same time assures strong levels of query and data privacy. To this end, 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 these 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.

The output of each privacy-critical subproblem translates in our system to leakage in terms of access pattern to database records, and database data structures. In addition, we work in the split-server model, in which an honest-but-curious third party maintains an encrypted version of the database, assisting clients' queries. Combining these two approaches allows our system to be exceptionally efficient while protecting the client's privacy from the server. We propose *Blind Seer: A Scalable Private DBMS*. We demonstrate via an implementation of a prototype and experimental results that the query running time of Blind Seer is within a small constant factor to plaintext solutions (MySQL). In particular, we show that many queries can be answered within milliseconds for hundred of millions records databases. In terms of functionality, Blind Seer allows for arbitrary boolean expression, free-text search, negation and ranges client queries. The system also provides efficient and robust access control. Results of this work are presented in Chapters 4 and 5.

Our next and final contribution is motivated by the poor understanding of the access pattern leakage of the schemes of Chapters 4 and 5 (and others). We ask whether a private search scheme can be as efficient and rich in functionality as Blind Seer, without requiring the above leakage to the third party. We build a new scheme that lies in-between the above two constructions in the privacy/efficiency/functionality trade-off space. Our novel private search scheme supports queries in DNF formulas on keywords and removes all important leakage to the third party. We provide a prototype of our system, and show experimental results. This system is presented in Chapter 6.

Organization. In Chapter 2 we present basic concepts and definition required for a clear and self-contained presentation of each contribution. Chapter 3 is dedicated to our protocol for sublinear secure two party computation. The following three chapters are concerned with the Private Search scenario. In particular, the basic construction of the Blind Seer system is presented in Chapter 4. The Blind Seer system is next augmented in Chapter 5 with a stronger security mechanism that prevents attacks from malicious database clients. Chapter 6 presents a new scheme based on Blind Seer that prevents important privacy leakage introduced by Blind Seer. We cover related work on Chapter 7, and conclude this thesis in Chapter 8.

Chapter 2

Technical Background

2.1 Bloom Filters

A Bloom filter (BF) [Blo70] is a simple data structure that facilitates efficient set membership queries. The filter B is an ℓ-bit string initialized with 0^ℓ and associated with a set of h different hash functions $\mathcal{H} = \{H_i : \{0,1\}^* \rightarrow [\ell]\}_{i=1}^{h}$. For an element $\alpha \in \{0,1\}^*$, let $\mathcal{H}(\alpha)$ the sequence of the hash results of α, i.e.,

$$\mathcal{H}(\alpha) = (H_1(\alpha), H_2(\alpha), \ldots, H_h(\alpha)).$$

To add an element α into the filter, we turn on the bits in the positions pointed by the hash result $\mathcal{H}(\alpha)$. To check whether an element β is in the filter, we compute the set $\mathcal{H}(\beta)$ and check if all pointed bits are set. Bloom filters guarantee no false negatives, however they do allow for a tunable false positive rate:

$$\mathsf{fp} = \Big(1 - (1 - \frac{1}{\ell})^{ht}\Big)^h \approx \Big(1 - e^{-\frac{ht}{\ell}}\Big)^h,$$

where t is the number of keywords in the Bloom filter.

Given a false positive rate fp and the number of elements t in the filter the optimal length ℓ of B and the optimal number of hash functions h to use can be approximately computed as :

$$\ell = \left\lceil \frac{t \ln p}{\ln \frac{1}{2^{\ln 2}}} \right\rceil \tag{2.1}$$

$$h = \left\lceil \ln 2 \frac{\ell}{t} \right\rceil \tag{2.2}$$

2.2 Random Access Machines

Let D be a memory array of n entries of size ℓ bits each. Each array element can be accessed in constant time via a read/write instructions. An instruction $I \in (\{\mathsf{read}, \mathsf{write}\} \times \mathbb{N} \times \{0,1\}^{\ell})$ takes the form (write, v, d) ("write data element d in location/address v") or $(\mathsf{read}, v, \perp)$ ("read the data element stored at location v").

In the RAM model, a function $f(x, D)$ (where x is assumed to be "small", and it can be read entirely in constant time) is computed by via a sequence of instructions over the memory array D. Each instruction is given by a "next instruction" function Π that takes as input the current state of the program and the last data read from D, and outputs an instruction I or an special stop instruction together with the final output of the program..

The execution of a RAM program can be viewed as follows:

- Set $\mathsf{state}_{\Pi} = (1^{\log n}, 1^{\ell}, \mathsf{start}, x)$ and $d = 0^{\ell}$. Then until termination do:

  1. Compute $(I, \mathsf{state}'_{\Pi}) = \Pi(\mathsf{state}_{\Pi}, d)$. Set $\mathsf{state}_{\Pi} = \mathsf{state}'_{\Pi}$.
  2. If $I = (\mathsf{stop}, 0, z)$ then terminate with output z.
  3. If $I = (\mathsf{write}, v, d')$ then set $D[v] = d'$.
  4. If $I = (\mathsf{read}, v, \perp)$ then set $d = D[v]$.

The memory array D can grow beyond n entries, so the RAM program may issue write (and then read) instructions for indices greater than n. The space complexity of a RAM program on initial inputs x, D is the maximum number of entries used by the memory array D during the course of the execution. The program's time complexity is the number of instructions issued in the execution as described above. For our application in Chapter 3, we do not want the running time of a RAM program to reveal anything about the input x.

Thus, we will assume that any RAM program has associated with it a polynomial t such that the running time on x, D is exactly $t(\log n, \ell, |x|)$.

2.3 Oblivious RAM

An oblivious-RAM (ORAM) [GO96] scheme is a mechanism that simulates a RAM read/write access to an underlying (virtual) array D via a series of accesses to some (real) array $\tilde{D}$. The key property of the scheme is obliviousness, meaning that no information about the virtual accesses to D is leaked by observation of the real accesses to $\tilde{D}$. An ORAM construction can be used to compile any RAM program into an oblivious version of that program, ensuring that the entity holding the array $\tilde{D}$ learns nothing from the program, excepts its running time.

An ORAM scheme consists of two algorithms OSetup and $\mathsf{OAccess}$ for initialization and execution, respectively. OSetup initializes some state state composed by access parameters param and data structure struct. The second algorithm, $\mathsf{OAccess}$, is used to compile a single read/write instruction I (on the virtual array D) into a sequence of read/write instructions $\tilde{I}_1, \tilde{I}_2, \ldots$ to be executed on (the real array) $\tilde{D}$. The compilation of an instruction I into $\tilde{I}_1, \tilde{I}_2, \ldots,$ can be adaptive; i.e., instruction $\tilde{I}_j$ may depend on the values read in some prior instructions. To capture this, we define an iterative procedure called $\mathsf{doInstruction}$ that makes repeated use of $\mathsf{OAccess}$. Given a read/write instruction I, we define $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I)$ as follows:

- Set $d = 0^\ell$. Then until termination do:

  1. Compute $(\tilde{I}, \mathsf{state}') \leftarrow \mathsf{OAccess}(\mathsf{state}, I, d)$, and set $\mathsf{state} = \mathsf{state}'$.
  2. If $\tilde{I} = (\mathsf{done}, z)$ then terminate with output z.
  3. If $\tilde{I} = (\mathsf{write}, v, d')$ then set $\tilde{D}[v] = d'$.
  4. If $\tilde{I} = (\mathsf{read}, v, \perp)$ then set $d = \tilde{D}[v]$.

If I was a read instruction with $I = (\mathsf{read}, v, \perp)$, then the final output z should be the value “written” at $D[v]$.

Correctness. Let $I_1, \ldots, I_k$ be any sequence of instructions with $I_k = (\mathsf{read}, v, \perp)$, and $I_j = (\mathsf{write}, v, d)$ the last instruction that writes to address v. If we start with $\tilde{D}$ initialized to empty and then run $\mathsf{state}_{\mathsf{oram}} \leftarrow \mathsf{OSetup}(1^\kappa)$ followed by $\mathsf{doInstruction}(I_1), \ldots, \mathsf{doInstruction}(I_k)$, then the final output is d with all but negligible probability.

Security. The security requirement is that for any two equal-length sequences of RAM instructions, the (real) access patterns generated by those instructions are indistinguishable. We state next the standard definition from the literature, which assumes the two instruction sequences are chosen in advance. However, it appears that existing ORAM constructions are secure even if the adversary is allowed to adaptively choose the next instruction after observing the access pattern on $\tilde{D}$ caused by the previous instruction, but this has not been claimed by any ORAM construction in the literature. Formally, let $\mathcal{ORAM} = \langle \mathsf{OSetup}, \mathsf{OAccess} \rangle$ be an ORAM construction and consider the following experiment:

Experiment $\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(\kappa)$:

1. The adversary Adv outputs two sequences of queries $(\mathbf{I}^0, \mathbf{I}^1)$, where $\mathbf{I}^0 = \{I_1^0, \ldots, I_k^0\}$ and $\mathbf{I}^1 = \{I_1^1, \ldots, I_k^1\}$ for arbitrary k.

2. Sample a uniformly random bit b.

3. Run $\mathsf{state} \leftarrow \mathsf{OSetup}(1^\kappa)$; initialize $\tilde{D}$ to empty; and then execute $\mathsf{doInstruction}(\mathsf{state}, I_1^b)$, $\ldots$, $\mathsf{doInstruction}(\mathsf{state}, I_k^b)$ (note that state is updated each time $\mathsf{doInstruction}$ is run). The adversary is allowed to observe $\tilde{D}$ the entire time.

4. Finally, the adversary outputs a guess $b' \in \{0, 1\}$. The experiment evaluates to 1 iff $b' = b$.

Definition 1. *An ORAM construction* $\mathcal{ORAM} = \langle \mathsf{OSetup}, \mathsf{OAccess} \rangle$ *is* access-pattern hiding *if for every* PPT *adversary* Adv *the following is negligible in* κ*:*

$$\left| \Pr\left[\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(1^\kappa, b) = 1\right] - \frac{1}{2} \right|$$

Where the probability is taken over the internal randomness of algorithms Adv*,* OSetup*, and* OAccess.

Time/Space complexity. Existing ORAM constructions have the following complexity for an array of length s: the server's storage is $s \cdot \mathsf{polylog}(s)$; the client's storage is $O(\log s)$; and the (amortized) work required to read/write one entry of the array is $\mathsf{polylog}(s)$.

2.4 Pseudorandom Functions and Generators.

Let k be a secret parameter. Let $F : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^\ell$ be a keyed, polynomial time computable function, and $G : \{0,1\}^k \rightarrow \{0,1\}^{\mathsf{poly(k)}}$ be a stretching, polynomial time computable function.

We say that F is a *pseudorandom function* (PRF) [GGM86] if no probabilistic polynomial time algorithm can distinguish any sequence of $(F(k,x_1),\ldots,F(k,x_n))$ from a uniformly random string of the same length when k is chosen uniformly random from $\{0,1\}^k$. This should hold even if the algorithm can choose n and $(x_1,\ldots,x_n)$ as it wishes adaptively. For notational convenience, hereafter we will use $F_k(x)$ to mean $F(k,x)$.

Similarly, we say that G is *pseudorandom generator* (PRG) [BM84] if no probabilistic polynomial time algorithm can distinguish between a uniformly random string in $\{0,1\}^{poly(k)}$ from $G(s)$ for an uniformly random string $s \in \{0,1\}^k$.

2.5 Semantically Secure Encryption.

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme. We define the security of an encryption scheme against an eavesdropping adversary via following game:

$\mathsf{Game}_{\Pi}^{\mathtt{IND}}(\mathcal{A},\kappa)$:

1. Adversary $\mathcal{A}$ chooses equal length messages m_0, m_1.
2. Run $\mathsf{Gen}(\kappa)$ to derive a key sk, sample a uniformly random bit b, and sends the ciphertext $\mathsf{Enc}_{sk}(m_b)$ to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a decision bit b'.
4. If $b' = b$, the game outputs 1, and otherwise 0.

Let $\mathbf{Adv}_{\Pi}^{\mathtt{IND}}(\mathcal{A},\kappa) = |\Pr[\mathsf{Game}_{\Pi}^{\mathtt{IND}}(\mathcal{A},\kappa) = 1] - \frac{1}{2}|$.

We say that Π is semantically secure if and only if $\mathbf{Adv}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \kappa) < \mathsf{negl}(\kappa)$ [GM84].

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

2.6 Yao's Garbled Circuits

Garbled Circuits (GC). Yao's garbled circuits [Yao82] allow to evaluate Boolean circuits on hidden inputs provided by another party. Let C be a Boolean circuit with n input wires, m gates, and (assume for simplicity) 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 given the input keys $w_i^{b_i}$ and $w_j^{b_j}$, it is possible to recover $w_k^{g(b_i,b_j)}$. 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 from the input 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 (OT). An oblivious transfer (OT) [EGL85; Rab81] 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. For the prototypes of systems described in Chapters 3,4, and 5, we use the Naor-Pinkas protocol [NP01] as a basis and optimize the performance using OT extension [IKNP03] and OT preprocessing [Bea95].

Secure Two-Party Computation. Yao's garbled circuit can be combined with and oblivious transfer protocol to build a constant-round protocol for secure two party computation for any Boolean circuit. One party acts as the garbled circuit generator G and the other as the circuit evaluator E. G builds the garbled circuit and send it to E together with the keys corresponding to G's input. Before E evaluates the garbles circuit, both parties engage in a series of OT execution to transfer the keys corresponding to E's input. The protocol can finish in two possible ways. One possibility is that E sends the computed output keys to G (G gets the output). The other option is to have G to send the key pairs corresponding to every output wire (in which case E gets the output).

Note that the only party that needs to know the circuit is G, and E only needs to known the circuit topology. Moreover, the garbled gate reveals nothing about the actual gate if the output keys are encrypted under a semantically secure encryption scheme. Hence, the protocol allows for a form of private function evaluation in which only G knows the function, while E only knows its topology.

The construction can be optimized in several ways. We made extensive use of the free-XOR technique [KS08a], that allows XOR gates to be evaluates at virtually no cost. We also use the point-and-permute [Rog91; MNPS04] that allows to reduce of cost of evaluating each gate by decripting only one of the four gate's entries. In addition, we use is the row-reduction technique, which decreases the size of the garbled gate, and hence, improving communication bandwidth.

We refer the reader to [BHR12; LP09] for clear description and proof of security of garbled circuits.

2.7 Private Database Search

In the second part of this Thesis we focus on the problem of private search over database. The setting involves three main players. The database owner, whom we usually call the server S, the client or querier C, and a third-party we call the index server whom we denote as IS. The database owner outsources its data to the third party, and gives search capabilities to clients, with the property that the third party learns nothing about the data nor the client's queries, and the client only learns the result set of its queries. This setting was called Outsourced Symmetric Private Information Retrieval in [JJK$^+$13], and it is defined by an algorithm Setup and a procedure Search. The following formal definition allows for a parameter fp denoting false positives in the result set of the client queries.

Definition 2. *Let λ be a security parameter, let $\mathsf{fp} \in [0,1]$ some tunable parameter, and let DB denote a dataset corresponding to a set of documents $\{D_i\}$, each associated with a set of searchable keywords W_i. We define an* OSPIR Scheme *as a pair of interactive algorithms*

- $(\mathsf{params}, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}, \mathsf{fp})$. S *inputs database* $\mathsf{DB} = (D_i, W_i)$, *and gets back* params. IS *gets* EDB.

- $(\mathsf{records}) \leftarrow \mathsf{Search}(\mathsf{params}, \mathsf{EDB}, q)$. S *inputs* params, IS *inputs* EDB *and* C *inputs* q. C *gets* $\mathsf{records}$.

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

Security for these schemes are defined via the standard ideal vs. real world paradigm. However, we defer the security definitions to Chapter 4 and Chapter 6, since the different protocols presented in those chapters allow for different security guarantees.

Part I

Sublinear Secure Two-Party Computation

Chapter 3

Secure Two-Party Computation in Sublinear Amortized Time

3.1 Introduction

Consider the task of searching over a sorted database of n items. Using binary search, this can be done in time $O(\log n)$. Now consider a secure version of this task where a client wishes to learn whether an item is in a sorted database held by a server, with neither party learning anything more. Applying generic secure computation [Yao86; GMW87] to this task, we would begin by expressing the computation as a (binary or arithmetic) circuit of size at least n, resulting in a protocol of complexity $\Omega(n)$. Moreover, (at least) linear time complexity is *inherent*: in any secure protocol for this problem the server must "touch" every entry of the database; otherwise, the server learns information about the client's input by observing which entries of its database were never accessed.

This linear time barrer seems to eliminate the possibility of ever performing practical secure computation over large datasets. However, one may notice two opportunities for improvement:

- Many interesting procedures (such as searching over a sorted database) can be done in *sublinear* time on a random-access machine (RAM). Thus, it might be convenient to have protocols for generic secure computation that are based on the RAM model of computation rather than circuits, as generic protocols do.

- The fact that linear work is inherent for secure computation of any non-trivial function f only applies when f is computed *once*. However, it does not rule out the possibility of doing better, in an amortized sense, when the parties compute the same function *multiple* times.

Inspired by the above, we explore scenarios where secure computation with *sublinear* amortized work is possible. We focus on a setting where a client and server repeatedly evaluate a function f, maintaining state across these executions, with the server's (huge) input D changing only a little between executions, and the client's (small) input x chosen anew each time f is evaluated. (It is useful to keep in mind the concrete application of a client making several read/write requests to a large database D, though our results are more general.) Our main result is:

Theorem 1. *Suppose f can be computed in time t and space s in the RAM model of computation. Then there is a secure two-party protocol for f in which the client and server run in amortized time $O(t) \cdot \mathsf{polylog}(s)$, the client uses space $O(\log(s))$, and the server uses space $s \cdot \mathsf{polylog}(s)$.*

The above holds in the semi-honest adversarial model.

3.1.1 Contributions

We show a generic protocol achieving the above bounds by applying any protocol for secure two-party computation in a particular way to any *oblivious RAM* (ORAM) construction (see section 2.3). This demonstrates the feasibility of secure computation with sublinear amortized complexity. We then explore a concrete, optimized instantiation of our protocol based on the tree-based ORAM construction of Shi et al. [SCSL11], and using Yao's garbled-circuit approach [Yao86] for the secure two-party computation. We chose the tree-based ORAM construction of Shi et al. because of its simplicity and its poly-logarithmic *worst-case* complexity (as opposed to other schemes that only achieve this in an amortized sense), it requires small client state, and its time complexity in practice (i.e., taking constant factors into account) is among the best known. We can also use the same techniques on the more recent tree-based construction in [SvDS+13] achieving better efficiency. (In Section 3.8 we briefly discuss why we expect other schemes to yield worse overall performance for our application.) We chose Yao's garbled-circuit approach for secure computation since several recent results [HEKM11; Mal11] show that it is both quite efficient and can scale to handle circuits with tens of millions of gates. When combining these two schemes, we apply a number of optimizations to reduce the sizes of the circuits that need to be evaluated using generic secure computation.

We implemented this optimized protocol, and evaluated it for the task of binary search. For small datasets our protocol is slower than standard protocols for secure computation, but our protocol outperforms the latter for databases containing more than 2^{18} entries.

3.1.2 Background

Ostrovsky and Shoup [OS97] observed that ORAM and secure computation can be combined, though in a different context and using a different approach. Specifically, they consider a (stateless) client storing data on *two* servers that are assumed not to collude. They focus on *private storage* of the data belonging to the client, rather than secure computation of a function over inputs held by a client and server as we do here.

They describe a protocol where the user stores its data according to any ORAM mechanism, using one server to simulate the ORAM server, and sharing the ORAM client state across both servers. Now a read or update can be executed by having the two servers apply generic secure two party computation to privately execute the appropriate ORAM instructions, communicating through the user (with the result being an updated ORAM storage, and updated shared client state).

Our generic protocol follows directly from this idea, although conceptually there are some differences in context. In particular, their setting has three parties, and the data all belongs to one trusted party (the user) storing it in untrusted remote locations (the two servers). In our context, we have two parties (a client and server), and the server stores *its own data* obliviously, according to the ORAM protocol, so as to allow secure computation on both parties' inputs.

Damgård et al. [DMN11] also observe that ORAM can be used for secure computation.

In their approach, which they only briefly sketch, players share the *entire* (super-linear) state of the ORAM, in contrast to our protocol where the client maintains only logarithmic state. They make no attempt to optimize the concrete efficiency of their protocol, nor do they offer any implementation or evaluation of their approach.

Though the above two works have a flavor similar to our own, our work is the first to explicitly point out that ORAM can be used to achieve secure two-party computation with sublinear complexity (for functions that can be computed in sublinear time on a RAM).

Oblivious RAM was introduced in [GO96], and in the past few years several improved constructions have been proposed (c.f. [WS08; WSC08; PR10a; GM11; GMOT12; KLO12; SCSL11; SSS12]). We refer the reader to [SCSL11; SSS12] for further discussion and pointers to the sizeable literature on this topic.

3.2 Overview

Our starting point is the ORAM primitive, introduced in [GO96], which allows a client (with a small memory) to perform RAM computations using the (large) memory of a remote untrusted server. At a high level, the client stores encrypted entries on the server, and then emulates a RAM computation of some function by replacing each read/write access of the original RAM computation with a series of read/write accesses such that the actual access pattern of the client remains hidden from the server.

The above suggests a method for computing $f(x, D)$ for any function f defined in the random-access model of computation, where the client holds (small) input x and the server holds (large) input D: store the memory array used during the computation on the server, and have the client access this array using an ORAM scheme. This requires an (expensive) pre-processing phase during which the client and server initialize the ORAM data structure with D; after this, however, the client and server can repeatedly evaluate $f(x_i, D)$ (on different inputs $x_1, \ldots$ of the client's choice) very efficiently. Specifically, if f can be evaluated in time t and space s on a RAM, then each evaluation of f in this client/server model now takes (amortized) time $t \cdot \mathsf{polylog}(s)$.

The above approach, however, only provides "one-sided security," in that it ensures privacy of the client's input against the server; it provides no security guarantees for the server against the client! We can address this by having the parties compute the next ORAM instruction "inside" a (standard) secure two-party computation protocol, with the intermediate state being shared between the client and server. The resulting ORAM instruction is output to the server, who can then read/write the appropriate entry in the ORAM data structure that it stores, and incorporate the result (in case of a read operation) in the shared state. The key observations here are that (1) it is ok to output the ORAM instructions to the server, since the ORAM itself ensures privacy for the client; thus, secure computation is needed only to determine the next instruction that should be executed. Moreover, (2) each computation of this "next-instruction function" is performed on *small* inputs whose lengths are *logarithmic* in s and independent of t: specifically, the inputs are just (shares of) the current state for the RAM computation of f (which we assume to have size $O(\log s)$, as is typically the case) and (shares of) the current state for the ORAM itself (which has size $O(\log s)$). Thus, the asymptotic work for the secure computation of f remains unchanged.

For our optimized construction, we rely on the specific ORAM construction of Shi et al. [SCSL11], and optimized versions of Yao's garbled-circuit protocol. We develop our concrete protocol with the aim of minimizing our reliance on garbled circuits for complex functionalities. Instead, we perform local computations whenever we can do so without losing security. For example, we carefully use encryption scheme where block-cipher computations can be done locally, with just an XOR computed via secure computation. For the parts of our protocol that do utilize generic secure computation, we rely on garbled-circuit optimization techniques such as the free-XOR approach [KS08a], oblivious-transfer extension [IKNP03], and pipelined circuit execution [HEKM11]. We also use precomputation (e.g., [Bea95]) to push expensive computations to a preprocessing stage. Our resulting scheme only requires simple XOR operations for oblivious-transfer computations in an online stage, while exponentiations and even hashing can be done as part of preprocessing.

3.3 Secure Computation for RAM machines

We focus on the setting where a server holds a (large) database D and a client wants to repeatedly compute $f(x, D)$ for different inputs x; moreover, f may also change the contents of D itself. We allow the client to keep (short) state between executions, and the server will keep state that reflects the (updated) contents of D.

For simplicity, we focus only on the two-party (client/server) setting in the semi-honest model but it is clear that our definitions can be extended to the multi-party case with malicious adversaries.

Definition of security. We use a standard simulation-based definition of secure computation [Gol04], comparing a real execution to that of an ideal (reactive) functionality F. In the ideal execution, the functionality maintains the updated state of D on behalf of the server. We also allow F to take a description of f as input (which allows us to consider a single ideal functionality).

The real-world execution proceeds as follows. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, and the client and server then run protocol Π_f (with the client using input init and the server using input D) that ends with the client and server each storing some state that they will maintain (and update) throughout the subsequent execution. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client; the client and server then run protocol Π_f (with the client using its state and input x_i, and the server using its state) with the client receiving output out_i. The client sends out_i to $\mathcal{Z}$, thus allowing adaptivity in $\mathcal{Z}$'s next input selection x_{i+1}. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. At this time, an honest player simply terminates execution; a corrupted player sends its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$ and some fixed value κ for the security parameter, we let $\textsc{real}_{\Pi_f,\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution in the real world.

In the ideal world, we let F be a trusted functionality that maintains state throughout the execution. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, which the server in turn sends to F. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client who sends this value to F. The trusted functionality then computes

$$(\mathsf{out}_i, D^{(i+1)}) \leftarrow f(x_i, D^{(i)}),$$

and sends out_i to the client. (Note the server does not learn anything from the execution, neither about out_i nor about the updated contents of D.) The client ends out_i to $\mathcal{Z}$. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. The honest player simply terminates execution; the corrupted player may send an arbitrary function of its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$, some fixed value κ for the security parameter, and some algorithm $\mathcal{S}$ being run by the corrupted party, we let $\text{IDEAL}_{F,\mathcal{S},\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution.

Definition 3. *We say that* protocol Π_f securely computes f *if there exists a probabilistic polynomial-time ideal-world adversary $\mathcal{S}$ (run by the corrupted player) such that for all non-uniform, polynomial-time environments $\mathcal{Z}$ there exists a negligible function* negl *such that*

$$\left|\Pr\left[\text{REAL}_{\Pi_f,\mathcal{Z}}(\kappa) = 1\right] - \Pr\left[\text{IDEAL}_{F,\mathcal{S},\mathcal{Z}}(\kappa) = 1\right]\right| \leq \mathsf{negl}(\kappa).$$

3.4 Generic Construction

In this section we present our generic solution for achieving secure computation with sublinear amortized work, based on any ORAM scheme and any secure two-party computation (2PC) protocol. While our optimized protocol (in Section 3.6) is more efficient, this generic protocol demonstrates theoretical feasibility and provides a conceptually clean illustration of our overall approach.

Recall from section 2.3 that the underlying ORAM is defined by two algorithms OSetup and $\mathsf{OAccess}$. The first represents the initialization algorithm, which establishes the client's initial state and can be viewed as also initializing an empty array that will be used as the main ORAM data structure. This algorithm takes as input κ (a security parameter), s (the length of the virtual array being emulated), and ℓ (the length of each entry in both the virtual and actual arrays). The second algorithm $\mathsf{OAccess}$ defines the actual ORAM

Secure initialization protocol

Input: The server has an array D of length n.

Protocol:

1. **Initialization.** The participants run a secure computation of $\mathsf{OSetup}(1^\kappa, 1^s, 1^\ell)$, which results in each party receiving a secret share of the initial ORAM state. We denote this by $[\mathsf{state}]$.

2. **Insertion.** For $v = 1, \ldots, n$ do

   (a) The server sets $I = (\mathsf{write}, v, D[v])$ and secret-shares I with the client. Denote the sharing by $[I]$.

   (b) The parties run $([\mathsf{state}'], [\perp]) \leftarrow \mathsf{doInstruction}([\mathsf{state}], [I])$ (see Figure 3.3), and set $[\mathsf{state}] = [\mathsf{state}']$.

Figure 3.1: Secure initialization protocol π_{Init}.

functionality, namely the process of mapping a virtual instruction I to a sequence of real instructions $\{\hat{I}_i\}_i$. Algorithm $\mathsf{OAccess}$ takes as input (1) the current ORAM state state, (2) the virtual instruction I being emulated, and (3) the last value d read from the ORAM array, and outputs (1) an updated ORAM state state' and (2) the next instruction $\hat{I}$ to run.

With the above in place, we can now define our protocol for secure computation of a function f over an input x held by the client (and assumed to be small) and an array $D \in (\{0,1\}^\ell)^n$ held by the server (and assumed to be large). We assume f is defined in the RAM model of computation in terms of a next-instruction function Π which, given the current state and value d (that will always be equal to the last-read element), outputs the next instruction and an updated state (see section 2.2). We let s denote a bound on the number of memory cells of length ℓ required by this computation (including storage of D in the first n positions of memory). Our protocol proceeds as follows:

1. The parties run a secure computation of OSetup. The resulting ORAM state state is shared between the client and server.

2. The parties run a secure computation of a sequence of (virtual) write instructions that insert each of the n elements of D into memory. The way this is done is described below.

Secure evaluation protocol π_f

Inputs: The server has array $\tilde{D}$ and the client has input $n, 1^\ell$, and x. They also have shares of an ORAM state, denoted $[\mathsf{state}]$.

Protocol:

1. The client sets $\mathsf{state}_\Pi = (n, 1^\ell, \mathsf{start}, x)$ and $d = 0^\ell$ and secret-shares both values with the server; we denote the shared values by $[\mathsf{state}_\Pi]$ and $[d]$, respectively.
2. Do:
   (a) The parties securely compute $([\mathsf{state}'_\Pi], [I]) \leftarrow \Pi([\mathsf{state}_\Pi], [d])$, and set $[\mathsf{state}_\Pi] = [\mathsf{state}'_\Pi]$.

   (b) The parties run a secure computation to see if $\mathsf{state}_\Pi = (\mathsf{stop}, z)$. If so, break.

   (c) The parties execute $([\mathsf{state}'], [d']) \leftarrow \mathsf{doInstruction}([\mathsf{state}], [I])$. They set $[\mathsf{state}] = [\mathsf{state}']$ and $[d] = [d']$.
3. The server sends (the appropriate portion of) its share of $[\mathsf{state}_\Pi]$ to the client, who recovers the output z.

Output: The client outputs z.

Figure 3.2: Secure evaluation of a RAM program defined by next-instruction function Π.

3. The parties compute f by using secure computation to evaluate the next-instruction function Π. This generates a sequence of (virtual) instructions, shared between the parties, each of which is computed as described below.
4. When computation of f is done, the state associated with this computation (state_Π) encodes the output z. The server sends the appropriate portion of its share of state_Π to the client, who can then recover z.

See Figures 3.1 and 3.2 (Initialization and evaluation protocols) for the secure initialization and secure computation of the RAM next-instruction. In the figures, we let $[v]$ denote a secret-sharing of a value v between the two parties. It remains to describe how a single virtual instruction I (shared between the two parties) is evaluated. This is done as follows (also see Figure 3.3):

1. The parties use repeated secure computation of $\mathsf{OAccess}$ to obtain a sequence of real instructions $\hat{I}_1, \ldots$. Each such instruction $\hat{I}$ is revealed to the server, who executes the instruction on the ORAM data structure that it stores. If $\hat{I}$ was a read instruction, then the value d that was read is secret-shared with the client.

2. After all the real instructions have been executed, emulation of instruction I is complete. If I was a read instruction, then the (virtual) value d' that was read is secret-shared between the client and server.

The key point to notice is that *each secure computation that is invoked is run only over* **small** *inputs.* This is what allows the amortized cost of the protocol to be sublinear.

The following summarizes our main theoretical result.

Theorem 2. *If an ORAM construction and a 2PC protocol secure against semi-honest*

The $\mathsf{doInstruction}$ subroutine

Inputs: The server has array $\tilde{D}$, and the server and client have shares of an ORAM state (denoted $[\mathsf{state}]$) and a RAM instruction (denoted $[I]$).

1. The server sets $d = 0^\ell$ and secret shares this value with the client; we denote the shared value by $[d]$.
2. Do:
   - (a) The parties securely compute $([\mathsf{state}'], [\hat{I}]) \leftarrow \mathsf{OAccess}([\mathsf{state}], [I], [d])$, and set $[\mathsf{state}] = [\mathsf{state}']$.
   - (b) The parties run a secure computation to see if $\hat{I} = (\mathsf{done}, z)$. If so, set $[d] = [z]$ and break.
   - (c) The client sends its share of $[\hat{I}]$ to the server, who reconstructs $[\hat{I}]$. Then:
     - i. If $\hat{I} = (\mathsf{write}, v, d')$, the server sets $\tilde{D}[v] = d'$ and sets $d = d'$.
     - ii. If $\hat{I} = (\mathsf{read}, v, \perp)$, the server sets $d = \tilde{D}[v]$.
   - (d) The server secret-shares d with the client.

Output: Each player outputs its shares of state and d.

Figure 3.3: Subroutine for executing one RAM instruction.

adversaries are used, then our protocol securely computes f against semi-honest adversaries. Furthermore, if f can be computed in time t and space s on a RAM, then our protocol runs in amortized time $O(t) \cdot \mathsf{polylog}(s)$, the client uses space $O(\log(s))$, and the server uses space $s \cdot \mathsf{polylog}(s)$.

We comment that if the underlying secure-computation is secure against *malicious* parties, then a simple change to our protocol will suffice for obtaining security against malicious parties as well. We simply change the outputs of all secure computations to include a signature on the outputs described above (using a signing key held by the other party), and we modify the functions used in the secure-computation such that they verify the signature on each input before continuing. We leave the proof of this informal claim to future work. We note that we cannot make such a claim for our more efficient, concrete solution presented in Section 3.6.2.

3.5 Security of the Generic Construction

We now prove that the construction presented in the previous section is a secure MPC protocol according to Definition 3.

At the very high level, security against the client holds because he only manipulates the data protected by secret sharing and MPC; the server additionally sees plaintext ORAM instructions – but they do not reveal anything by the ORAM guarantee. (ORAM security [GO96] is proven in the non-adaptive setting only. However, our security simulation goes through, since the adaptive input and function selection by $\mathcal{Z}$ does not depend on protocol message view (that is, $\mathcal{Z}$ receives the view from the adversary after the stop message), and hence the simulators can query the ORAM functions *after* $\mathcal{Z}$ had completed the adaptive selection.)

We start with the descriptions of the client simulator S_{cl}, who interacts with $\mathcal{Z}$. In i-th computation, S_{cl} receives x_i and $y_i = f(x_i, D^{(i-1)})$, stores them, and postpones it simulation until he receives the special end symbol from $\mathcal{Z}$. At this point, S_{cl} outputs entire simulation, as follows:

Pre-processing. S_{cl} simulates pre-processing by generating an appropriate number of random ORAM state shares:

1. S_{cl} runs the $\mathsf{OSetup}(1^\kappa)$ functionality to obtain an initial state for the ORAM, and generates a uniformly random share $[\mathsf{state}]_c$ for the client.
2. Let $I_1, \ldots, I_{|D^{(0)}|}$ be instructions of the form $(\mathsf{write}, v, \bar{0})$ for $1 \leq v \leq |D^{(0)}|$. S_{cl} sequentially applies $\mathsf{OAccess}(state, I_i)$ for each i, updating the ORAM state and generating a uniformly random share of it after each execution.

Computation. For each RAM f to be evaluated, S_{cl} will simulate its execution evaluating the same number of instructions of the form $(\mathsf{read}, 0, \perp)$ using $\mathsf{OAccess}$. Denote by $|f|$ the execution length of RAM f. Then, for each functionality f:

1. S_{cl} starts with a previously generated share $[\mathsf{state}]_c$ of the ORAM state that was generated during the pre-processing, or during the last computation.
2. Let $I_1, \ldots, I_{|f|}$ be instructions of the form $(\mathsf{read}, 0, \perp)$. As in the pre-processing phase, S_{cl} sequentially runs $\mathsf{OAccess}$ on $I_1, \ldots, I_{|f|}$, along with the current ORAM state. After each instruction is evaluated, $\mathsf{OAccess}$ returns an updated state, and S_{cl} generates a new uniformly random state share $[\mathsf{state}]'_c$.
3. The output reconstruction is simulated by opening to y_i the secret sharing of the output.

The server simulator S_{serv} proceeds similarly to S_{cl}. The notable difference is that the generated view additionally contains the instructions issued by $\mathsf{OAccess}$. Specifically, during pre-processing, $\mathsf{OAccess}$ is used to evaluate instructions of the form $I_j = (\mathsf{write}, v, \bar{0})$ for $1 \leq v \leq |D^{(i-1)}|$. For each such instruction, $\mathsf{OAccess}$ generates a sequence of subqueries $\hat{\mathbf{I}}_j$, which are included in the generated view. Similarly, during the computation of each functionality f, each instruction is converted by $\mathsf{OAccess}$ into a sequence of subqueries. These subqueries are included in the generated view (in addition to the state shares).

It is not hard to see that these simulators produce views indistinguishable from real execution. The reduction to the (non-adaptive) security of ORAM is straightforward, given our prior observation that the simulators produce their output only after the entire sequence of x_i is specified by $\mathcal{Z}$ (and hence the adaptively chosen sequence of x_i can be fed non-adaptively into the ORAM security experiment).

3.6 An Optimized Protocol

In Sections 3.4 and 3.5 we showed that any ORAM protocol can be combined with any secure two-party computation scheme to obtain a secure computation scheme with sublinear amortized complexity. In this section we present a far more efficient and practical scheme, based on instantiating our generic protocol with Yao's garbled circuits and the ORAM construction of Shi et. al [SCSL11]. However, rather than applying the secure computation primitive on the entire ORAM instruction, we deviate from the generic protocol by performing parts of the computation locally, whenever we could do so without violating security. This section describes our scheme, including concrete algorithmic and engineering decisions we made when instantiating our protocol, as well as implementation choices and complexity analysis. In Section 3.7 we present experimental performance results, demonstrating considerable improvement over using traditional secure computation over the entire input (i.e. without ORAM).

3.6.1 The Binary Tree ORAM

We begin with an overview of the ORAM construction of [SCSL11], which is the starting point of our protocol. The main data storage structure used in this scheme is a binary tree with the following properties. To store N data items in the ORAM, we construct a binary tree of height $\log N$, where each node has the capacity to hold $\log N$ data items. Every item stored in the binary tree is assigned to a randomly chosen leaf node. The identifier of this leaf node is appended to the item, and the item, along with its assignment, is encrypted and stored somewhere on the path between the root and its assigned leaf node. To find a data item, the client begins by retrieving the leaf node identifier associated with that item; we will explain how this is done below. He sends the identifier of the leaf node to the server, who then fetches and sends all items along the appropriate path, one node at a time. The client decrypts the content of each node and searches for the item he is looking for.

When he finds it, he removes it from its current node, assigns it a new leaf identifier chosen uniformly at random and inserts the item at the root node of the tree. He then continues searching all nodes along the path in order to prevent the server from learning where he found the item of interest.

Since the above look-up process will work only while there is room in the root node for new incoming items, the authors of [SCSL11] devise the following load balancing mechanism to prevent nodes from overflowing. After each ORAM access instruction, two nodes are chosen at random from each level of the tree. One arbitrary item is evicted from each of these nodes, and is inserted in the child node that lies on the path towards its assigned leaf node. While the server will learn which nodes were chosen for this eviction, it should not learn which children receive the evicted items. To hide this information, the client insert encrypted data in *both* of the two child nodes, performing a "dummy" insertion in one node, and a real insertion in the other. In a more recent scheme [SvDS$^+$13], eviction is done across the same path retrieved earlier, pushing down element as they can.

All that remains to describe is how the client recovers the leaf identifier associated with the item of interest. The number of such identifiers is linear in the size of the database, so storing the identifiers on the client side is not an option. The solution is to store these assignments on the server, recursively using the same type of binary trees. A crucial property which makes this solution possible is that an item can store more than a single mapping. If an item stores r mappings, then the total number of recursively built trees is $\log_r N$. The smallest tree will have very few items, and can thus be stored by the client. As an example, let the largest tree contain items with virtual addresses $v_1^{(1)}, \ldots, v_N^{(1)}$ that are assigned leaf identifiers $L_1^{(1)}, \ldots, L_N^{(1)}$. Then the tree at level 2 has $\frac{N}{r}$ items with virtual addresses $v_1^{(2)}, \ldots, v_{\frac{N}{r}}^{(2)}$, where the item with virtual address $v_j^{(2)}$ contains mappings $(v_i^{(1)}, L_i^{(1)})$ for $(j-1)r < i \leq jr$. With this modification, an ORAM look-up consists of a series of look-ups, one in each of these trees, beginning with the smallest tree. In particular, given a virtual address v for a database query, the client derives the look-up values that he needs to use in tree i by computing $v^{(i)} = \lfloor \frac{v}{r^i} \rfloor$ for $0 \leq i \leq \log_r N$. Having these values the client starts with a look-up in the smallest tree for value $v^{(\log_r N)}$. He retrieves $L^{(\log_r N)}$ from his local memory and finds in it the mapping $(v^{(\log_r N-1)}, L^{(\log_r N-1)})$. Now he looks for

$v^{(\log_r N-1)}$ in the next smallest tree using leaf identifier $L^{(\log_r N-1)}$. This process continues until the client retrieves the real database item at address v from the largest tree at the top level of the recursion. In each tree, the accessed item is assigned a new leaf node at random, and the item is inserted back in the tree's root node. In addition, its mapping is updated in the tree below to record its new leaf node.

The intuition for the security of this scheme can be summarized as follows. Every time the client looks up item v_i, he assigns it a new leaf node and re-inserts it at the root. It follows that the paths taken to find v_i in two different look-ups are independent of one another, and cannot be distinguished from the look-up of any other two nodes. During the eviction process, a node is just as likely to accept a new item as it is to lose an item. Shi et al. prove in their work that with buckets of size $O(\log(MN/\delta))$ the probability that a bucket will overflow after M ORAM instructions is less than δ. It follows that with a bucket size of $O(\log N)$, the probability of overflow is negligible in the security parameter. However, as we shall see below, the precise constant makes a big difference, both in the resulting security and in the efficiency of the scheme.

3.6.2 High Level Protocol

As above, we assume a database of N items, and we allow each item in each recursive level to hold r mappings between virtual addresses and leaf identifiers from the level above. The client and a server perform the following steps to access an item at an address v:

1. The parties have shares v_C and v_S of the virtual address for the query in the database $v = v_C \oplus v_S$.

2. The client and the server run a two party computation protocol to produce shares $v_C^{(1)}, \ldots, v_C^{(\log_r N)}$ and $v_S^{(1)}, \ldots, v_S^{(\log_r N)}$ of the virtual addresses that they will look-up in each tree of the ORAM storage: $\lfloor \frac{v}{r^i} \rfloor = v_C^{(i)} \oplus v_S^{(i)}$ for $0 \le i \le \log_r N$.

3. The client generates random leaf identifiers $\tilde{L}^{(1)}, \ldots, \tilde{L}^{(\log_r N)}$ that will be assigned to items as they are re-inserted at the root.

4. The last tree in the ORAM storage has only a constant number of nodes, each containing a constant number of items. The client and server store shares of the leaf

identifiers for these items. They execute a two party protocol that takes these shares as inputs, as well as the shares $v_C^{(\log_r N)}$ and $v_S^{(\log_r N)}$. The server's output of the secure computation is the leaf value $L^{(\log_r N)}$. The client has no output.

5. For each i such that $\log_r N \geq i \geq 2$:

   (a) The server retrieves the nodes on the path between the root and the leaf $L^{(i)}$ in the i-th tree.

   (b) The parties execute a secure two party protocol. The server's inputs are the nodes recovered above, and the secret share $v_S^{(i-1)}$. The client's input is $v_C^{(i-1)}$. The server receives value $L^{(i-1)}$ as output, which is the value stored at address $v_C^{(i-1)} \oplus v_S^{(i-1)}$, and which lies somewhere along the path to $L^{(i)}$.

   (c) The parties execute a secure two party protocol to update the content of item $v^{(i)}$ with the value of the new leaf identifier $\tilde{L}^{(i-1)}$ that will be assigned to $v_C^{(i-1)} \oplus v_S^{(i-1)}$ in the $i-1$-th tree.

   (d) The parties execute a secure two party protocol to tag item $v^{(i)}$ with it's new leaf node assignment $\tilde{L}^{(i)}$, and to insert $v^{(i)}$ in the first empty position of the root node.

6. For the first level tree that contains the actual items for the database, the server retrieves the nodes on the path between the root and the leaf $L^{(1)}$. The parties execute a secure two party protocol to find item $v = v_C^{(1)} \oplus v_S^{(1)}$. The outputs of the protocol are secret shares of the data $d = d_C \oplus d_S$ found at virtual address v. The server tags v with $\tilde{L}^{(1)}$, and the parties perform another secure protocol to insert v at the first empty spot in the root node.

3.6.3 Optimizations and Implementation Choices

Encryption and Decryption In our protocol description so far, we have left implicit the fact all data stored in the database at the server must be encrypted. Every time a data item is used in the RAM computation, it must first be decrypted, and it must be re-encrypted before it is re-inserted at the root node. In a naive application of our generic solution, the parties would have to decrypt, and later re-encrypt the data item completely inside a Yao circuit, which can be very time consuming. We choose the following encryption scheme, with an eye towards minimizing the computation done inside the garbled circuit: $\mathsf{Enc}(m; r) = (F_K(r) \oplus m, r)$, where F can be any pseudo-random function. The key K is stored by the client, and kept secret from the server. To ensure that encryption and decryption can be efficiently computed inside a garbled circuit, the server sends r to the client in the clear, along with a random r' that will be used for re-encryption. The client computes $F_K(r)$ and $F_K(r')$ *outside* the secure computation. Now the only part of decryption or re-encryption that has to be done inside the garbled circuit is Boolean XOR, which is very cheap.

While this greatly improves the efficiency of our scheme, we note that it has to be done with care: sending the encryption randomness to the client could reveal information about the access pattern of the RAM, and, consequently, about the server's data. The issue arises during the eviction procedure, when a data item is moved from a parent to one of its children. During this process, it is important that neither player learn which child received the evicted data; the construction of Shi et al. [SCSL11] has the client touch both children, writing a dummy item to one of the two nodes, and the evicted item to the other node, thereby hiding from the server which node received the real item. In our case, this must be hidden from the client as well, which is ensured by performing the operation inside a secure computation. However, the exact way in which randomness is assigned to ciphertext has a crucial effect on security. For example, suppose the server sends randomness r_1 and r_2 to be used in the re-encryption, and our operation is designed so that r_1 is always used for encrypting the dummy item and r_2 is always used for the real item. The client can then keep track of the real item by waiting to receive r_2 for decryption in the future! We must therefore design the re-encryption operation so that randomness is associated with a node in the tree rather than the content of the ciphertext. Then, r_1 is always used in the left child, and r_2 in the right, independent of which node receives the real item and which receives the dummy item.

Although this issue is easily handled[1], it demonstrates the subtlety that arises when we

[1] To give some intuition of security, note that as long as the assignment of the encryption randomness is

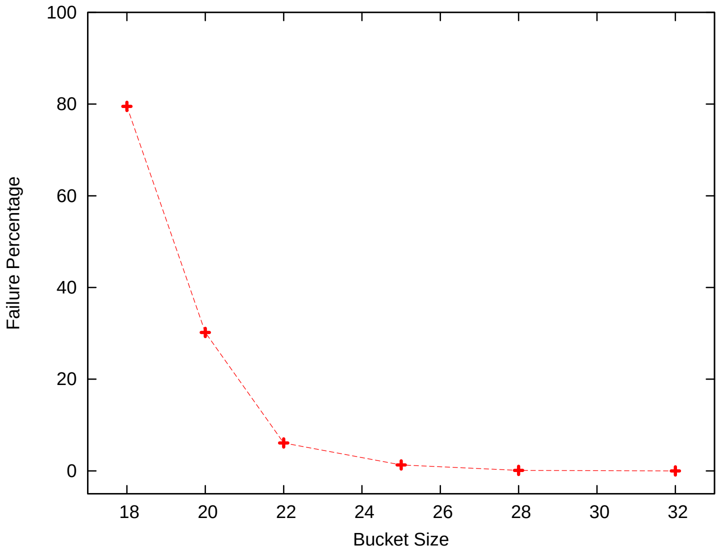


Figure 3.4: Overflow probability as a function of bucket size, for 65536 virtual instructions on a database of 65536 items.

depart from the generic protocol in order to improve the efficiency of the scheme.

Choosing a Bucket Size At each node of the ORAM structure we have a bucket of items, and choosing the size of each bucket can have a big impact on the efficiency of the scheme: we have to perform searches over $B \log N$ items for buckets of size B. However, if the buckets are too small, there is a high probability that some element will "overflow" its bucket during the eviction process. This overflow event can leak information about the access pattern, so it is important to choose large enough buckets. Shi et al. [SCSL11] prove that in an ORAM containing N elements, if the buckets are of size $O(\log(MN/\delta))$, then after M memory accesses, the probability of overflow is less than δ. It follows that to get, say, security 2^{-20}, it suffices to have buckets of size $O(\log N)$, but the constant in the notation is important.

independent of the access pattern, nothing can be learned by the client during decryption. To make this formal, we show that we can simulate his view by choosing random values for each bucket, storing them between look-ups, and sending those same values the next time that bucket is scanned. This simulation would fail only if the assignment of the random values to buckets were somehow dependent on the particular content of the RAM.

In Figure 3.4 we provide our results from simulating overflow for various bucket sizes. Notice that the value approaches 0 only as we approach $2\log N$, and in fact the probability of failure is very high for values closer to $\log N$. Based on these simulations, we have chosen to use buckets of size $2\log N$. We ran our experiment with $N = 2^{16}$ and estimated the probability of overflowing any bucket when we insert all N items, and then perform an additional N operations on the resulting database. We used 10,000 trials in the experiment. Note that for the specific example of binary search, we only need to perform $\log N$ operations on the database; for 2^{16} elements and a bucket size of 32, we determined with confidence of 98% that the probability of overflow is less than .0001. The runtime of our protocol (roughly) doubles when our bucket size doubles, so although we might prefer still stronger security guarantees, increasing the bucket size to $3\log N$ will have a considerable impact on performance. [2]

Computing Addresses Recursively Recall that the leaf node assigned to item $v^{(i)}$ in the ith tree is stored in item $v^{(i+1)} = \lfloor \frac{v^{(i)}}{r} \rfloor$ of the $i + 1$th tree. In Step 2, where the two parties compute shares of $v^{(i)}$ for each i in $1, \ldots, \log_r N$, we observe that if r is a power of 2, each party can compute its own shares locally from its share of v. If $r = 2^j$ and $v = v_C \oplus v_S$, then we can obtain $v^{(i)} = \lfloor \frac{v}{r^i} \rfloor$ by deleting the last $i \cdot j$ bits of v. Similarly $v_C^{(i)}$ and $v_S^{(i)}$ can be obtained by deleting the last $i \cdot j$ bits from the values v_C and v_S. This allows us to avoid performing another secure computation when recovering shares of the recursive addresses.

Node Storage Instantiation Shi et al. [SCSL11] point out that the data stored in each node of the tree could itself be stored in another ORAM scheme, either using the same tree-based scheme described above, or using any of the other existing schemes. We have chosen to simply store the items in an array, performing a linear scan on the entire node. For the data sets we consider, $N = 10^6$ or 10^7, and $20 \leq \log N \leq 25$. Replacing a linear scan with an ORAM scheme costing $O(\log^3 N)$, or even $O(\log^2 N)$, simply does not pay off. We could

[2] The work of this Chapter we developed before newer versions of the tree-based ORAM were proposed (Path-ORAM [SvDS+13]). This newer technique allows us to reduce the bucket size to a small constant. In [GGH+13] authors shows a new eviction procedure that allow to reduce the buckets capacity to as low as 2 elements.

consider the simpler ORAM of Goldreich and Ostrovsky [GO96] that has overhead $O(\sqrt{N})$, but the cost of shuffling around data items and computing pseudorandom functions inside garbled circuits would certainly erase any savings.

Using Client Storage When the client and server search for an item v, after they recover the leaf node assigned to v, the server fetches the $\log N$ buckets along the path to the leaf, each bucket containing up to $\log N$ items. The parties then perform a secure computation over the items, looking for a match on v. We have a choice in how to carry out this secure computation: we could compare one item at a time with v, or search as many as $\log^2 N$ items in one secure computation. The advantage to searching fewer items is that the garbled circuit will be smaller, requiring less client-side storage for the computation. The disadvantage is that each computation will have to output secret shares of the state of the search, indicating whether v has already been found, and, if so, what the value of its payload is; each computation will also have to take the shares of this state as input, and reconstruct the state before continuing with the search. The extra state information will require additional wires and gates in the garbled circuits, as well as additional encryptions and decryptions for preparing and evaluating the circuit. We have chosen to perform just a single secure computation over $\log^2 N$ items, using the maximal client storage, and the minimal computation. However, we note that the additional computation would have little impact,[3] and we could instead reduce the client storage at relatively little cost. To compute the circuit that searches $\log^2 N$ items, the client needs to store approximately $400,000$ encryption keys, each 80 bits long.

Garbled Circuit Optimizations The most computationally expensive part of Yao's garbled circuit protocol is often thought to be the oblivious transfer (OT) sub-protocol [EGL85]. The parties must employ OT once for every input wire of the party that evaluates the circuit, and each such application (naively performed) requires expensive operations such as exponentiations. We use the following known optimizations to reduce OT costs and to further push its computation almost entirely to the preprocessing stage, before the

[3] This is because sharing the state and reconstructing the state are both done using XOR gates, which are particularly cheap for garbled circuits, as we discuss below.

parties begin the computation (even before they have their inputs), reducing the on-line OT computations to just simple XOR operations.

The most important technique we use is the OT extension protocol of Ishai et al. [IKNP03], which allows to compute an arbitrary number of OT instances, given a small (security parameter) number of "base" OT instances. We implement the base instances using the protocol of Naor and Pinkas [NP01], which requires six exponentiations in a prime order group, three of which can be computed during pre-processing. Following [IKNP03], the remaining OT instances will only cost us a couple of hash evaluations per instance. We then push these hash function evaluations to the preprocessing stage, in a way that requires only XOR during the on-line stage. Finally, Beaver's technique [Bea95] allows us to start computing the OT's in the preprocessing stage as well, by running OT random inputs for both parties; the output is then corrected by appropriately sending real input XORed with the used random inputs in the online stage.

We rely on several other known garbled circuit optimizations. First, we use the *free XOR gates* technique of Kolesnikov and Schneider [KS08a], which results in more than 60% improvement in the evaluation time for an XOR gate, compared to other gates. Accordingly, we aim to construct our circuits using as few non-XOR gates as possible.

Second, we utilize a wider variety of gates (as opposed to the traditional Boolean AND, OR, XOR, NAND gates). This pays off since in the garbled circuit construction every non-XOR gate requires performing encryption and decryption, and all gates of the same size are equally costly in this regard. In our implementation we construct and use 10 of the 16 possible gates that have 2 input bits and one output bit. We also rely heavily on the multiplexer gate on 3 input bits; this gate uses the first input bit to select the output from the other two input bits. In one circuit, we use a 16-bit multiplexer, which uses 4 input bits to select from 16 other inputs.

Finally, we utilize *pipelined circuit execution*, which avoids the naive traditional approach where one party sends the garbled circuit in its entirety to the second one. This naive approach is often impractical, as for large inputs the garbled circuits can be several gigabytes in size, and the receiving party cannot start the evaluation until the entire garbled circuit has been generated and transmitted and stored in his memory. To mitigate that, we follow

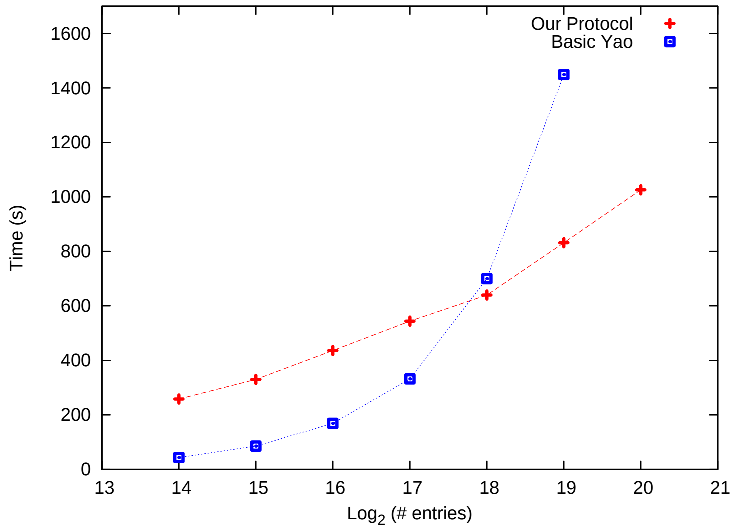


Figure 3.5: Time for performing binary search using our protocol vs. time for performing binary search using a standard garbled-circuit protocol as a function of the number of database entries. Each entry is 512 bits long.

the technique introduced by Huang et al. [HEKM11], allowing the generation and evaluation of the garbled circuit to be executed in parallel, where the sender can transmit each garbled gate as soon as he generates it, and continue to garble the next gates while the receiver is evaluating the received gates, thus improving the total evaluation time. This also alleviates the memory requirements for both parties since the garbler can discards the gates he has sent, and the receiver can discard a gate that he has evaluated.

3.7 Implementation

The goal of our experiments was to evaluate and compare execution times for two protocols implementing binary search: one using standard optimized Yao, and the other using our ORAM-based approach described in the previous section. In our experiments, each of the two parties was run on a different server, each with a Intel Xeon 2.5GHz CPU, 16 GB of RAM, two 500 GB hard disk drives, and running a 64-bit Ubuntu operating system. They each had a 1 Gbit ethernet interface, and were connected through a 1Gbit switch.

Before running our experiments, we first populated the database structure on the server side: in our ORAM protocol, we randomly placed the encrypted data throughout the ORAM structure, and in the Yao protocol performing a linear scan, we simply stored the data in a large array. We then generated and stored the necessary circuit descriptions on each machine. Finally, the two parties interacted to pre-process the expensive part of the OT operations, in a manner that is independent of their inputs. We did not create the garbled gates for the circuits during pre-processing; the server begins generating these once contacted by the client. However, the server sent garbled gates to the client as they were ready, so as to minimize the impact on the total computation time. When we measured time in our experiments, we included: 1) the online phase of the OT protocol, 2) the time it takes to create the garbled gates and transfer the resulting ciphertexts, and 3) the processing time of the garbled circuits.

3.7.1 Performance

In Figure 3.5, we compare the performance of our construction when computing a ORAM-based binary search to the performance of a Yao-based linear scan. We have plotted the x-axis on a logarithmic scale to improve readability. From the plot it can be seen that we outperform the Yao linear scan by a factor of 3 when dealing with input of size 2^{19}, completing the $\log N$ operations in less than 7 minutes, compared to 24 minutes for Yao. For input of size 2^{20}, we complete our computation in 8.3 minutes, while the Yao implementation failed to complete due to memory constrains. While we had no trouble running our ORAM-based protocol on input of size 2^{20}, for $N = 2^{21}$, we ran out of memory when populating the server's ORAM during pre-processing.

In Figure 3.6 we demonstrate how our protocol performs when evaluating a single read operation over N data elements of size 512 bits, for $N \in \{2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$. We note that runtime for binary search using the ORAM is almost exactly the time it takes to run $\log N$ single look-ups; this is expected, since the circuit for computing the next RAM instruction is very small. For 2^{16} items and a bucket size of 32, a single operation takes 27 seconds, while for 2^{20} items and buckets of size 40, it takes about 50 seconds. Recall that when relying only on secure computation, computing *any* function, even those making a constant number of look-ups, requires a full linear scan; in this scenario, the performance

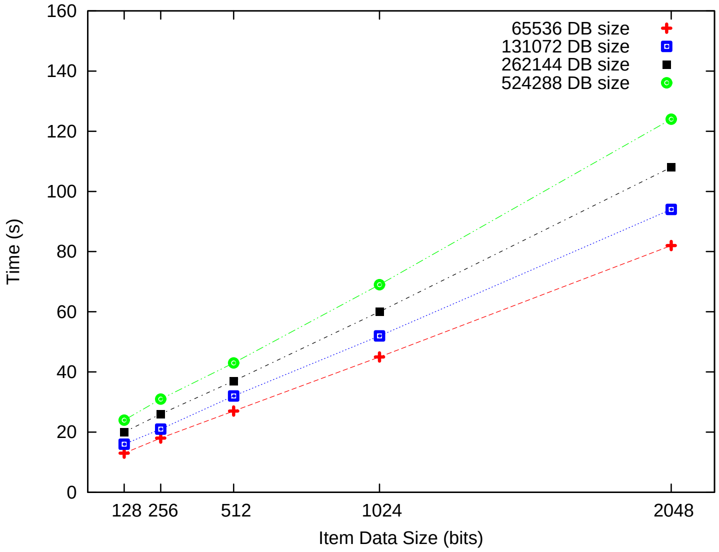


Figure 3.6: Single ORAM look-up times for different database sizes and item data lengths.

gain is more than 30-fold. One example of such a function allows the client to search a large social network, in order to fetch the neighborhood of a particular node.

3.7.2 Discussion

Memory Constraints Memory is the primary limitation on scaling the computation to larger values of N. For the linear scan, the problem stems from the size of the circuit description, which is more than 23 gigabytes and 850 million wires, if $N = 2^{19}$ and the data elements are 512 bits. The pipe-lining technique of Huang et al. [HEKM11] prevents the parties from storing all 23 gigabytes in RAM, but the client still stores an 80 bit secret key for every wire in the circuit, and the server stores two; this requires 8.5 gigabytes of memory for the client and 17 gigabytes for server. This ends up requiring far more space than the data itself, which is only $512N = 33$ megabytes.

In contrast, when $N = 2^{19}$ and the data size is 512, the largest circuit in our protocol is less than 50 megabytes, and contains about 1million wires. On the other hand, each level of the data storage has a factor of $4 \log N$ overhead (when our bucket size is $2 \log N$), so server storage for the top level alone is more than $40000N = 2.5$ gigabytes. This explains why we eventually ran into trouble when pre-processing the data; to broaden the scale of what we can handle, we will need to improve the way we handle memory while inserting elements into the ORAM structure.

Pre-processing We have not done any calculations regarding the time required for secure pre-processing. As explained above, when running our experiments, we populated the ORAM structure by randomly placing items in the trees. This is of course insecure, since the server will know where all the items are in the ORAM: to ensure security, the insertion of the data would have to be interactive. One naive way to ensure security is to insert each item, one at a time, by performing the "write" protocol inside a secure computation, precisely as we have described an ORAM look-up. If we start this process with a data structure large enough to hold all items, we can estimate the time it will take to insert 2^{16} elements of 512 bits each, by multiplying the 13 seconds we require for a write operation by 2^{16}. It seems this would take almost 20 days to compute! We leave the problem of finding a more efficient method for data insertion to future work. One natural approach would be to start with smaller structures, repeatedly doubling their size in some secure manner as insertion progresses. We stress that the pre-processing we do in our work is *fully secure* in a three-party model, where the database owner pre-processes his data, and then transfers the encrypted data to a semi-honest third party, who performs the secure computation on his behalf.

The Recursion Parameter In all of our experiments, we have chosen $r = 16$; that is, every item in tree $i > 1$ stores the leaf nodes of 16 items from tree $i-1$. This is a parameter that we could change, and it may have an impact on performance. However, one parameter we did investigate is the choice of how far to recurse. As can be seen in Table 1, the best performance occurs when the bottom level, which requires a linear scan, holds fewer than 2^{12} items. Interestingly, beyond that, further recursion does not seem to make a difference. The ith tree

Counting Gates Let N be the number of elements, let d be the length of each element, and let B denote the bucket size of each node. We calculate the number of non-XOR gates in the garbled circuits of our ORAM operation, and provide some relevant observations. We

DB size	**2 trees**	**3 trees**	**4 trees**	**5 trees**
2^{20}	35	14	**12.5**	13
2^{19}	20	**11.5**	12.5	-
2^{18}	12.5	**9.5**	**9.5**	-

Table 3.1: Time in seconds of a single ORAM access, with various numbers of recursion levels in the ORAM structure. The number of items in the bottom level is 2^{N-4i+4} when there are i trees.

first consider the top level tree that contains the database items. During a lookup we need to check $\log N$ nodes along the path to the leaf associated with the searched item. Each of these nodes contains B elements of size $\log N + d$: a virtual address of size $\log N$ and a data element of size d. We use approximately 1 non-XOR gates for each of these. Therefore, a single lookup consists of $B \log N(\log N + d)$ non-free gates. In the eviction process that follows, we scan $2 \log N$ nodes for eviction, and write to both of children of each node (one write is dummy). Thus, the eviction circuits require $6B \log N(\log N + d)$ non-free gates, which gives us a total of $7B \log N(\log N + d)$ non-free gates for each ORAM operation in the top level tree. The analysis at the lower level trees is similar, but asymptotically, this dominates the computation, since the lower level trees have only $N/16^i$ elements. We provide concrete numbers in Table 2, taken directly from our circuits. We considere $B = 2 \log N$ and d=512. We note that our circuits grow linearly in the size of each bucket. Also interesting is that it grows linearly in d. Since the Yao linear scan is also linear in the data size, with dN gates, we see that varying the length of the data element will have little impact on our comparison.

3.8 Using Other ORAM Schemes

In our concrete protocol we instantiated (and then optimized) our generic construction using the tree-based ORAM scheme of [SCSL11][4]. However, there are several other oblivious RAM schemes which we considered as possible instantiations for our ORAM component.

[4] As mentioned earlier, we could have also use the Path-ORAM scheme [SvDS$^+$13]

DB size	XOR gates	Non-free gates	Wires
2^{20}	19,159,883	3,730,546	44,039,222
2^{19}	16,519,818	3,166,420	37,656,448
2^{18}	14,219,281	2,700,966	30,941,947
2^{17}	12,185,264	2,302,208	27,366,108
2^{16}	10,377,527	1,954,042	23,655,368

Table 3.2: Gate and wires counts for different size databases with item data of length 512

We discovered that these schemes would entail higher complexity in the context of a two party computation protocol[5] since they involve more complicated building blocks such as pseudorandom shuffling protocol and Cuckoo hashing.

For example, the ORAM protocol of Goldreich and Ostrovsky [GO96] introduced the basic hierarchical structure that underlies many subsequent ORAM protocols. This approach crucially relies on two components that turn out to be quite inefficient when evaluated with a secure two-party computation: (1) the use of a pseudorandom function (PRF) in order to consistently generate a random mapping from virtual addresses to physical addresses; and (2) a joint shuffling procedure for mixing the different levels in the ORAM data structure. We direct the reader to [GO96] for the full details of the scheme.

Several more-recent ORAM solutions [PR10b; GM11; GMOT12; KLO12] rely on cuckoo hashing (in addition to also using PRF computations). For their security, a new construction for a cuckoo hash table is needed [GM11], which involves building the corresponding cuckoo graph and conducting breadth-first search on the graph in order to allocate each new item inserted into the cuckoo table. Compiling this step into a secure two-party computation protocol seems likely to introduce a prohibitive performance hit.

[5] Note that a better performing ORAM protocol does not necessarily translate to a better performing protocol when put through a generic secure computation.

3.9 Conclusion

In this work we showed efficient protocols for secure two party computation achieving only a small polylogarithmic overhead over the running time of the insecure version of the same functionality. This is a significant asymptotic improvement over traditional generic secure computation techniques, which inherently impose computation overhead at least linear in the input size. Our protocols rely on any (arbitrary) underlying oblivious RAM and generic two party computation protocols. We further investigate the most efficient instantiation of the protocol and demonstrated, empirically, the expected theoretical improvement. In particular, we implemented a protocol that performs a single access to a databases of size 2^{18} elements, outperforming an implementation of basic secure computation by a factor of 60. This translates also to a three-fold improvement in the running time of binary search. In addition to these concrete improvements, our work sheds light on many of the details faced when implementing ORAM and secure computation.

Part II

Private Database Search

Chapter 4

Blind Seer: A Scalable Private DBMS

4.1 Introduction

As noted in the very introduction of this thesis, the amount of data generated and stored is now reaching dramatic proportions, and it is touching every aspect of our life (social, political, commercial, scientific, medical, and legal contexts). Personal, corporate and government concerns about privacy increase with the rise in size and potential applications utilizing these data. For example, the recent revelation of the U.S. Government's data collection programs reignited the privacy debate.

In this and the following chapters 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 administrator. 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 that is not publicly accounted for.
- 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 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 [Kay12; PI05]. Often in these scenarios the (private) query should be answered only if it satisfies a certain (secret) policy.

While achieving full privacy for these scenarios is possible by building on cryptographic tools such as SPIR [GIKM00], FHE [Gen09], ORAM [GO96] or multiparty computation (MPC), those solutions either run in polynomial time, or have very expensive basic steps in the sublinear algorithms. Recall the ORAM based system of Chapter 3. Figure 3.5 show that it takes about 1000 seconds to run a binary search on 2^{20} entries; subsequent works [LO13a; GGH$^+$13] 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, rich in functionality, and with 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 that 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 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 [CJJ+13; JJK+13], it was *not* known how to efficiently avoid this leakage. (See Section 4.1.2 and Chapter 7 for extended discussion on related work.)

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; having C communicating mainly with the third party IS, allows keeping S oblivious of the client queries, and hence we can aim 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. On the other hand, to perform practical-efficient equality check of encrypted data, we would likely require the use of deterministic encryption, which allows to identify and accumulate access and search patterns. In addition, 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 4.5.2 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 4.5.

In Section 4.8 we provide further motivation, examples and discussion of our setting and choices.

4.1.1 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 (similar 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 4.4.2).

- 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 queries (e.g., just keyword search), or provide weaker privacy guarantees. The independent and concurrent work of [CJJ$^+$13; JJK$^+$13] is the only system comparable to Blind Seer, 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.

4.1.2 Background

The closest to our setting/work is a OSPIR-OXT [JJK$^+$13], a very recent extension of the SSE solution of [CJJ$^+$13](called OXT protocol), 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 [CJJ$^+$13; JJK$^+$13] is performed independently and concurrently to the development of this work. OSPIR-OXT supports the same class of functions as OXT, that is, formulas of the type $k_0 \wedge \phi(k_1, ..., k_{m-1})$. Their search time complexity is $O(m \times D(k_0))$, where $D(k_0)$ is the number of records containing keyword k_0. 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 OSPIR-OXT can be linear in the DB size. In contrast, the solution for general formulas proposed here does not depend on the client's knowledge of data distribution or representation choice (beyond the size of the formula). However, 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 OSPIR-OXT with asymptotic complexity similar to Blind Seer. In terms of security, our guarantees vary: OSPIR-OXT achieves security against malicious client, which is much stronger than our semi-honest setting, and of particular importance for the policy enforcement (this is handled in Chapter 5). Blind Seer and OSPIR-OXT leakage profile vary and are incomparable; different access pattern structures (search tree for Blind Seer and index lookups for OSPIR-OXT). Because of the use of a more expensive basic step of SFE, the protection of query-related data, at least in some cases, is somewhat better in Blind Seer. For example, depending on the DB data, we may hide everything about the individual terms of the query, while OSPIR-OXT 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 OSPIR-OXT 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 4.4.2), 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 [JJK$^+$13]. The use of GC as the basic block similarly provides significant flexibility and opportunities for feature expansion. A strong point of OSPIR-OXT 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, OSPIR-OXT 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 [JJK$^+$13], 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.

4.2 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 4.1.

Approach. We present a high-level overview of our approach and refer the reader to Section 4.3 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,

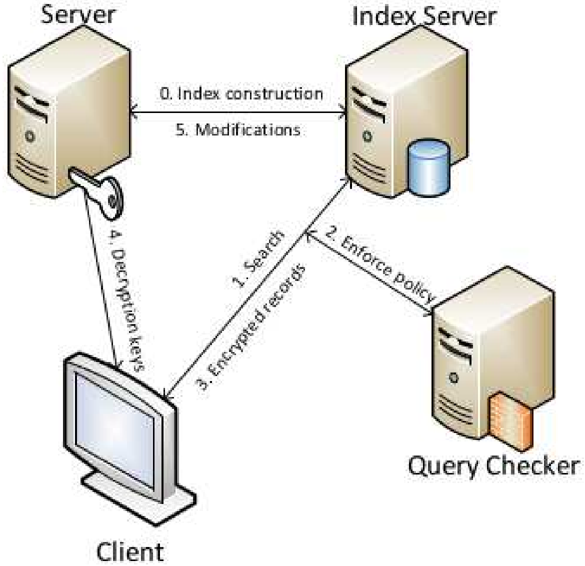


Figure 4.1: High-level overview of Blind Seer. There are three different operations depicted: preprocessing (step 0), database searching (step 1-4) and data modifications (step 5).

unlike the protected input and the internals of this subtask, its output, obtained in plaintext by IS and C, 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 4.3. We stress that many of these details are technically involved.

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 $1110{\ddagger_S} = 0100$.

4.3 Basic System Design

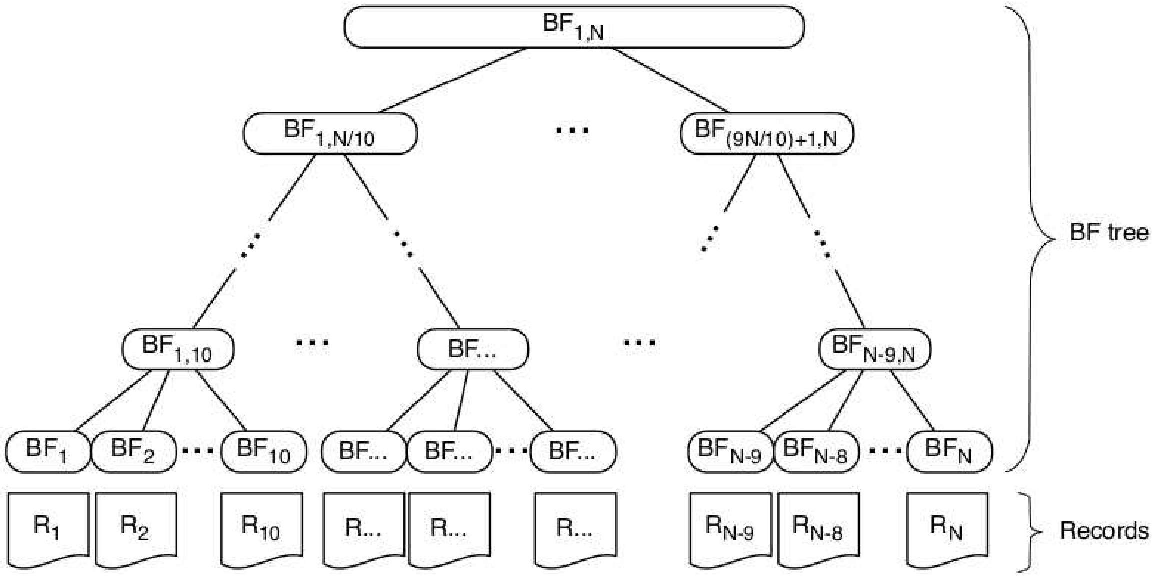
In this section, we will begin by describing the basic system design supporting only simple private query. In the next section, we will augment this basic design with additional features.

4.3.1 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 $\log_b n$ [1] (we assume $n = b^z$ from some positive integer z for simplicity). 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 $\mathcal{H}(\alpha))$ 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 4.2.

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

[1] In our prototype implementation, b is set to 10.



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 4.2: Index structure: Bloom-filter-based search tree.

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.

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

4.3.2 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 pseudo-random function (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: $\tilde{B}_v = B_v \oplus F_k(v)$.

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.

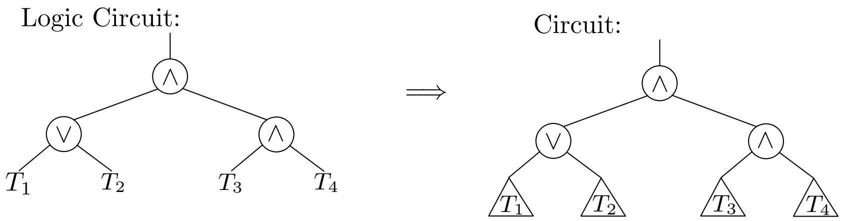
Preparing record decryption keys. To save the decryption time in the on-line phase, the index server and the server precompute record decryption keys as follows:

> (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 $s'_{\psi(i)} = s_i r_i$.

4.3.3 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 4.3 as an example.

Query: `SELECT * FROM main WHERE`
`(fname = JEFF OR fname = JOHN) AND zip = 34301 AND income` $\leq$ `200`



T_1:`fname = JEFF` T_3:`zip = 34301`
T_2:`fname = JOHN` T_4:`income`$\leq$`200`

Figure 4.3: High level circuit representation of a query.

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 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.

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 4.3 the variable T_1 indicates whether the Bloom filter B_v in a given node v contains the keyword $\alpha =$ `‘fname:JEFF’`. Consider the Bloom filter hash values for the keyword α, and let Z denote the positions to be checked. 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

$$B_v{\downarrow_Z} \overset{?}{=} 1^\eta. \tag{4.1}$$

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 (4.1). 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 4.3 for example. Using the technique from [RVBM09], 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 a range of size 2^i that start on an even position. That is, $[x2^i, (x+1)2^i)$ for some integer x. 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 within $\mathbb{Z}_{2^4}$. 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 . The client and the index server execute the following in a node v 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 (corresponding to $\mathrm{PRF}_k(v)\ddagger_Z$).

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.

4.4 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.

4.4.1 Policy Enforcement

The policy enforcement over a query 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 later in this section). 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 $\mathbf{b}$ via 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 next describe in detail our own implemented policy circuit.

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)      (4.2)
```

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 (4.2) 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, ....`

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.

4.4.2 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:

$$\mathcal{D}_\alpha(x) = \begin{cases} 1/\alpha & \text{if } 1 \leq x \leq \alpha - 1 \\ 1/\alpha \cdot 1/2^{x-\alpha+1} & \text{if } x \geq \alpha \end{cases}$$

Here, α is a tunable parameter. The client chooses $x \leftarrow \mathcal{D}$, and then it also chooses x uniformly random indices $(j_1, \ldots, j_x)$ in $[n]$. 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.

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.

One-case indistinguishability proof

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 previously with δ is defined as follows:

> Let $(D_0, q, \mathbf{r})$ be a database, a query and a record. 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:

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

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

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

We show this by using case analysis:

- When $x \leq 1$, it never comes from $\mathcal{D}^+$, so the inequality trivially holds.
- When $2 \leq x \leq \alpha - 1$, it holds that

$$\Pr[b=1|\; x] = \frac{\Pr[X=(1,x)]}{\Pr[x]} = \frac{\delta/\alpha}{\delta/\alpha + (1-\delta)/\alpha} = \delta.$$

- When $x \geq \alpha$, it holds that

$$\begin{aligned}\Pr[b = 1 | \ x] = \frac{\Pr[X = (1, x)]}{\Pr[x]} \\ = \quad & \frac{\delta \cdot (1/\alpha) \cdot 1/2^{x-\alpha}}{\delta \cdot (1/\alpha) \cdot 1/2^{x-\alpha} + (1-\delta) \cdot (1/\alpha) \cdot 1/2^{x-\alpha+1}} \\ = \quad & \frac{\delta}{\delta + (1-\delta)/2} = \frac{2\delta}{1+\delta} \leq 2\delta.\end{aligned}$$

4.4.3 Delete, Insert, and Update from the Server

Blind Seer 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. Finally, update is simply handled by atomically issuing a delete and an insert command.

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.

4.5 Security Analysis

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 4.4, 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.

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 4.3); 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 4.4: The Ideal Functionality $\mathcal{F}_{db}$

[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.

4.5.1 Security of Our System

With empty leakage profile, the ideal functionality $\mathcal{F}_{db}$ in Figure 4.4 captures the privacy requirement of a database management system in which each query is handled deterministically. The client obtains only the query results, and nothing more. The index server and the server learn nothing. We show now that Blind Seer realizes the functionality $\mathcal{F}_{db}$ with aproppiate leakage profile.

4.5.1.1 Simulating Client's Adversarial Behaviour

Leakage to C*in Init.* The leakage to C is n, that is, the total number of records.
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. We denote this leakage as $\mathsf{SearchPattern}$. The protocol also leaks the permuted id's of the retrieved records $\pi(id(r_1)), ..., \pi(id(r_\ell))$.
Simulating Preprocessing. The simulator $\mathcal{S}$ stores n, runs the adversary $\mathcal{A}$ that will follow the protocol description for the client. The client is suppose to receive $n, (k, \mathcal{H})\}$. $\mathcal{S}$ can randomly choose $k, \mathcal{H}$.
Simulating Query. The simulator $\mathcal{S}$ works as follows:

1. Send query q to the ideal functionality, and receive back the records $(r_1, ..., r_\ell)$ as well as the leakage $(\pi(id(r_1), ..., \pi(id(r_\ell)))$, $\mathsf{SearchPattern}$, and cnt. The simulator $\mathcal{S}$ runs the adversary $\mathcal{A}(q)$. $\mathcal{S}$ now simulates the tree traversal with adversary $\mathcal{A}$ starting at the root as follows:

    - If node v is not a leaf of the search tree, $\mathcal{S}$ checks whether v is in $\mathsf{SearchPattern}$. If so, $\mathcal{S}$ simulates Yao's protocol such that the output to the client is 1. Otherwise, simulate Yao's protocol such that the output to the client is 0.

    - If node v is a leaf of the tree then $\mathcal{S}$ checks if v belongs to $\mathsf{SearchPattern}$ (v is $\pi(id(r_j))$ for some j), $\mathcal{S}$ simulates Yao's protocol so that output is 1. Simulator now encrypts r_j, and sends it to $\mathcal{A}$ along with additional information following the the protocol specification.

4.5.1.2 Simulating Server's Adversarial Behaviour

Leakage to S *in Init.* None.

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

Simulating Preprocessing. Since IS has no input, the simulator can just follow IScode to produce $\tilde{s}'_1, .., \tilde{s}'_n$, where each $\tilde{s}'_i$ is computed as $\mathsf{Enc}_p k(r_i)$ for some random r_i. Simulator sends this values back to the adversary $\mathcal{A}$.

Simulating Search. Given the permuted record indices $\pi(id(1)), ..., \pi(id(n))$, the simulator acting as the client sends this indices to the adversary.

4.5.1.3 Simulating Index Server's Adversarial Behaviour

Leakage to IS *in Init.* The number of records in the database n and the size of the records $|D_1| = |D_2| = \cdots = |D_n|$.

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, let's denote this leakage as $\mathsf{iSearchPattern}$. Also, the topology of the query circuit $\texttt{topo}(q)$ and of the policy circuit $\texttt{topo}(p)$ is leaked to IS as well. Finally, the BF indices $(\mathcal{H}(q))$ 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.

Simulating Preprocessing. After receiving the number of documents n and the length of the documents len as leakage, $\mathcal{S}$ runs the adversary $\mathcal{A}$. Simulating the server, $\mathcal{S}$ generates a key pair $(pk, sk) \leftarrow \mathsf{Gen}(\lambda)$. For $i \in [n]$ $\mathcal{S}$ samples s_i, then it encrypts s_i under pk $(\tilde{s}_i)$, and finaly encrypts a dummy record as $\tilde{R}_i \leftarrow \mathsf{Enc}_{s_i}(0^{\mathsf{len}})$. $\mathcal{S}$ gives $(pk, \{\tilde{s}_i \tilde{R}_i, \})$ to $\mathcal{A}$. Then it receives $\{\tilde{s}'_i\}$ from the adversary. To build the masked Bloom filter tree, he simulator samples a key k for a pseudorandom function F and for each node v for the tree construct the "Bloom" filter as $\tilde{B}_v = G(F_k(v))$, where G is a pseudorandom generator. $\mathcal{S}$ sends the Bloom filter tree to $\mathcal{A}$.

Simulating Search. Run the adversart $\mathcal{A}$ providing it with $\mathcal{H}(q)$. $\mathcal{A}$ and $\mathcal{S}$ traverse the search tree starting at the root. For each node v the $\mathcal{S}$ simulates Yao's protocol as if it was the client. If node v belongs to iSearchPattern, then $\mathcal{S}$ tell $\mathcal{A}$ that evaluation was successful, and they visit the node's children (if node was a leaf, $\mathcal{S}$ ask $\mathcal{A}$ to provide the encrypted record and decryption information). Otherwise, $\mathcal{S}$ tell $\mathcal{A}$ that evaluation was unsuccessful.

4.5.2 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 [KS08b] 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 4.7.3 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.

4.6 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 the cryptographic building blocks described in the preceding sections. 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 implementation of Blind Seer supports parallel preprocessing and per-query threading when searching. For all the multi-threading features we used Intel's

[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.

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.

4.6.1 Additional Search functionality

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 [ste].

In addition to our performance and security improvements, we have extended the functionality of Blind Seer to support additional types of queries.

M-of-N queries. An M-of-N threshold query contains N clauses, and returns true if and only if at least M of the clauses return true. To support threshold queries, we construct query circuits for each individual clause, and use a Boolean counting circuit to count the number of positive results. The counting circuit is implemented by chaining full adders to compute each digit of a binary representation of the sum. Finally, the result is compared to M with a simple comparison circuit.

Ranking M-of-N queries. In addition to supporting M-of-N queries, we can return results to the querier ordered by the number of clauses they satisfy. The circuit construction for this is the same as for normal M-of-N queries. However, instead of receiving only the final bit of the comparison circuit from the garbled circuit evaluation, the querier receives all of the bits from the counting circuit output.

Ultimately, we can sort all record results by rank on the querier's side. However, we also would like to reach higher ranking results more quickly. For instance, in a query with many low ranking results, it would be advantageous for the querier to see the high ranking results quickly.

To accomplish this, we use a priority queue seeded both by the depth of the node and the value from the counting circuit during tree traversal. We thus do a depth-first search favoring those branches with large amounts of matching clauses early on.

This a heuristic, it is not guaranteed to reach the best results first; even if a parent node has a high number of matching clauses, it is not necessarily true that its individual children are a high match. However, in the common case, the method can allow us to return high ranking results quickly.

Nearness queries. Another type of query we support is a proximity query, which takes two terms in a text segment along with a range r, and returns all records for which the two terms coincide within a window of r words.

This is not straightforward to do with a Bloom Filter representation of terms that does not store term locations. As such, we are only able to do this efficiently for values of r that are known during preprocessing. We can then add into the Bloom Filter all term pairs that fit within that range. The nearness query becomes a simple check against the Bloom Filter for the appropriate term.

4.7 Evaluation

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

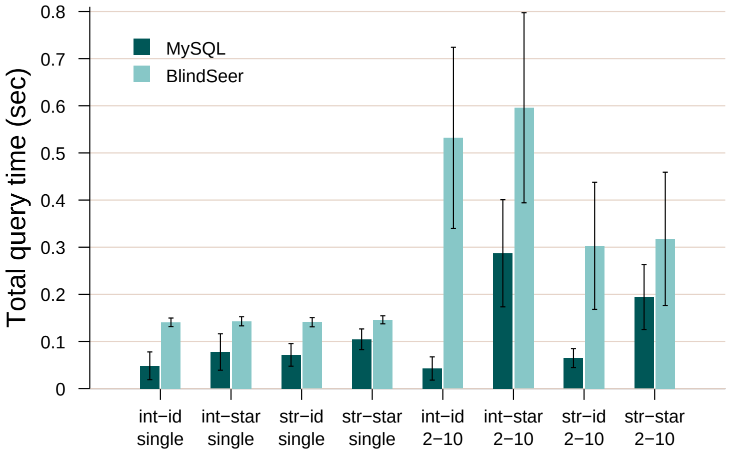


Figure 4.5: Comparison with MySQL for single-term queries that have a single result (first four bar groups) and 2 to 10 results (last four bar groups). The search terms are either strings (str) or integers (int) and the returned result is either the id or the whole record (star).

Testbed. The 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.

4.7.1 Querying Performance

Single term queries with a small result set. Figure 4.5 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

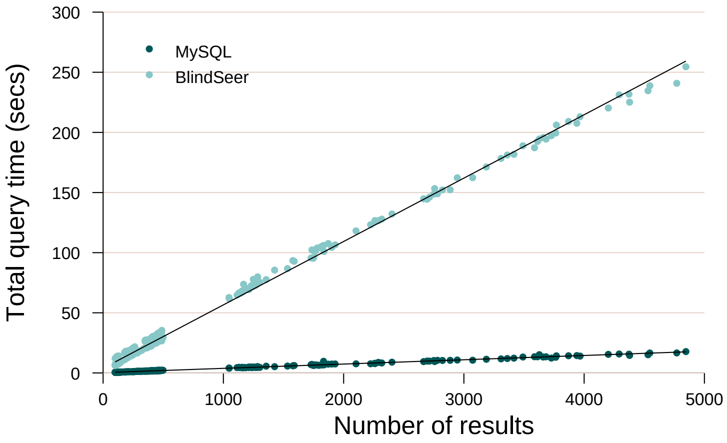


Figure 4.6: Comparison of the scaling factor with respect to the result set size, using single-term queries. Both MySQL and Blind Seer scale linearly, however, Blind Seer's constant factor is 15× worse (mostly due to increased network communication).

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× 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.

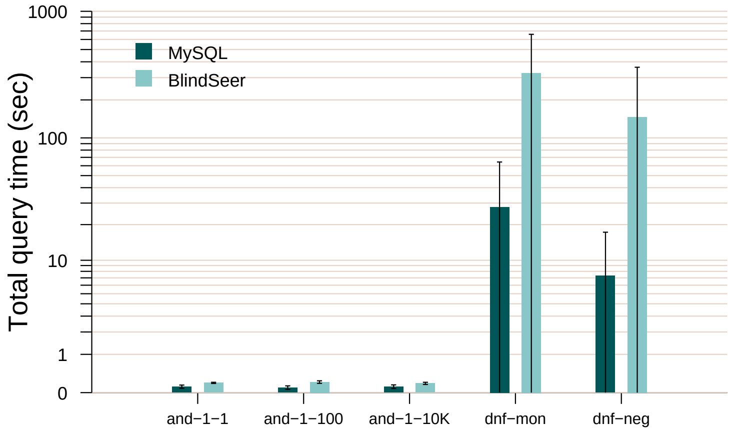


Figure 4.7: Boolean queries having a few results (< 10). The first three are two-term AND queries where one of the terms has a single result and the other varies from 1 to 10K results. The fourth group includes monotonic DNF queries with 4-9 terms, the last includes 5-term DNF queries with negations.

Scaling with result set size. Figure 4.6 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× 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 protocol. Although this is inherent, we believe that there is room for implementation optimizations that could lower this constant factor.

Boolean queries. Figure 4.7 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 [Kol05] or its recent extension sliced-GESS [KK12], 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.

4.7.2 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.

Also, 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 4.4.3). 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.

4.7.3 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 for conjunctions, 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.

4.8 Discussion

Semi-honest model. Semi-honest model is often reasonable in practice, especially in the Government use scenarios. For example, C, S and ISmay 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 Chapter 5 we show how to amend our protocols to protect against a malicious clients at a very small cost. 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 phone number = xxx AND callee phone number = yyy AND time of call` $\in$ `{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 argue that Blind Seer 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. We study this approach in Chapter 6.

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.

4.9 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× to 3× slowdown.

We introduced a policy checking mechanism over the queries as an extra feature of our system. In real-life scenarios such a property is of great importance, and often mandatory (otherwise a client can download the entire database by sending a sinfle `SELECT *` query, for example). However, its security relies on semi-honest client behaviour. A malicious client can easily circumvent the mechanism by providing different queries for the tree traversal and for the policy procedures. We believe that this drawback is of great importance. In the next chapter we show how to modify the Blind Seer system to enforce security against a malicious client at virtually no cost.

Chapter 5

Malicious Client Security on Blind Seer

5.1 Introduction

In the preceding chapter we introduced Blind Seer; a highly practical, sublinear, and privacy protected DB querying, with provable security with respect to a controlled amount of information leakage (e.g., search patterns across multiple queries). OSPIR-OXT [JJK$^+$13](see Section 4.1.2) is another highly efficient private DBMS that was concurrently developted with Blind Seer. No other system has been proposed that achieve the level of efficiency, functionality and privacy of Blind Seer and OSPIR-OXT. These two systems offer varying features and relative advantages, making each system better suited for different application scenarios. In terms of privacy, the Blind Seer system offers stronger guarantees in that it is formally ensured that the individual terms of the query formula are privacy-protected. (In contrast, OSPIR-OXT leaks support sizes of the disjunctive formula terms.)

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

The goal of this Chapter is to lifting the Blind Seer protocol into the malicious setting. Achieving security against malicious players can be achived by using standard techniques; however, these are very expensive. For example, the cut-and-choose approach to malicious MPC (cf. [MF06; LP07; Lin13]) carries the cost of at least 128-fold performance degradation for 2^{-128} security. These costs can be made somewhat better using very recent amortized garbled circuit techniques [HKK+14; LR14]. Still, state-of-the-art generic or specialized techniques result in order(s) of magnitude cost overhead.

The main result presented in this Chapter is, surprisingly, that security against a malicous client in Blind Seer can be obtained *for free*. That is, we show how to protect against a *malicious* client at virtually *no additional performance cost*, as well as no privacy or functionality degradation.

Our result applies not only to Blind Seer, but also to any setting where a potentially malicious party needs to evaluate a private function on a semi-honest party's private inputs. When both parties are semi-honest (or at least the party holding the private function is semi-honest), the *Yao Garbled Circuits* (Yao GC) protocol is a practical method that entirely preserves the privacy of the inputs and reveals no more than the private function's circuit topology. While it is well-known how to achieve this functionality (or even stronger privacy) for malicious players using general and expensive techniques, our technique is as efficient and achieves the same level of privacy as Yao GC.

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

5.1.1 Contributions

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

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

– **Formal proofs:** We formally prove security against arbitrary malicious behavior of the client. We note that full cryptographic proofs are unusual for large systems such as ours. Our other privacy features and leakage profiles remain nearly the same as those of the original Chapter 4 Blind Seer.

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

5.2 Overview

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

Preliminaries We use Bloom filters (BF), semantically secure encryption (both public key and symmetric key), Yao Garbled Circuits (GC), and Oblivious Transfer (OT). All these standard cryptographic primitives are described in Chapter 2.

We use Yao GC to achieve *secure computation*, also called *secure function evaluation*

(SFE), which intuitively means that the two-party function is computed so that each of the parties learns no information except what follows from their own inputs and outputs. One may also consider *Private Function SFE (PF-SFE)*, where in addition the function that is being computed is itself the input of one of the parties, and remains hidden from the other party. A generalization of this is *Semi-Private Function SFE (SPF-SFE)*, where the function is known to belong to some class of functions, but beyond that remains hidden. Here, we will consider SPF-SFE where the function is known to have a certain *topology*. Explicitly, the structure of the gates in the circuit describing the function is known, but the operation of each gate (e.g., OR or AND) remains hidden. (See more details in Section 5.3). Yao GC is an example of a protocol that achieves this property. Yao GC involves one party sending a "garbled circuit" to the other, and while the technique was not designed to hide the function, it turns out to hide the values of the gates.

In our setting, *OT preprocessing* [Bea95] and *OT extension* [IKNP03] dramatically improve performance. We use the Naor-Pinkas protocol [NP01] for the "base" OTs that seed OT extension. We then use a version of the IKNP protocol for OT extension suggested by Nielsen [Nie07] that is robust against a malicious receiver with small additional cost.

Blind Seer's Design and Vulnerabilities

We review here the basic features of the Blind Seer DBMS. We refer the reader to Chapter 4 for details, as well as discussion and motivation for the setting and design choices (some of which are also discussed in [CJJ+13; JJK+13]). However, we stress that further details of the Blind Seer design beyond what is described here and in Section 5.4 are not necessary for understanding the malicious-client vulnerability of the original design and the contributions of the present chapter.

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

A fourth logical entity, called the query checker QC, is responsible for enforcing policy restrictions on the client's queries. For instance, a policy restriction might prohibit queries that ask for records associated with an important political figure. QC may be run by the server S or index server IS, but we will view it as a separate logical entity for sake of generality. This way, we also demonstrate that we can keep the policy hidden from S and IS in addition to the client C. As long as the separate parties do not collude, C only learns the results of queries that pass the policy, C's query is kept private from all other parties, the policy is kept private from C, and the results of the policy check are unknown to any party.

Architecture. The basic architecture of Blind Seer is depicted in Figure 4.1, Chapter 4. The server S, who holds the DB, will hand an encrypted copy of the DB to the third party server IS. In addition, S builds an encrypted Bloom filter (BF) tree index to the DB and sends it to IS.

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

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

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

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

Privacy and Leakage. Ideally, the client C would learn nothing except the result sets of its authorized queries, and the servers S and IS (as well as the query checker QC) would learn nothing at all. Blind Seer achieves privacy up to a controlled amount of leakage of *search patterns*. IS may correlate repeated queries, and both C and IS may observe traversal patterns through the tree index across multiple queries, possibly obtaining some correlation information between different queries. Additionally, since Yao's GC technique is used, IS does learn certain structural information about C's query and QC's policy, namely their circuit topologies. However, the individual search terms (i.e., keywords) and logical gates are hidden (as Yao GC provides SPF-SFE, leaking only the circuit's topology).

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

Malicious-Client Vulnerabilities in Blind Seer

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

The Solution

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

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

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

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

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

Universal query circuit. IS can play the role of generator in the query evaluation protocol without knowing the query by using a *universal circuit.* A universal circuit $UC_{\mathcal{F}}$ is a well-known construction that can simulate any circuit C in the family $\mathcal{F}$. Specifically, it is a circuit that will take as input the description of any circuit $C \in \mathcal{F}$, any input x, and will output $C(x)$. There are many constructions of universal circuits UC_k that can simulate any circuit of size k [Val76; KS08b]. This is a very powerful tool for PF-SFE since it can be used to hide everything about the private function except its size. Indeed, a universal circuit provides exactly what we need in terms of swapping the role of the circuit generator to be input provider, while maintaining circuit privacy. Unfortunately, however, constructing a general universal circuit UC_k results in a significant increase in circuit size, and thus a very high overhead in performance when evaluating it through the Yao GC technique.

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

Our construction of $UC_{\mathcal{T}}$ from $\mathcal{T}$ increases the number of gates by a factor two, but the extra gates are *all* XOR gates. Thus, when using the *free-XOR* technique [KS08a], the cost of securely computing $UC_{\mathcal{T}}$ with Yao GC has practically the same cost as securely computing any monotone circuit C with topology $\mathcal{T}$.

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

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

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

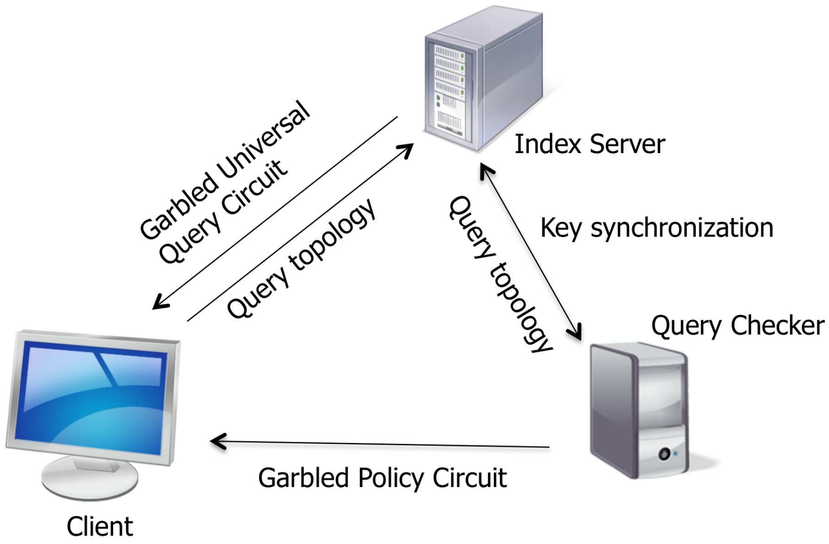


Figure 5.1: Malicious-client secure protocol overview.

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

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

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

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

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

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

We formalize and prove security properties of the solution in Section 5.5. These properties are proved with respect to a malicious client adversary and a semi-honest index server adversary. The definitions are general enough so that open ended design decisions do not invalidate any of the proofs (e.g. where to evaluate the PC), and are intended to elucidate the tradeoffs of such decisions.

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

5.3 Semi-Private Function Evaluation

Private Function Secure Function Evaluation (PF-SFE) is a two-party functionality in which a private function known only to one party (the *selector*) is evaluated on private inputs of both parties, and nothing but the outputs and function size is revealed. *Semi-private function SFE* (SPF-SFE) [PSS09] is a generalization of PF-SFE in which the private function is chosen from any restricted class of functions known to both parties.

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

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

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

Yao Semi-Private Function SFE with Selector As Evaluator

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

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

Figure 5.2: Protocol for Yao SPF-SFE with Selector As Evaluator

Efficiency. Each gate of the original circuit is replaced with a cluster of 3 XOR gates and 1 OR gate. Thus, the number of non-XOR gates remains constant. The cost of applying the free-XOR GC protocol to $UC_{\mathcal{T}}$ is roughly the same as applying it to the original circuit, since no communication and no expensive cryptographic hash function are needed to generate/evaluate XOR gates.

Applications. The capabilities of the generator and evaluator in Yao GC are not symmetric, particularly when dealing with malicious adversaries. For instance, Protocol 5.2 is useful for *repeated* SPF-SFE, where the same private function is to be evaluated more than once on different inputs, or possibly given as input to other functions. The selector could cheat as the garbler by using different functions for each set of inputs. But when the selector is the evaluator, the garbler can enforce consistency of the function. A special case is the problem that we address in Blind Seer, where the private function (client's query) must be evaluated on the DB index and also supplied as input to the policy check. A second capability of the generator is to encrypt a message under an output key from the garbled circuit so that the evaluator can only decrypt the message contingent on the output of the function.

5.4 System Protocol

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

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

Stage 2: Client queries. C builds a logical circuit Q representing its query, sends Q's topology to IS, together with hashes of all the field names and field values used in Q. IS uses Q's topology to construct a *universal query circuit UQ* (Section 5.2) equivalent to Q with output keys out_u^{UQ}, $u \in \{0,1\}$.

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

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

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

Stage 6: Record retrieval. When a query is successful in satisfying both UQ and PC, C uses the output keys out_1^{UQ} and out_1^{PC} to obtain out_1, and decrypts $\mathtt{Enc}_{\mathsf{out}_1}(\tilde{R})$. Finally, C uses the decryption keys obtained from the server S to decrypt $\tilde{R}$. Note that C always asks for the decryption information from S and always receives an encrypted record from IS, however, it only is able to successfully decrypt the record if both UQ and PC evaluated to true.

Stage 1: Preprocessing

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

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

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

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

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

Hash function generation. S chooses random keys $k_c, k_s \leftarrow \{0,1\}^\lambda$. S sends k_c to C and QC, and sends k_s to IS. S then generates hash functions $\mathcal{H} = \{h_i : \{0,1\}^* \to [\ell]\}_{i=1}^\eta$. S can choose H_1, H_2 independently and set $h_i(x) = H_1(x) + i \cdot H_2(x)$ [KM08]. (ℓ is chosen to satisfy the desired false positive rate). We use the notation $\mathcal{H}(x) = \{h_i(x) : h_i \in \mathcal{H}\}$. Keyed hashes are derived naturally as $\mathcal{H}_k(x) = \mathcal{H}(k||x)$.

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

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

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

Prepare Decryption Keys. To reduce online query latency, the index server and the server precompute decryption keys. S holds the decryption keys $s_1, ..., s_n$ that it used to symmetrically encrypt the records. It also holds $\tilde{s}_i = \mathsf{Enc}_{pk}(s_i)$ for all i. S sends to IS these values $\tilde{s}_1, ..., \tilde{s}_n$. IS chooses a random permutation ψ, and random values $r_1, ..., r_n$. IS homomorphically computes $\tilde{s}_i + \mathsf{Enc}(r_i) = \mathsf{Enc}(s_i + r_i) = \tilde{s}'_{\psi(i)}$ for all i, and sends these value back to S. S decrypts and stores $s'_{\psi(1)}, ..., s'_{\psi(n)}$.

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

Stage 2: Client Queries

Every query in the Blind Seer DBMS can be represented as a monotone boolean logical formula over atomic search terms. There are three types of atomic search terms: keywords, ranges, and negations. However, in Blind Seer, any range query from a field with value range r is translated into a disjunction of $O(\log r)$ keywords queries. A negation of a keyword α is translated into a disjunction of two range queries (i.e. $x < \alpha$ `OR` $x > \alpha$), which is again converted into disjunctions of keyword queries. Thus, the final representation of a query is a monotone logical formula over solely keyword terms (see range/negation condition paragraph in Section 4.3.3 for details).

Query circuits. Queries are computed by *query circuits.* A query circuit transforms each atomic keyword term α into a small circuit computing the presence of α in an input Bloom filter. This is simply a conjunction of the bits at the η BF hash indices of α, but since C

and IS hold separate shares of the input Bloom filter, it is actually a conjunction of η XOR gates.

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

1. C sends $\texttt{topo}(Q)$ to IS.
2. For each keyword of the form $\texttt{field} : \texttt{value}$, C sends to IS the hashes: $H_{k_c}(\texttt{field})||H_{k_c}(\texttt{field} : \texttt{value})$.
3. IS generates a garbled universal circuit UQ from $\texttt{topo}(Q)$ as described in Section 5.2. Each gate in the *query formula* layer of Q is associated with a pair of keys in UQ. $\texttt{OR}$ maps to a 0-key, and $\texttt{AND}$ maps to a 1-key.
4. C does OT with IS to receive the keys corresponding to the gate values (i.e. $\texttt{AND}/\texttt{OR}$) of UQ.
5. IS computes the set of indices:
   $\mathcal{H}_{k_s}(H_{k_c}(\texttt{field})||H_{k_c}(\texttt{field} : \texttt{value}))$
   for each keyword $\texttt{field} : \texttt{value}$, and sends these back to C.

Stage 3: Policy evaluation

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

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

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

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

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

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

$$[\mathtt{enc}_{k_{\pi(1)}}(k^{\text{OUT}}_{b(f_{\pi(1)})}), \ldots, \mathtt{enc}_{k_{\pi(m)}}(k^{\text{OUT}}_{b(f_{\pi(m)})})]$$

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

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

Policy protocol. IS initiates the policy evaluation.

1. IS sends $\mathtt{topo}(Q)$ to C along with key pairs for the gate value wires of the garbled UQ (i.e. the key pairs for the inputs G_Q).

2. Given $\mathtt{topo}(Q)$, QC generates the policy circuit PC. The input to PC is $(G_Q, H_{k_c}(K_Q), H_{k_c}(F_Q))$.

3. QC generates a garbled circuit from PC. It sets the key pairs for the inputs G_Q using the key pairs received from IS, and it generates all other key pairs randomly (as in the usual Yao garbled circuit construction). It sends the garbled tables of PC to C, and sends the key pairs for all the input wires to IS. Additionally, it sends the key pair $\{\text{OUT}_0^{PC}, \text{OUT}_1^{PC}\}$ for the policy output wire to IS.

4. Using the table of keys received from QC, IS identifies the input keys to PC corresponding to inputs $H_{k_c}(K_Q)$ and $H_{k_c}(F_Q)$. IS sends these keys to the client. Note that the client has already received the keys for the inputs G_Q via OT in Stage 2 (these are the same input keys that C will use for evaluating UQ in Stage 5).

5. Finally, C uses the keys received from IS in (4) and the garbled tables received from QC in (3) to evaluate PC, and obtains the output policy key OUT_p^{PC}, $p \in \{0, 1\}$.

Stage 4: Tree traversal

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

1. C garbles Q and sends the garbled circuit to IS. C also sends keys for its own input bits.

2. IS executes OT with C to obtain Yao keys for its input bits, evaluates the garbled circuit, and sends the output key back to C.

3. C learns the output value from the output key.

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

Stage 5: Leaf nodes

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

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

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

1. IS generates a new garbled UC using fresh key pairs for all wires except the gate value input wires.

2. IS sends to C the keys corresponding to its input bits $\{\tilde{B}_v[i]\}_{i \in \mathcal{I}_\alpha}$ for each α.

3. C performs OT with IS to receive the keys corresponding to its input bits $\{F_k(v)[i]\}_{i \in \mathcal{I}_\alpha}$ for each α.

4. Finally, C evaluates the garbled UC and obtains an output key out_u^{UC}, $u \in \{0,1\}$.

C has already obtained an output key out_p^{PC}, $p \in \{0,1\}$, from PC in Stage 3. If $p = u = 1$, then C can successfully obtain out_1 and decrypt $\mathtt{Enc}_{\mathsf{out}_1}(\tilde{R}_i)$. Otherwise, C cannot feasibly deduce any information about $\tilde{R}_i$ other than its size.

Stage 6: Record retrieval

When C reaches a record R_i, IS will send it $\psi(i), r_i$, and $\mathtt{Enc}_{\mathsf{out}_1}(\tilde{R}_i)$. C then sends $\psi(i)$ to S, who sends back $s'_\psi(i)$. C obtains $s_i = s'_{\psi(i)} - r_i$. If C successfully decrypted $\mathtt{Enc}_{\mathsf{out}_1}(\tilde{R}_i)$ to obtain $\tilde{R}_i$ in Stage 5, then C now decrypts $\tilde{R}_i$ using s_i, and obtains the record R_i. Otherwise, C cannot feasibly deduce any information about R_i other than its size.

5.5 Security and Privacy Analysis

Privacy and security in the Blind Seer system was previously achieved with respect to semi-honest static adversaries. An ideal functionality $\mathcal{F}_{db}$ was defined, and included a leakage profile describing the precise information that is leaked to each of the parties C, S, and IS. A standard simulation argument in the semi-honest static adversary model attests that Blind Seer securely realizes $\mathcal{F}_{db}$ (Figure 4.4).

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

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

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

The modified leakage to IS is precisely the pairs of hashes $H_{k_c}(\texttt{field})||H_{k_c}(\texttt{field} : \texttt{value})$. The additional leakage allows IS to correlate common fields of different keywords in the queries. In the original Blind Seer, IS learned the BF hash indices of the keywords, i.e. $\mathcal{H}(\texttt{field} : \texttt{value})$, but could not correlated common field patterns. The modified leakage to QC is the topology of the query. We note that it is straightforward to make the above modifications to the leakage profile in the simulation argument.

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

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

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

Indistinguishability vs. simulation. Our analysis will use indistinguishability games rather than the simulation paradigm. Some of the primitives we implemented are not simulatable in malicious and/or concurrent settings, specifically Naor-Pinkas OT, which we use during the multi-threaded traversal of the Bloom filter. This technical issue could be resolved by implementing a UC-secure OT primitive such as PVW [PVW08] instead of Naor-Pinkas [NP01] and using simulatable UC-secure OT extension protocols[1], or the server S could actually distribute random OTs during the preprocessing phase.

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

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

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

[1] There are various OT extension protocols offering full simulation security in the malicious adversary model that also have constant amortized cost [HIKN08; NNOB12], but these protocols still have constant factor overheads that are relatively expensive.

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

Proof. Note that the protocol $\text{OT-EXT}_m^{p(k)}$ is a randomized functionality that does not take any external input. Internally, the `Receiver` samples $\mathbf{r} \leftarrow \{0,1\}^{p(k)}$ and $\mathbf{T} \leftarrow \{0,1\}^{k \times k}$, and `Sender` samples $\mathbf{s} \leftarrow \{0,1\}^k$. An adversary $\mathcal{A}$ can therefore simulate any session of $\text{OT-EXT}_m^{p(k)}$, producing a view that is statistically indistinguishable from its view of the real session. If $\mathcal{A}$ is active and substitutes arbitrary input $(\mathbf{r}', \mathbf{T}')$ in place of the `Receiver`'s randomly sampled input, it can still produce a statistically indistinguishable view of the session by using the same $\mathbf{r}', \mathbf{T}'$ and randomly sampling $\mathbf{s}$ for `Sender`'s input.[2]

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

Recall that our $\text{OT-EXT}_m^{p(k)}$ is Nielsen's OT extension protocol for malicious receivers [Nie07], using Naor-Pinkas to instantiate the base OT protocol. Nielsen gave a concrete analysis showing that $\mathcal{A}$ cannot output both e_i^0 and e_i^1 for any i with probability greater than $\mathsf{negl}(k)$ when the base OT is instantiated with an ideal OT box. However, the starting point of Nielsen's analysis only assumes that the output of the base OT is correct, and that $\mathbf{s}$ remains uniformly random in the view of $\mathcal{A}$. $\mathbf{s}$ is the choice vector of the `Sender` acting as receiver in the base OT protocol. Sequential or parallel Naor-Pinkas OT guarantees both correctness and the property that $\mathcal{A}$'s views of the protocol (as sender) executed with different choice vectors of the receiver are statistically indistinguishable [NP01]. Equivalently, conditioned on $\mathcal{A}$'s view, $\mathbf{s}$ remains statistically indistinguishable from a uniformly random vector. It follows that the adversary cannot gain a non-negligible advantage when the ideal OT box is replaced with Naor-Pinkas in Nielsen's OT extension (as otherwise this protocol could be used to distinguish $\mathbf{s}$ from random). ■

[2] Note that this is different than *simulatability*, where the simulator in an ideal world secure execution of the protocol is required to simulate the attacks of any adversary in the real world with the same results.

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

Client query privacy

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

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

Leakage to IS in each query. The leakage to IS includes the BF-search tree traversal paths, the topology of the query $\mathtt{topo}(Q)$, and the pairs of hashes $H_{k_c}(\mathtt{field})||H_{k_c}(\mathtt{field} : \mathtt{value})$ for each of the client's keywords $\mathtt{field} : \mathtt{value}$ included in the query. This leakage reveals to IS when two different keywords in the query share the same field. (In contrast, the previous design only leaked hashes of the keywords in the query, and so IS only learned when full keywords repeated).

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

There we consider the views of both parties, or a third environment party observing the system, while here we are only concerned with a single party's view.

Policy and query soundness

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

Definition 4. *Let $\Sigma(D, \lambda)$ denote a DBMS mechanism executed on input database D with security parameter λ. We define the following game $\mathsf{Game}_{\Sigma}^{\mathtt{PAYIND}}(\mathcal{A}, \lambda)$ played with an adversary $\mathcal{A}$.*

$\mathsf{Game}_{\Sigma}^{\mathtt{PAYIND}}(\mathcal{A}, \lambda)$*:*

– *$\mathcal{A}$ chooses $D_0 = (R_0^1, ..., R_0^n)$ and $D_1 = (R_1^1, ..., R_1^n)$ where $R_b^i = (\mathcal{L}_b^i, \mathcal{M}_b^i)$ and $\mathcal{M}_0^i = \mathcal{M}_1^i$ $\forall i$.*

– *Sample $b \in \{0, 1\}$ uniformly at random.*

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

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

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

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

Theorem 3. *An execution of* mBS *on any policy $\mathbf{p}$ and query $\mathbf{q}$ such that $\mathbf{p}(\mathbf{q}) = 0$ is payload indistinguishable.*

Proof. We prove the claim by reduction to the semantically secure encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ used by mBS.

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

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

Any record payload $\mathcal{L}$ that $\mathcal{A}$ receives is encrypted using Enc with the key $\mathsf{out}_1^{PC} = \mathsf{Gen}(1^k)$. We have seen that $\mathcal{A}$ cannot compute out_1^{PC} when $\mathbf{p}(Q) = 0$ except with some negligible probability $\epsilon(k)$. Thus, it is easy to see that $\mathbf{Adv}_{\Sigma}^{\mathtt{PAYIND}}(\mathcal{A}, \lambda) \leq \mathbf{Adv}_{\Pi}^{\mathtt{IND}}(\mathcal{A}, \lambda) + \epsilon(k) < \mathsf{negl}(k)$ by semantic security of Π (cf. Appendix 2.5). ■

Theorem 4. *For any query* $\mathbf{q}$*, let* $\bar{\mathbf{q}}$ *denote its negation so that* $\bar{\mathbf{q}}(D)$ *is the set of records that fail* $\mathbf{q}$*. An execution of* mBS *on any query* $\mathbf{q}$ *is* $\bar{\mathbf{q}}$*-payload indistinguishable.*

Proof. For any record $R = (\mathcal{M}, \mathcal{L})$ that $\mathcal{A}$ receives, $\mathcal{L}$ is encrypted using Enc with a key out_1^{UQ} that is output by the garbled UQ circuit evaluated on M. As noted in the analysis of policy soundness (Theorem 1), the query $\mathbf{q}$ is uniquely determined by $\mathcal{A}$'s messages to IS except with probability negligible in k, the security parameter of $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

We show that if $\mathbf{q}(\mathcal{M}) = 0$, i.e. $\bar{\mathbf{q}}(\mathcal{M}) = 1$, then $\mathcal{A}$ cannot obtain out_1^{UQ} with probability greater than $\mathsf{negl}(\lambda)$ where λ is the minimum of k and the Bloom filter FPR used at the leaf nodes of the tree index. In the $\bar{\mathbf{q}}$-payload indistinguishability game, the payloads $\mathcal{L}_0^i$ and $\mathcal{L}_1^i$ of records R_0^i and R_1^i are identical unless $\bar{\mathbf{q}}(\mathcal{M}_0^i) = \bar{\mathbf{q}}(\mathcal{M}_1^i) = 1$. Once we establish that $\mathcal{A}$'s probability of obtaining out_1^{UC} when $\bar{\mathbf{q}}(\mathcal{M}) = 0$ is negligible in λ, $\bar{\mathbf{q}}$-payload indistinguishability reduces to the semantic security of Π, as in Theorem 1.

Consider $\mathcal{A}$'s evaluation of UQ on record R. As with the policy, $\mathcal{A}$ can only obtain one valid set of input keys, and can only compute one valid output key, except with probability

$\mathsf{negl}(k)$. However, unlike the policy evaluation keys, the keys that $\mathcal{A}$ obtains to evaluate UQ are not solely determined from the information that defines the query circuit Q. The reason is that $\mathcal{A}$ has some control over the DB inputs to UQ. Recall that for each BF node B_v, the client C holds a mask string $\mathsf{m}_v = F_k(v)$ and IS holds $\tilde{B}_v = B_v \oplus \mathsf{m}_v$. The circuit UQ computes XORs of bits taken from $\mathtt{m}$ and B_v. Nothing prevents $\mathcal{A}$ from flipping bits in m_v when evaluating UQ.

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

$\mathbf{z}_\alpha$ is actually a random variable over the randomness in the initialization of B_v. The bits of B_v were set via insertions of keywords (distinct from α) using $\mathcal{H}_{k_s}$, where k_s is secret from $\mathcal{A}$, and $\mathcal{H}$ is a family of random oracles by assumption. Therefore, $\mathbf{z}_\alpha$ is distributed independently from $\mathcal{A}$'s view of the protocol. Recall that the BF parameters are set so that the FPR is $2^{-\eta}$. Together with the simplifying assumption that the bits z_{α_i} are η-wise independent, the FPR implies that $Pr[z_{\alpha_i} = 1] = 1/2$ for each i, and that the min entropy $\mathbf{z}_\alpha$ is $2^{-\eta}$.[3] ■

False positives. The above theorems (policy soundness and query soundness) do not account for false positives in the result set $\mathbf{q}(\mathrm{D})$ due to false positive hits in the Bloom filters representing the indexed records. This threat becomes negligible when the FPR of the Bloom filter is made negligible. Even a malicious client who has learned some bits of the Bloom filter (e.g. from previous queries) and attempts to deliberately choose a query

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

that incurs a false positive has negligible advantage over the FPR because our Bloom filters use non-invertible (cryptographic) hash functions.

Policy compliance indistinguishability

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

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

Definition 5. *We define the policy compliance indistinguishability game* $\mathsf{Game}_{\Sigma}^{\mathtt{PCIND}}(\mathcal{A}, \mathbb{E}_{\mathbf{q}}, \lambda)$ *as follows.*

$\mathsf{Game}_{\Sigma}^{\mathtt{PCIND}}(\mathcal{A}, \mathbb{E}_{\mathbf{q}}, \lambda)$*:*

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

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

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

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

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

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

Theorem 5. mBS *satisfies policy compliance indistinguishability with respect to* $\equiv_{\mathbf{q}}$.

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

5.6 Implementation

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

OT pool. One important component of our system is the OT pool. This pool contains preprocessed Oblivious Transfers that are used in garbled circuit evaluations. The OT pool is filled regularly using the OT extension protocol of [IKNP03; Nie07]. We use the Naor-Pinkas [NP01] OT protocol as a base for OT extension.

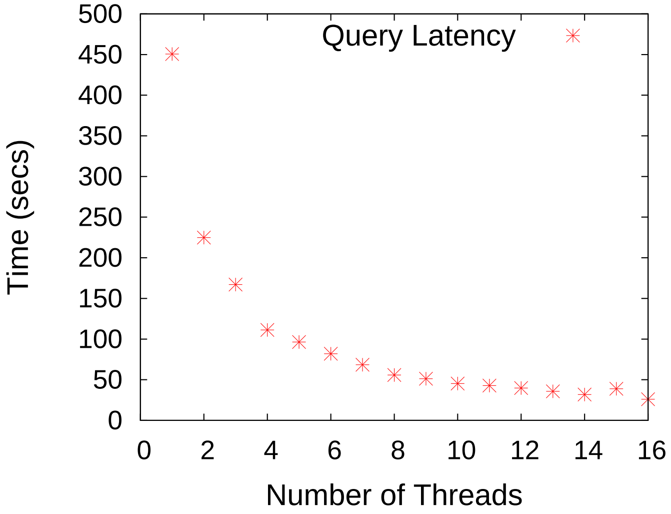


Figure 5.3: Query Latency versus number of threads on a 10^7 record database. Run on a Boolean query with individually frequent terms but with sparse aggregate results.

Parallelism within queries. The most important efficiency improvement comes from query parallelization. While the original Blind Seer implementation supported parallelization between queries to improve throughput (Parallelization paragraph Section 4.6), the new system supports multi-threaded evaluation of the tree structures within a single query to improve latency. Instead of having one global OT pool for all threads, each thread has its own OT pool. This significantly decreases the standby time of filling a single OT pool, as well as bypassing synchronization issues. We used the Intel Threading Building Block library to implement most of the parallelization used in the system.

Figure 5.3 shows query latency time plotted against the number of threads for the query

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

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

Cryptographic Tools. Our system requires pseudorandom bits, used primarily for garbling circuits. We avoid expensive system calls to /dev/urandom by implementing a PRG using AES as a building block. All AES operations were performed using 128-bit key length.

We used SHA256 for hashing, and implemented the keyed hash for keyword ingestion as an HMAC using SHA256 as the underlying hash function. Since OpenSSL uses the AES-NI instruction, we used this library to implement our basic cryptographic primitives. Our system also depends on operation on a group to which DDH intractability applies. We used the standard quadratic residues subgroup of $\mathbb{Z}_p$, where p is a 2048-bit strong prime[4]. The group operations were implemented using the GNU Multiple Precision library.

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

5.7 Evaluation

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

Our system was compared against MySQL (having separate indexes for each field) using a special-purpose testbed implemented by Lincoln Labs specifically for this purpose. Each party ran on a separate machine equipped with two Intel Xeon X5650 processors of 2.66Ghz and 12M cache, 96GB RAM[5]at 1066 MHz, and Broadcom 1GB Ethernet NICS with TOE each. Index Server and Server each had a 20TB raid array attached also. All machines ran 64-bit Ubuntu 12.04LTS as base OS.

[4] $p = 2 \cdot q + 1$ where q is also prime

[5] Although we ran the client on a machine with 96GB RAM, the client's actual memory consumption in

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

Boolean SELECT-id Queries. We compare the performance of our system against MySQL. Figures 5.4, 5.5, and 5.6 show the query latency for single-term, 3-term conjunctions (with two low-entropy terms and one medium-entropy term), and 3-term disjunctions as SELECT-id queries plotted against the number of records satisfying them. While original Blind Seer was 15 times slower than MySQL, our implementation manages to be only 2-3 times slower than MySQL for the case of single terms queries and 3-6 times slower for conjunctive and disjunctive queries on 3 terms. This saving is due to parallelization and low overhead of our new construction. Note that these queries are SELECT-id, and thus delivering a small payload. The relative overhead of our system would decrease with larger payloads since this cost is constant for both systems.

Single-term SELECT-* Queries. We measured query latency of our system and MySQL for queries returning 100KB records. Our system, as well as original Blind Seer, performs better (compared to MySQL) when the records retrieved are bigger. In this case, the standby time required to submit records' data dominate the network usage. We can observe from the results shown in Figure 5.7 that our system is only 10% slower than MySQL in this setting.

our system is insignificant. The client only needs to store a key and pre-processed OTs. In our experiments, the client used approximately 200MB of RAM.

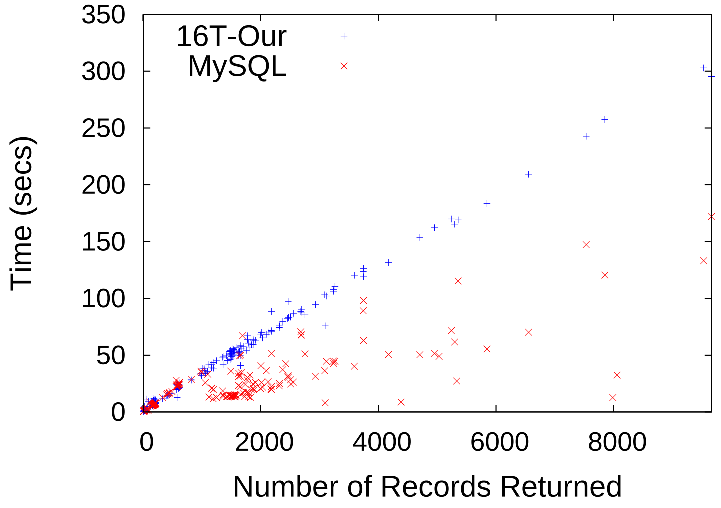


Figure 5.4: Single-term SELECT-id performance against number of records returned for our system and MySQL.

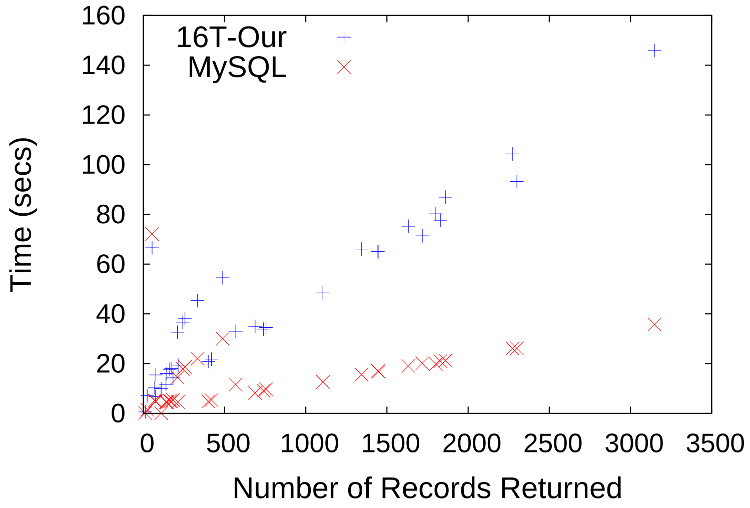
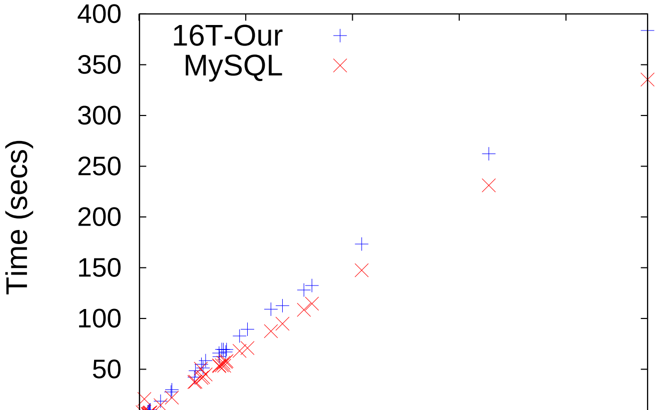


Figure 5.5: 3-terms conjunction SELECT-id queries performance against number of records returned for our system and MySQL.



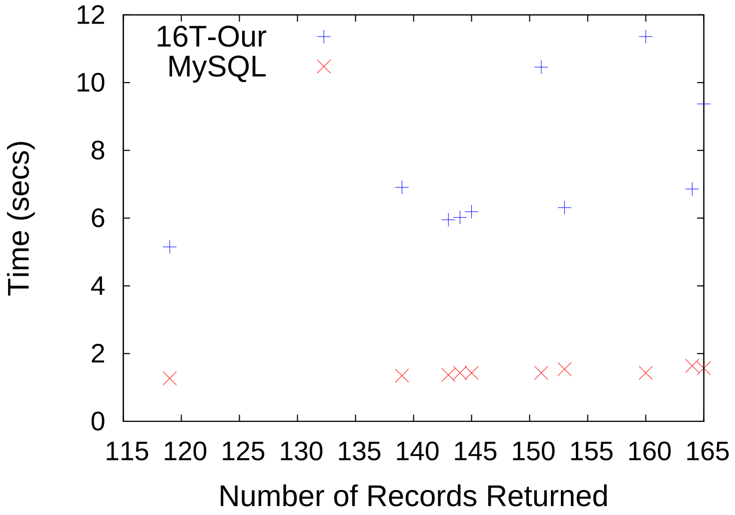


Figure 5.6: 3-terms disjunction SELECT-id queries performance against number of records returned for our system and MySQL.

Overhead of malicious-client security. As a validation of the costs of our malicious-client algorithms, we also compare the performance of our malicious-client secure Blind Seer to that of the original Blind Seer. To eliminate extraneous implementation performance variables, we compare the two designs by running our current implementation of Blind Seer with and without the design changes introduced in this work. In particular, the design differences potentially affecting performance include two additional hash function calls per keyword, the new semi-private SFE using a universal circuit, an additional Oblivious Transfer per gate of the query circuit, and one additional symmetric encryption and decryption per record returned to C. The experiment consisted of running single-threaded SELECT-id queries for single-term, 3-term boolean queries containing a conjunction and a disjunction, and range queries (which are each translated to a 5-7 term disjunction) over a *1-million* record database. The results are shown in figures 5.8, 5.9, 5.10. As expected, the design changes for achieving malicious-client security incur no significant overhead. The costs of the design changes for malicious-client security are proportional to the query size and number of leaf nodes reached in the Bloom filter tree index traversal, and not the total number of tree nodes processed. Thus, the performance gap will be even smaller on large databases when the number of leaf nodes reached is small compared to the number of

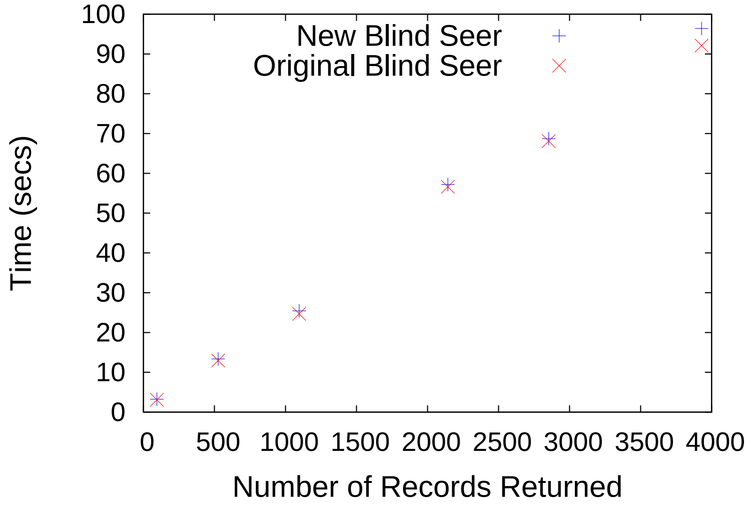


Figure 5.8: Performance of our prototype against original Blind Seer for single-term queries.

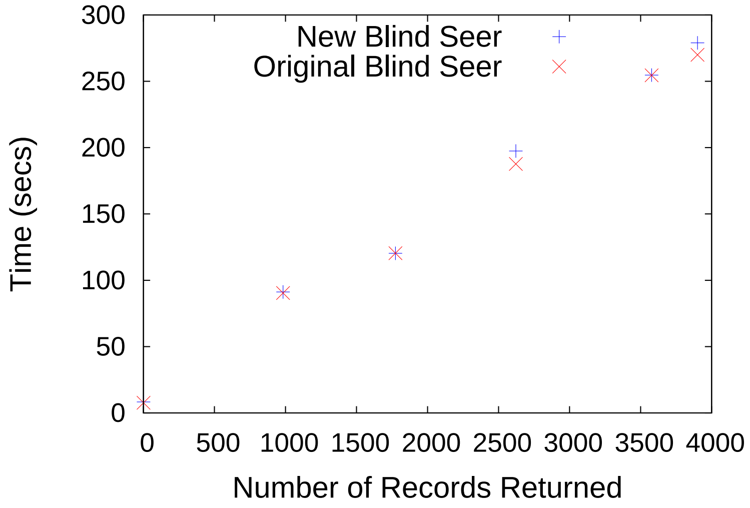


Figure 5.9: Performance of our prototype against original Blind Seer for 3-term boolean queries.

internal nodes processed, which we expect to often be the case in practice.

Caching. We did not run experiments on repeated queries. On repeated queries, MySQL benefits from server-side caching of query results. In our system, the client caches result sets (database record indices) locally.

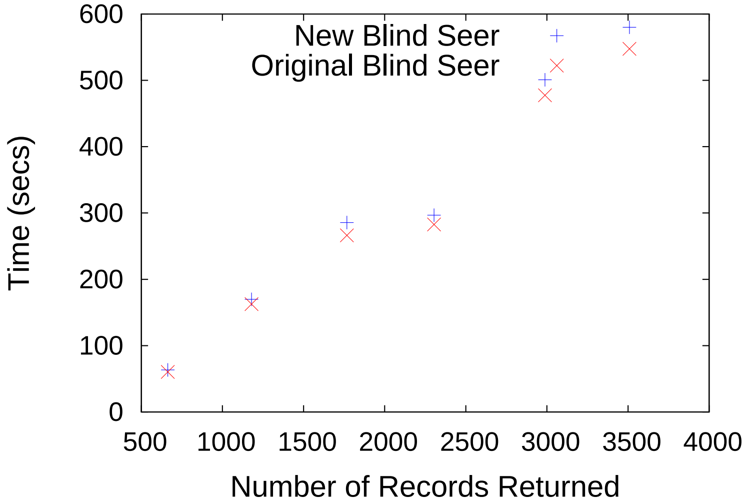


Figure 5.10: Performance of our prototype against original Blind Seer for range queries.

Chapter 6

Low-Leakage Private Search of Boolean Queries

6.1 Introduction

In the previous two chapters we presented a private search system called Blind Seer. We have seen that Blind Seer manages to be very efficient and allows for powerful queries. However, to acomplish high efficiency the scheme accepts some privacy loss on the client's queries and server's data. In this Chapter we are mainly concern in how much the storage server (a third party) learns about the client's queries and the database. While the encryption of records and the index's data helped by hidding the content stored on the server, the client privacy is still vulnerable since its access pattern to index and data is revealed to the server. A recent work [IKK12] has demonstrated that this leakage can be substantial even in the simplest search scenario of exact match such as keyword search. For more complex types of queries, such as Boolean queries (as supported by Blind Seer), this leakage can have even more serious security implications.

This chapter is motivated by the lack of a good grasp on analyzing the above privacy leakage in the Boolean search protocols such as Blind Seer and OSPIR-OXT [JJK$^+$13]. Here we propose a new construction that adopts the general approach of combining access pattern hiding properties of ORAM together with small secure computation steps, as done in the general-purpose secure two-party computation scheme of section 3. This approach was also taken by Gentry et al. [GHJR15] to construct a single keyword search scheme. We go further and focus on Boolean queries, and we develop tailored and optimized solutions for that. Our solution for secure search enables the same functionality for Boolean queries as Blind Seer, and it diminishes the access pattern leakage, while preserving the sublinear efficiency overhead for queries that are executed in sublinear time in these protocols. As expected, the concrete efficiency overhead for our protocol increases compared with these solutions that reveal complete access patterns.

Although direct comparison with the work of Gentry et al.[GHJR15] is hard, since their work implements much simpler queries compared to our protocols, our protocols achieve much better efficiency for comparable functionality.

Setting. As in Chapters 4 and 5 we are interested in the Outsourced Symmetric Private Information Retrieval (OSPIR) setting [JJK+13]. It captures the scenario in which the data owner outsources the data to a server, and gives search capabilities to clients. Recall that such a setting involve three main actors: S, IS, and C. The server S owns the database $(D_i, W_i)_{i=1}^{d}$, builds an index and outsources the list of documents (D_i) and the index to the index server IS. The client C has a *query* $\phi(W)$ composed by a Boolean formula $\phi(\cdot)$ over a set of keywords W. C gets the set of documents D_i, whose associated searchable keywords W_i satisfy the query.

In a Setup phase, the server (or data owner S) on input DB, does some preliminary computation on the data and produces an *encrypted* database EDB and access parameters params. EDB is then given to the index server (IS). In the Search phase, a client (C) inputs a query q, S inputs params, and IS inputs EDB. After protocol execution, C obtains database records satisfying its query. Such a scheme was formally defined in section 2.7.

6.1.1 Contributions

In this chapter we provide a new Private Database Search mechanism supporing Boolean queries, that avoids the main leakage introduced by Blind Seer and OSPIR-OXT systems.

We introduce new cryptographic primitives we called Mul-OPRF and Masked MOPRF. We use this primive together with Oblivious RAM holding a Bloom filter tree to obliviously evaluate queries on each index node.

We provide a prototype implementation of the system, and give analysis of performance. Our experimental results shows that for simple queries, records can be returned within a few seconds.

To our knowledge, and despite generic protocols, this is the first Private Database Search scheme that uses ORAM techniques, and also allows for Boolean queries. The use of ORAM diminishes the leakage to the server holding the database and its index.

6.1.2 Background

Between the range of Private DB search schemes available (see Section 7), the HE-over-ORAM approach [GHJR15], and the Blind Seer and OXT-OSPIR[JJK+13] are of particular interest. First, these schemes focus on the delegated query scenario, that is, the client is not the owner of the data. Secondly, while HE-over-ORAM aims for a secure asymptotically sublinear solution for single keyword search, the Blind Seer and OXT-OSPIR systems focus on practicality: they both support a rich set of queries and their efficiency is close to the plaintext database case. Our goal is to build a system that lies in-between these systems in terms of the privacy vs. efficiency trade-off.

OSPIR-OXT and Blind Seer. The first solutions for the OSPIR setting was proposed by Jarecki et al. [JJK+13](OSPIR-OXT) and Pappas et al. [PKV+14] (Blind Seer). Although they solve the same problem, they provide very different approaches. OSPIR-OXT is an extension of the OXT searchable encryption scheme [CJJ+13]. This solution allows for Boolean queries in Searchable Normal Form $(t_1 \wedge \phi(t_2, ..., t_n))$, and runs in time proportional to the number of records satisfying the term t_1. The solution is based on an inverted index approach, which is used to search information about the leading term t_1. This information is used then to search for the records satisfying the sub-queries $t_1 \wedge t_i$. A completely different approach was taken in Blind Seer. Instead of using an inverted index, Blind Seer builds a Search Tree on the searchable keywords of the database. Each leaf of the tree is associated with a record in the database, and each internal node holds to a *masked* Bloom filter containing the searchable keywords of the records in its subtree. Hence, a Boolean formula is answered by following paths root-to-leaves where the nodes' Bloom filter satisfy the query.

The above two systems are incomparable in terms of leakage since their underlying data structures are completely different. Blind Seer, though, has the advantage that the search procedure does not reveal partial evaluation results.

HE-over-ORAM Database Search. Gentry et al. [GHJR15] proposed a private DB system with no leakage based on ORAM and Somewhat Homomorphic Encryption. ORAM is used to protect the client's access patterns and the owner's data from the server. To protect the database information from the client, data is also encrypted using a variation of a Somewhat Homomorphic Encryption Scheme that enables Equal-to-Zero and Comparison operations. These operation enable the client to blindly perform ORAM operations until the requested value is found. Although this work shows the feasibility of the HE-over-ORAM approach, it has significant limitations in efficiency and functionality. In terms of efficiency, their experimental results shows that it requires 30 minutes to execute a single keyword query on a 2^{22} record database. In terms of functionality, the system only allows single keyword queries, and conjunction may be enable by a trivial addition of the keywords into the database index.

Approach. We use the Bloom filter Search Tree of Blind Seer as our search structure, but storing the encrypted data and its index in ORAM structures at the server. Then, we give the ORAM access parameters to the client, as done in the HE-over-ORAM scheme. To avoid the case where the client learns more information than strictly necessary, the actual data in ORAM should be encrypted in a special way. While this is done using Somewhat Homomorphic Encryption by Gentry et al. [GHJR15], we provide a new encoding scheme that allows parties to securely evaluate an index node, revealing to the client only the necessary information to continue the search procedure. We accomplish this by using a novel protocol for conjunctive query evaluation on specially encrypted Bloom filters. This protocol is then extended to handle queries in Disjunctive Normal Form.

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

Notation.

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

6.2 Cryptographic Primitives

In this section, we describe some cryptographic primitives we use in the construction of our private search scheme. These primitive are presented in a modular way, and can be of independent interest.

Oblivious PRF. First, our solution uses an Oblivioius Pseudorandom Function (OPRF). It involves two parties, $\mathcal{C}$ having input m and $\mathcal{S}$ having input k, who jointly evaluate a pseudorandom function $F_k(m)$, while keeping their input private. A simple construction proposed by Jarecki and Liu [JL10] uses the Hashed Diffie-Hellman PRF ($F_k(m) = \mathsf{Hash}(m)^k$). The protocol is described in Figure 6.1. $\mathcal{C}$ starts by sampling a uniformly random invertible exponent α and sends $X = \mathsf{Hash}(m)^\alpha$ to $\mathcal{S}$. $\mathcal{S}$ replies with $Y = X^k$, and $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}} = \mathsf{Hash}(m)^k$.

Two-party Protocol OPRF

Parameter: A random hash function $\mathsf{Hash} : \{0,1\}^* \to \mathcal{G}$.

Input $\mathcal{C}$: $w \in \{0,1\}^*$. $\mathcal{S}$: k.

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

Figure 6.1: The Two-Party Protocol OPRF

MUL-OPRF. In a simple variation of the above primitive, C inputs a set $\{m_1, ..., m_n\}$, S inputs the secret key k, and $\mathcal{C}$ receives as output $\prod F_k(m_i)$. We call this new primitive MUL-OPRF. We obtain a secure protocol for this primitive by using $\prod \mathsf{Hash}(m_i)$, as the random hash function in the protocol of Figure 6.1.

Masked MOPRF. For the purpose of the construction in Section 6.3, we require a slight modification on the above MUL-OPRF functionality. We call this new primitive a Masked MOPRF. In this primitive, $\mathcal{C}$ gets the result of the MUL-OPRF protocol masked with a random value R, while $\mathcal{S}$ obtains the mask R. This simple modification is achieved by adding one extra message in the protocol (Figure 6.2). The server starts by sampling a uniformly random exponent β, and sending $W = g^\beta$ back to $\mathcal{C}$. $\mathcal{C}$ responds with $X = (W \cdot \prod \mathsf{Hash}(m_i))^\alpha$ for the uniformly random invertible exponent α. $\mathcal{S}$ replies with $Y = X^k$, and outputs $R = g^{\beta \cdot k}$. $\mathcal{C}$ outputs $Z = Y^{\alpha^{-1}} = R \cdot \prod \mathsf{Hash}(m_i)^k$.

Security. The security of the MUL-OPRF protocol follows directly form the security of the Hashed DH Oblivious PRF protocol [JL10] by using $\prod \mathsf{Hash}(\cdot)$ as the random function in the random oracle model. The security and correctness of the Masked MUL-OPRF protocol follows directly from DDH assumption since it implies that the value $g^{\beta \cdot k} \times \prod F_k(m_i)$ is pseudo-random even given g^β (and even if the adversary somehow knows $\prod F_k(m_i)$).

Two-party Protocol Masked-MOPRF

Parameter: A random hash function $\mathsf{Hash} : \{0,1\}^* \to \mathcal{G}$.

Input $\mathcal{C}$: $\{m_i\}_{i \in [n]}$. $\mathcal{S}$: k.

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

Figure 6.2: The Two-Party Protocol Masked-MOPRF

6.3 Scheme

In this section, we present our private search scheme. Our ultimate goal is a secure search functionality that enables oblivious delegated queries on outsourced data to a party we call index sever (IS), where the data owner (S) can obliviously issue a search token to a client (C) for a query that remains hidden from the owner. Given this search token, C should only learn the data matching the query, while minimizing the information that IS learns about the issued queries (we analyze what IS learns formally in Section 6.4).

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

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

In Section 6.3.1, we introduce techniques that allow for secure delegated queries leveraging Bloom filter tree and ORAM approaches. We first show how to generate query tokens without reveling the client's query to either party. We then describe how to securely evaluate single term queries, conjunctions and DNF queries on each Bloom filter, allowing C to traverse the tree and find the documents satisfying its query. Finally, we describe how C can decrypt the retrieved documents without any party knowing the identifiers of these documents.

In Section 6.3.2, we present the complete construction of our private search scheme.

6.3.1 Building Block Techniques

Obliviously Generating Search Tokens. Before C can evaluate its query, it needs to be able to compute Bloom filter indices corresponding to the terms in the query for each Bloom filter in the tree. For security reasons, these indices are derived from a PRF, whose key is held by the database owner: each term is mapped through the use of this PRF to a *search token*, which is then hashed to get the Bloom filter indices. Similarly to Jarecki et al. [JJK$^+$13], we use the Hashed Diffie-Hellman PRF $H_{s_{\mathsf{bf}}}(m) = \mathsf{Hash}(m)^{s_{\mathsf{bf}}}$ as our PRF to compute search tokens for each term. This PRF is computed via the oblivious protocol in Figure 6.1, hiding the key from the client, and the keyword from the server.

Single Term Queries. For single keyword queries, $\phi(w) = w$, the client needs to learn if all the bits queried in a Bloom filter are set. For this purpose, we leverage the $\mathsf{Masked\text{-}MOPRF}$ protocol making use of the underlying PRF to encrypt each bit. We encode a bit to an arbitrary element in the range group of the PRF F, and use F to encrypt the bit. Let g be a uniformly random group generator that we keep secret to the client (and given to the index server); we map a bit b to g^b and encrypt it as $r_i \leftarrow \{0,1\}^\lambda$, $\langle g^b \cdot F_k(r_i), r_i \rangle$ for position i in the Bloom filter[1]. The client and the index server use the $\mathsf{Masked\text{-}MOPRF}$ primitive described

[1] The values r_i accross different Bloom filters are sampled independently.

in Section 6.2 to evaluate a Bloom filter query that reveal no additional information to the client as follows. They execute the Masked-MOPRF protocol with inputs a set of $\{r_i\}_{i\in S}$, for the client, and a PRF key k for the index server, where S is the set of BF indices corresponding to the query. The client obtains as output $R \cdot \prod_{i\in S} F_k(r_i)$, while the index server obtains the random value R. Next, the client computes $\prod_{i\in S}\left(g^{b_i} \cdot F_k(r_i)\right)$, and using its output from the Masked-MOPRF protocol obtains

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

The index server now provides $H(R^{-1} \cdot g^h)$ so that the client can do the matching evaluating $H(R^{-1} \cdot g^{\sum_{i\in S} b_i})$ and doing the comparison. The random element R prevents the client from learning the value $g^{\sum b_i}$. The hash over the server-side matching key $R^{-1} \cdot g^h$, impede canceling out R^{-1}, and hence computing $g^{h-\sum b_i}$. Note that if the generator g was known to the client, he could multiply $R^{-1} \cdot g^{\sum_i b_i}$ with g^j for $j \in [h]$ and compare it against $H(R \cdot g^h)$, hence learning $\sum b_i$ exactly, hence we require the generator to be secret to the client. We will prove in section 6.4.4 the security of this Bloom filter matching procedure.

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

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

- For each conjunctive clause C the client and the index server execute the protocol for the single query matching (without the final stage where index server reveals the hashed matching key), and the client learns the value

  $$R_C \cdot g^{\sum_{i \in S_C} b_i}$$

  where S_C denotes the set of Bloom filter positions to be checked for clause C.

- Each of the resulting values is blinded by public a random exponent L_C, and the final matching evaluation key is computed as

  $$\prod_{C \in \phi} (R_C \cdot g^{\sum_{i \in S_C} b_i})^{L_C}$$

  .

- The index server computes the set $\mathsf{Matching}$ of all the possible matching values, and the client *obliviously* does the matching. There are several ways to do the matching. One possibility is to hash and permute all the matching keys, before sending them to the client. Another approach is through a Bloom filter.

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

Record Retrieval. After finding the list of identifiers of records satisfying the query, C can actually retrieve them by querying the ORAM that contains the records. However, as mentioned earlier, in the case of the index ORAM, each ORAM access can potentially reveal records that do not satisfy C's query. Hence, each document should be encrypted under a key unknown to C. However, the client should be allowed to decrypt the satisfying records. To support this feature, during the preprocessing phase, S samples a secret key s_{r} and, using again the Hashed Diffie-Hellman PRF, it derives for each document D_i a document encryption key $k_i \leftarrow H(i)^{s_{\mathsf{r}}}$. Them, in the online phase, S and C execute the OPRFprotocol in Figure 6.1 to derive the decryption keys: C inputs the document identifiers, and S inputs s_{r}.

Procedure Setup

Public: Group $\mathcal{G}$ of prime order q and public generator g'.

Parameters: A security parameter λ, false positive rate fp, the tree degree d.

Input: Database $\{D_i, W_i\}_{i=1}^{D}$.

1. Sample $s_{\mathsf{k}} \leftarrow \{0,1\}^{\lambda}$, and compute sets $\{\tilde{W}_i = \{H_{s_{\mathsf{k}}}(w) | w \in W_i\}\}$.
2. Sample $s_{\mathsf{r}} \leftarrow \{0,1\}^{\lambda}$, and compute $k_i \leftarrow F_{s_{\mathsf{r}}}(i)$, $\tilde{D}_i \leftarrow \mathsf{Enc}_{k_i}(D_i)$.
3. $\langle \mathcal{H}, \mathsf{BFT} \rangle \leftarrow \mathsf{BFTBuild}(\{\tilde{W}_i\}, \mathsf{fp}, d)$.
4. $\langle \mathsf{EBFT}, s_{\mathsf{bf}}, g \rangle \leftarrow \mathsf{BFTEncrypt}(\mathsf{BFT}, 1^{\lambda})$.
5. $\langle \mathsf{param}_I, \mathsf{struct}_I \rangle \leftarrow \mathsf{OSetup}(\mathsf{EBFT}, 1^{\lambda})$.
6. $\langle \mathsf{param}_D, \mathsf{struct}_D \rangle \leftarrow \mathsf{OSetup}(\{\tilde{D}_i\}, 1^{\lambda})$.

Output:

S : $\langle s_{\mathsf{k}}, s_{\mathsf{r}} \rangle$

C : $\langle \mathsf{param}_I, \mathsf{param}_D \rangle$

IS : $\langle s_{\mathsf{bf}}, \mathsf{struct}_I, \mathsf{struct}_D \rangle$.

Figure 6.3: The preprocessing procedure Setup

6.3.2 Final Scheme

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

The setup phase is formally described by procedure Setup in Figure 6.3. We describe next the basic procedures used by Setup:

- $\mathsf{BFTBuild}(\{\tilde{W}_i\}_{i=1}^{D}, \mathsf{fp}, d)$**.** Let BFT be a balanced d-ary tree of D leafs. Let $L = \lceil \log_d D \rceil$ be the height of the tree. We build the tree level by level, starting from the bottom level L. We then proceed recursively until reaching the root of the tree. Let $N_L = \max |W_i|$. Using N_L and fp, compute Bloom filters length n_L and number of Bloom filter hash function h_L . Then, we sample h_L independent hash function $\mathcal{H}_L = \{H_1, ..., H_{h_L}\}$ with image $\{0, 1, ..., n_L - 1\}$. For each $i \in [D]$, we build a Bloom filter B_i (using $\mathcal{H}_L$) inserting the elements of $\tilde{W}_i$. We maintain each Bloom filter in a unique leaf of BFT. The internal nodes of the tree are built recursively as follows: we associate each node at level ℓ with the keywords held in its children. That is, for each internal node, we build a Bloom filter that contains the elements from all its d children. Return $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, ..., \mathcal{H}_L\}$ and tree BFT. To avoid revealing the level of the nodes being evaluates, we force the set of hash functions to be of the same size $h = h_L = |\mathcal{H}_L| \approx \log(1/\mathsf{fp})$.

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

Search. C inputs a DNF formula $\phi(\mathbf{W}) = C_1 \vee C_2 \vee \cdots \vee C_q$ on keywords in $\mathbf{W}$. The client reveals the query *topology* (number of clauses and size of each clause) to IS. C and S then execute the protocol in Figure 6.1 to obtain search tokens for each keyword in each clause . For each clause C in the query, C (or IS) uniformly samples L_C from $[q]$ and sends it to IS (C). C and IS then start the tree traversal protocol. For each node being evaluated, both parties proceed as follows:

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

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

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

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

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

After the tree traversal, C gets the indices of all documents satisfying the query. It can obtain the documents by querying the documents ORAM structure. To obtain the document decryption keys, C and S execute protocol OPRF, where S inputs key s_{r} and C inputs the document identifiers. A formal description of the search protocol is presented in Figure 6.4.

Protocol Search

Public: Group $\mathcal{G}$ of prime order q.

Inputs.

C**:** DNF query $\phi(\mathbf{W})$, and ORAMs' parameters param_I, param_D.

IS**:** Key s_{bf}, group generator g, and structures struct_I, struct_D.

S**:** Keys s_{k} and s_{r}.

I. Tree Traversal

1. C. Set $\mathsf{OUT}_I = \emptyset$. $\forall$ `clause` $C \in \phi(\cdot)$, $\text{Ł}_C \overset{\$}{\leftarrow} [q]$. Send $(\mathsf{topo}(\phi), \{L_C\}_{C\in\phi(\cdot)}$ to IS.
2. C-S. $\forall C \in \phi(\cdot)$, $\forall w \in C$ $(\tilde{w}, \perp) \leftarrow \mathsf{OPRF}(w, s_{\mathsf{k}})$.
3. C-IS start the tree traversal. C set queue $Q = \{(0,0)\}$.
   (a) if Q is empty go to **Document Retrieval**, otherwise $v \leftarrow \mathsf{Q.pop}()$.

   (b) For each clause C derive from v and from Bloom filter bits positions $\mathcal{H}_{v.\mathsf{level}}(\tilde{w})$ the set of addresses $\{i\}$ of the Bloom fiter bits in the index, and do:

   i. $\{(e_i, r_i) \leftarrow \mathsf{ORead}(i, \mathsf{struct}_I)\}$.

   ii.$\langle \pi_C, R_C \rangle \leftarrow \mathsf{Masked\text{-}MOPRF}(\{r_i\}, s_{\mathsf{bf}})$.

   iii. C computes $\zeta_C = \pi_C^{-1} \cdot \prod_i e_i$.

   (c) C. Set $\mathsf{FinalKey} = \prod_C \zeta_C^{L_C}$.

   (d) IS. Compute set $\mathsf{Matching}$, and corresponding Bloom filter $\mathsf{BF} \leftarrow \mathsf{BFBuild}(\mathsf{Matching}, 2^{-\lambda})$. Send BF to client.

   (e) C. If $\mathsf{BFMatch}(\mathsf{BF}, \mathsf{FinalKey})$ do:

   if $v.\mathsf{level} = L$ **then** $\mathtt{OUT}_I = \mathtt{OUT}_I \cup \{v\}$,

   else $\mathsf{Q} = \mathsf{Q} \cup \mathsf{Children}(v)$.

   (f) Go to 3a

II. Document Retrieval. C sets $\mathsf{OUT}_D = \emptyset$. For each index $i \in \mathsf{OUT}_I$ do:

1. C-IS. $\tilde{D}_i \leftarrow \mathsf{ORead}(i, \mathsf{ORead}\rangle, \mathsf{struct}_D)$.
2. C-S. $(k_i, \perp) \leftarrow \mathsf{OPRF}(i, s_{\mathsf{r}})$
3. C. $D_i \leftarrow \mathsf{Dec}_{k_i}(\tilde{D}_i)$. $\mathsf{OUT}_D \leftarrow \mathsf{OUT}_D \cup D_i$

Output: Client output set OUT_D.

Figure 6.4: The Protocol Search

6.4 Security

Our protocol guarantee security in the semi-honest case against the client, server and data owner. We discuss next the correctness for our protocols, define the leakage to each party during the execution of the protocol. We then provide formal security definitions and proofs of security.

6.4.1 Protocol Correctness

Without considering false positives, the correctness of our protocol follows from the correctness of ORAM, Bloom filter Search tree, and node evaluation. Note that if a clause C is satisfied by the keywords held in the node, then the $\mathsf{FinalKey}$ obtained by the client is $(R_C \cdot g^{|C| \cdot h})^{L_C} \cdot \prod_{C' \neq C} (R_{C'} \cdot g^{\nu_{C'}})^{L_{C'}}$ for $\nu'_C \in \{0, ..., |C'| \cdot h\}$ for every C'. This key belongs to set $\mathsf{matching}_C$, and hence the node evaluation is correct.

We now consider sources of false positives. In our protocol there are three posibilities for false positives: the false positive rate of the BF instances that we use in the search tree, collisions between node evaluation values that correspond to satisfying and unsatisfying assignments, and false positive in the final matching of the $\mathsf{FinalKey}$ against set $\mathsf{matching}$.

Note that false positives that occur in Bloom filters look-ups at intermediate levels of the search tree may result in longer search time for the query; however, only the false positives at the tree leaves may result in the client obtaining a document that does not match its query.

We analyze the probability of each of the three independent events that cause a false positive Bloom filter match. If we use Bloom filters with the false positive rate fp at the leaves of the search tree, then for a DNF query of n clauses that consist of $|C_1|, \ldots, |C_n|$ terms (or keywords), the probability for a false positive match coming from a false positive on the underlying terms is $\sum_{i=1}^{n} \mathsf{fp}^{|C_i|}$.

The second event that causes an incorrect match on a Bloom filter DNF query is a collision on the node evaluation final keys. For each clause C_i, let r_{C_i} such that $g^{r_{C_i}} = R_{C_i}$, then we can look at the evaluation key as $g^{\sum_{i=1}^{n} L_{C_i}(r_{C_i} + \nu_{C_i})}$ where ν_{C_i} ranges between 0 and $|C_i| \cdot h$. In a matching evaluation key, at least for some i ν_{C_i} equals $|C_i| \cdot h$, whereas in a mismatching key, none of the ν_{C_i} equals $|C_i| \cdot h$. Let $\{\nu_{C_i}\}_i$ be the number of bits set for each clause C_i in a satisfying assignment (that is, for some i $\nu_{C_i} = |C_i| \cdot h$), and let $\{\nu'_{C_i}\}$ the numer of bits set for each clause in an unsatisfying one. Then, the probability that the corresponding keys collide is the probability that $\sum_i (\nu_{C_i} - \nu'_{C_i}) L_{C_i} = 0$, which is negligible in $|\mathbb{Z}_q| = \mathsf{negl}(\lambda)$ by the Schwartz-Zippel lemma [Sch80; Zip79].

The third source of false positives is the final key matching procedure. In Figure 6.4 we use a negligible false positive rate of $2^{-\lambda}$, we note however, that this can be set as a tunnable parameter of the system.

Therefore, the false positive rate is bounded by $\sum_{i=1}^{n} \mathsf{fp}^{|C_i|} + \mathsf{negl}(\lambda)$

6.4.2 Protocol Leakage

In this section, we describe the leakage to each party in the setup and search phases of the protocol.

Leakage During Setup. After Setup procedure is called by S, IS gets the encrypted database, and PRF key s_{bf}. The encrypted database consists of ORAM structures struct_I and struct_D. The structures reveal the number of elements contained in the ORAMs, and the size of the elements in them. Hence, the leakage to the server after the setup phase is the Bloom filter tree total size $|\mathsf{BFT}|$, the size of the database records $|D_1|$, and the number of records in the database $D = |\mathsf{DB}|$. We call this leakage $\mathcal{L}_{\mathsf{setup},is}$.

Before each query search starts the client needs to know the structure of the Bloom filter tree, and the number of records in the database D. We call this leakage $\mathcal{L}_{\mathsf{setup},c}$.

The data owner leakage $\mathcal{L}_{\mathsf{setup},o}$ is empty since C and IS have no input.

Leakage During Search. During the search procedure, there is leakage to all parties. We describe them individually next:

- IS starts by learning the query topology $\mathsf{topo}(q)$ (number of keywords in each conjunctive clause). During the tree traversal, it learns the number of Bloom filter tree node evaluations N. Finally, index server learns the number of records retrieved by the client $|\mathsf{DB}_{\mathsf{fp}}(q)|$. We call this leakage $\mathcal{L}_{\mathsf{search},is}$.

- S learns the number of terms in the query $|q|_{\mathtt{terms}}$, and the number of records retrieved by C $|\mathsf{DB}_{\mathsf{fp}}(q)|$. We call this leakage $\mathcal{L}_{\mathsf{search},o}$.

- C learns the tree traversal pattern `TraversalPattern`, composed by the identifiers of the evaluated nodes and the result of each evaluation. We call this leakage $\mathcal{L}_{\mathsf{search},c}$.

6.4.3 Security Definitions

We define security in the standard ideal vs. real world paradigm assuming semi-honest adversarial behavior. In both worlds we let an adversary to choose the protocol inputs (the database and the set of queries). In the real world the participant follows the prescribed protocol, but we let the adversary to see the *view* (i.e. incoming messages and internal randomness) of a single party. In the ideal world the adversary interacts with a simulator of the protocol which doesn't know the information we are protecting from the adversary. If the view of the adversary in the real world is indistinguishable from the view produced by the simulator we say that the protocol is secure, since whatever the adversary learns about the private information in the real execution of the protocol can also be learned from the simulator which have no information about the private information.

In our construction, however, we do allow for some limited privacy leakage. For example, the index server knows how many nodes of the Bloom filter search tree the client evaluates. To formally capture this leakage, we give the simulator access this leakage. Therefore, security in this setting implies that an adversary cannot learn anything beyond the specified leakage and output of the queries. In what follows we define the real and ideal procedures of an adversary attacking a participant P:

$\textsc{real}^{\Pi}_{\mathcal{A},P}(\lambda)$**.** Run $\mathcal{A}(1^\lambda)$ to obtain database $\mathsf{DB} = \{D_i, W_i\}_{i=1}^{D}$. Execute Π with honest parties allowing $\mathcal{A}$ to adaptively select queries at his choice. After each query give the view of party P to $\mathcal{A}$. When $\mathcal{A}$ halts, output the entire view given to $\mathcal{A}$. We denote the output as $\textsc{real}^{\Pi}_{\mathcal{A},P}(\lambda)$.

$\textsc{ideal}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{query}}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$**.** Run $\mathcal{A}(1^\lambda)$ to obtain database $\mathsf{DB} = \{D_i, W_i\}_{i=1}^{D}$. Run $\mathcal{S}_P(1^\lambda, \mathtt{setup}, \mathcal{L}_{\mathtt{setup}}(|DB|))$ to produce output to $\mathcal{A}$. Have $\mathcal{A}$ repeatedly select queries q and run $\mathcal{S}_P(1^\lambda, \mathtt{query}, \mathcal{L}_{\mathtt{query}}(q))$ and give output to $\mathcal{A}$. We denote the output as $\textsc{ideal}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{query}}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$.

Definition 6. *We say that an OSPIR protocol* Π *is* $(\mathcal{L}_{\mathsf{setup}}, \mathcal{L}_{\mathsf{search}})$*-adaptively secure against* *semi-honest party P if for every polynomial time stateful algorithm $\mathcal{A}$ there exists a polynomial time stateful algorithm $\mathcal{S}_P$ such that the random variables* $\text{IDEAL}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{search}}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$ *and* $\text{REAL}^{\Pi}_{\mathcal{A},P}(\lambda)$ *are computationally indistinguishable, for all sufficiently large λs.*

We also consider a weaker *selective* security definition, in which the adversary can adaptively select queries traversing the database index, but can only request the satisfying documents once all the queries have been executed. The real and ideal procedures are redefined in this setting next.

$\text{REAL}^{\Pi}_{\mathcal{A},P}(\lambda)$**.** Run $\mathcal{A}(1^\lambda)$ to obtain database $\mathsf{DB} = \{D_i, W_i\}_{i=1}^D$. Execute $\Pi.\mathsf{Setup}$ with honest parties. Then, for each adversary's query, execute $\Pi.\mathsf{Search.TreeTraversal}$ giving the view of party P to $\mathcal{A}$. When $\mathcal{A}$ finishes its queries, parties execute $\Pi.\mathsf{Search.DocumentRetrieval}$ for each of the document satisfying the client's queries, giving the view of party P to $\mathcal{A}$. Output the entire view given to $\mathcal{A}$. We denote the output as $\text{REAL}^{\Pi,\mathsf{selective}}_{\mathcal{A},P}(\lambda)$.

$\text{IDEAL}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{query}}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$**.** Run $\mathcal{A}(1^\lambda)$ to obtain database $\mathsf{DB} = \{D_i, W_i\}_{i=1}^D$. Run $\mathcal{S}_P(1^\lambda, \texttt{setup}, \mathcal{L}_{\texttt{setup}}(DB))$ to produce output to $\mathcal{A}$. Have $\mathcal{A}$ repeatedly select queries q_i and run $\mathcal{S}_P(1^\lambda, \texttt{traversal}, \mathcal{L}_{\texttt{search}}(q_i))$ and give output to $\mathcal{A}$. Then, execute $\mathcal{S}_P(1^\lambda, \texttt{Document Retrieval}, \{\mathsf{DB}_{\mathsf{fp}}(q_i)\}_i)$ We denote the output as $\text{IDEAL}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{query}},\mathsf{selective}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$.

Definition 7. *We say that an OSPIR protocol Π is $(\mathcal{L}_{\mathsf{setup}}, \mathcal{L}_{\mathsf{search}})$-selectively secure against semi-honest party P if for every polynomial time stateful algorithm $\mathcal{A}$ there exists a polynomial time stateful algorithm $\mathcal{S}_P$ such that the random variables* $\text{IDEAL}^{\Pi,\mathcal{L}_{\mathsf{setup}},\mathcal{L}_{\mathsf{search}},\mathsf{selective}}_{\mathcal{A},\mathcal{S}_P}(\lambda)$ *and* $\text{REAL}^{\Pi,\mathsf{selective}}_{\mathcal{A},P}(\lambda)$ *are computationally indistinguishable for all sufficiently large λs.*

We will prove next that our system is *adaptively* secure against semi-honest owner and server (separately) behaviors, and *selectively* secure against semi-honest client behavior. We are not able to prove adaptive security for the semi-honest client case for the following reason. On each ORAM access the client sees several encrypted records (in addition to the one being queried); if these records are never requested by the client, the encryption can be easily simulated by encrypting a dummy element. On the other hand, if some of these records are requested later, the client should see the actual encrypted records, since at some point it will be able to decrypt them. Hence, we allow the simulator to "build" the ORAM containing the records, and simulate the queries after it knows which records are being satisfied by the queries (including false positives).

6.4.4 Proofs of Security

Let Π be the protocol described in Figures 6.3 and 6.4. Let $(\mathcal{L}_{\mathsf{setup},is}, \mathcal{L}_{\mathsf{search},is})$, $(\mathcal{L}_{\mathsf{setup},o}, \mathcal{L}_{\mathsf{search},o})$ $(\mathcal{L}_{\mathsf{setup},c}, \mathcal{L}_{\mathsf{search},c})$ as described in section 6.4.2.

Theorem 6. Π *is* $(\mathcal{L}_{\mathsf{setup},is}, \mathcal{L}_{\mathsf{search},is})$*-secure against semi-honest adaptive index servers.*

The security of the scheme against semi-honest adaptive index servers follows almost directly from the security of the underlying ORAM scheme used, and the security of the Masked-MOPRF protocol. In fact, all incoming messages that are not ORAM are independent and uniformly random elements of $\mathcal{G}$; hence, they are perfectly simulatable. For ORAM accesses, we use the security of the ORAM scheme. In particular, we assume the existence of simulators S_{OSetup}, S_{OAccess} that simulates the OSetup procedure, and OAccess procedure given only access to the number of elements in the ORAM and the size of the elements. [2]

Proof. We show that the index server's view given to $\mathcal{A}$ in the real world experiment is computationally indistinguishable from the one produced by the following algorithm S in the ideal world experiment.

$S(1^\lambda, \texttt{setup}, \mathcal{L}_{\mathsf{search},is}(\mathsf{DB}))$**:** Given index size $\ell = |\mathsf{BFT}|$, use the simulator S_{OSetup} to build an ORAM structure for ℓ elements of size $\mathsf{len} + \lambda$, where $\mathsf{len} = \mathsf{poly}(\lambda)$ is the bit-length of elements in $\mathcal{G}$. Given the number of records D and size of the records, use the ORAM simulator S_{OSetup} to build parameters and structure of another ORAM. Give both ORAM structures and a uniformly random key sk to the adversary. Finally, derive number of Bloom Filter hash functions.

$S(1^\lambda, \texttt{query}, \mathcal{L}_{\mathsf{search},is}(\mathsf{DB}, q))$**:** Given query topology $\mathsf{topo}(q)$, the number of nodes traversed N, and the number of records returned $|\mathsf{DB}_{\mathsf{fp}}(q)|$, simulate each node evaluation as:

- Derive number of ORAM lookups from $\mathsf{topo}(q)$ and number of hash functions used. Then use ORAM access simulator to perform the ORAM lookups.

[2] For the Path-ORAM, for example, the S_{OSetup} can create an ORAM structure out of dummy elements, and S_{OAccess} can be simulated by requesting random paths root to leaf.

- Simulate $\mathsf{Masked\text{-}MOPRF}$ protocol by sampling uniformly random element in $\mathcal{G}$.

The index server's view is composed by ORAM lookups and incoming messages from the $\mathsf{Masked\text{-}MOPRF}$ protocol. The view's of this two procedures are independent since the element received in each the $\mathsf{Masked\text{-}MOPRF}$ protocol execution $(W \cdot \prod H(m_i))^\alpha$, for uniformly random α, is a uniformly random element in the group (assumed to be of prime order). Hence, the security of the scheme relies on the simulators S_{OSetup} and S_{OAccess}. We conclude that the adversary cannot distinguish between the execution of the real protocol from the execution of the ideal world. ■

Theorem 7. Π *is* $(\mathcal{L}_{\mathsf{setup},o}, \mathcal{L}_{\mathsf{search},o})$*-secure against semi-honest adaptive owners.*

Proof. The owner S is involved only in OPRF computations at query token generation, and the computation of records decryption keys. The view of S is $h_i \leftarrow H(x_i)^{r_i}$ for many uniformly and independent random $r_i \in \mathbb{Z}_q^*$. Since H is onto $\mathcal{G}\setminus\{1\}$, and $\mathcal{G}$ is of prime order, for every $x \in \{0,1\}^*$, $H(x_i)^{r_i}$ is uniformly and independently distributed in $\mathcal{G}$. Hence, the each invocation to the OPRF protocol can be perfectly simulated. The simulator only needs the number of terms in the query $|q|_{\mathtt{terms}}$, and the number of satisfying records $|\mathsf{DB}_{\mathsf{fp}}(q)|$. ■

Theorem 8. Π *is* $(\mathcal{L}_{\mathsf{setup},c}, \mathcal{L}_{\mathsf{search},c})$*-selectively secure against semi-honest clients.*

Proof. The leakage $\mathcal{L}_{\mathsf{setup},c}$ is composed by the number of documents in the database $D = |DB|$, the size of the documents $|D_1|$, and the size of the index structure. The leakage $\mathcal{L}_{\mathsf{search},c}$ is the tree traversal pattern for the query q. We next present the ideal world simulator S.

$S(1^\lambda, \mathtt{setup}, \mathcal{L}_{\mathsf{setup},c}(DB))$: Create an ORAM for the index using random elements $\{(e,r)\}$, where e is a uniformly random element of $\mathcal{G}$, and r is uniformly random in $\{0,1\}^\lambda$. Sample uniform exponents s_{k} and s_{bf} to be used as key for the token generation PRF and ORAM element encryption PRF respectively. Give ORAM parameter to the adversary and save the ORAM structure, and keys.

$S(1^\lambda, \mathtt{Tree\ Traversal}, \mathcal{L}_{\mathsf{search},c}, q)$: 1)Token generation: For each term t_i in the query sample a random invertible exponent r, and produce $H(t_i)^{r \cdot s_{\mathsf{k}}}$ as the incoming message. 2)Tree traversal: From tokens derive the address of the elements in the ORAM, and honestly perform lookups on the saved ORAM structure holding the simulated index. Let (e_i, r_i) be the random elements gotten in the lookup. For each clause C_j, simulate the $\mathsf{Masked\text{-}MOPRF}$ protocol, by sampling random element g^{β}, exponent α, and computing $g^{\beta \cdot \alpha \cdot s_{\mathsf{bf}}} \cdot \prod H(r_i)^{\alpha \cdot s_{\mathsf{bf}}}$ as the incoming message from the index server. The final step corresponds to the key matching procedure, in which the client's evaluation key is matched against the set of all possible matching keys. When the tree traversal pattern indicates that the node satisfies the query, compute the matching key as $\prod_j (\prod e_i \cdot g^{-\beta \cdot s_{\mathsf{bf}}} \cdot \prod H(r_i)^{-s_{\mathsf{bf}}})^{L_C}$ and sample uniformly random elements for the rest of the possible matching keys. Otherwise, all possible "matching keys" are sampled at random from the group. The keys are then inserted in a Bloom filter of false positive rate $2^{-\lambda}$.

$S(1^{\lambda}, \texttt{Document Retrieval}, \{\mathsf{DB}_{\mathsf{fp}}(q_i)\}_i)$. Sample a random exponent s_{r}, and encrypt each document in $\bigcup_i \mathsf{DB}_{\mathsf{fp}}(q_i)$ using $H(i)^{s_{\mathsf{r}}}$ as the respective key. Then, create an ORAM simulating the records' ORAM, in which we maintain the encrypted elements of $\bigcup_i \mathsf{DB}_{\mathsf{fp}}(q_i)$, as well encryptions of dummy records simulating the elements not requested by client. Give the ORAM parameters to the adversary, and then honestly perform ORAM queries for elements in $\bigcup_i \mathsf{DB}_{\mathsf{fp}}(q_i)$, giving the client's view to the adversary. To decrypt each record D_i, we simulate the OPRF protocol by sampling a random exponent r, and computing $H(i)^{s_{\mathsf{r}} \cdot r}$.

We now argue that the view simulated by the above algorithm is computationally indistinguishable from the client's view in a real execution of the protocol. Let's divide the client's view in the following groups:

1. ORAM access parameters.
2. OPRF messages for token generation and decryption keys.
3. ORAM elements gotten in lookups at tree traversal.
4. Masked MOPRF messages.
5. Hashed matching evaluation keys.
6. ORAM elements gotten in lookups for document retrieval.

7. Decrypted documents.

We first note that messages in groups 1, 2, 6, and 7 are independent from messages in groups 3, 4 and 5 corresponding to node evaluation procedures. Also it is easy to see that these first messages in the real world experiment are indistinguishable from the corresponding view produced by S in the ideal world. In fact, since ORAM access parameters only depend on the size of the data, S perfectly simulates them from the leakage profile. OPRF messages are perfectly simulated by sampling a random keys s_{k}, s_{r} and following the prescribed protocol. The document retrieval procedure is also correctly simulated by S since it generates the document's ORAM knowing which elements the client gets. Hence, security reduce to the security of the underlying encryption scheme for the records not retrieved by the client.

We now prove the messages in groups 3, 4, and 5 are correctly simulated by S. Assume without loss of generality that query correspond to a single term. In the following analysis, we do not include the encrypted bits gotten during ORAM lookup that do not correspond the Bloom filter bits we are evaluating. We can safely do so because these encryptions are independent from the rest of the messages, and the encryption scheme protects their values. Hence, we focus on the set I corresponding to addresses of Bloom filter bits derived from the query tokens.

The client's view on each node evaluation correspond to:

$$\texttt{ViewReal} = \{g^{b_i} \cdot F_{s_{\mathsf{bf}}}(r_i), r_i\}_{i \in I}, g^{\beta}, g^{\beta \cdot sk} \cdot \prod_{i \in I} F_{s_{\mathsf{bf}}}(r_i), H(g^{-\beta \cdot sk} \cdot g^{h})$$

The simulator produce the following view:

$$\texttt{ViewIdeal} = \{e_i, r_i\}_{i \in I}, g^{\beta}, g^{\beta \cdot sk} \cdot \prod_{i \in I} F_{s_{\mathsf{bf}}}(r_i), T$$

Where e_is are independent and uniformly random elements, and T is $H(g^{-\beta \cdot sk} \cdot (\prod_{i \in I} F_{s_{\mathsf{bf}}}(r_i)) \prod e_i)$ if evaluation is a match, and T is uniformly random if evaluation is a missmatch. Note that $g^h = (\prod_{i \in I} g^{b_i}) \cdot g^{h - \sum_{i \in I} b_i}$, and let's make the change of variables $e'_i = g^{b_i} \cdot F_{s_{\mathsf{bf}}}$ in `ViewReal`:

$$\texttt{ViewReal} = \{e'_i, r_i\}_{i\in I}, g^{\beta}, g^{\beta\cdot sk}\cdot\prod_{i\in I} F_{s_{\mathsf{bf}}}(r_i), H(g^{-\beta\cdot sk}\cdot\prod_{i\in I}(e'_i\cdot F_{s_{\mathsf{bf}}}(r_i)^{-1})\cdot g^{h-\sum_{i\in I} b_i})$$

Let's assume for now that e'_i is uniformly random. Them, if $h=\sum b_i$, then evaluation correspond to a match, and `ViewReal` is identically distributed to `ViewIdeal`. We next prove that if e'_i is uniformly random and $h > \sum b_i$, then there is no PPT distinguisher for `ViewReal` vs. `ViewIdeal`. In fact, a distinguisher would need to tell $H(g^{-\beta\cdot sk}\cdot\prod_{i\in I}(e'_i\cdot F_{s_{\mathsf{bf}}}(r_i)^{-1})\cdot g^{h-\sum_{i\in I} b_i})$ from uniformly random T. However, since H is random function, the distinguisher can only win if he is able to compute the input $g^{-\beta\cdot sk}\cdot\prod_{i\in I}(e'_i\cdot F_{s_{\mathsf{bf}}}(r_i)^{-1})\cdot g^{h-\sum_{i\in I} b_i}$. It is not hard to see that this is equivalent to computing $g^{h-\sum_{i\in I} b_i}$. Nevertheless, since g is a uniformly random generator of a prime-order group that is *hidden* from the client, the distinguisher cannot compute g^c with noticeable probability for any $c>0$. Hence, the views are indistinguishable.

We show now that $\{e'_i\}_{i\in I}$ is a pseudorandom element, and hence, the above argument holds.

We use an hybrid argument to get to `ViewReal` in which e'_i are uniformly random elements. We first use DDH assumption to change $g^{\beta\cdot s_{\mathsf{bf}}}$ with a random element R. Indeed, given public generator g', and g'^{α}, g'^{β}, and the challenge C, a distinguisher can simulate one of hybrids by sampling a private random exponent x to create a secret generator $g=g'^{x}$ and simulating the hybrid as $\{g^{b_i}\cdot F_{s_{\mathsf{bf}}}(r_i), r_i\}_{i\in I}, g^{\alpha}, C\cdot\prod_{i\in I} F_{s_{\mathsf{bf}}}(r_i), H(C\cdot g^h)$ and call a distinguisher for `ViewReal` vs. `ViewIdeal`, where $F_{s_{\mathsf{bf}}}(r)$ is implemented as $g^{\beta\cdot\texttt{hash}(r)}$, and $\texttt{hash}: \{0,1\}^{\lambda}\to\mathbb{Z}_p$ is a random oracle. Hence, we can change `ViewReal` to

$$\texttt{ViewReal}' = \{e'_i, r_i\}, g^{\beta}, R\cdot\prod_{i\in I} F_{s_{\mathsf{bf}}}(r_i), H(R^{-1}\cdot\prod(e'_i\cdot F_{s_{\mathsf{bf}}}(r_i)^{-1})\cdot g^{h-\sum b_i})$$

Let $Q = R\cdot\prod F_{s_{\mathsf{bf}}}(r_i)$. Then we can rewrite the above as

$$\texttt{ViewReal}' = \{e'_i, r_i\}, g^{\beta}, Q, H(Q^{-1}\cdot\prod e'_i\cdot g^{h-\sum b_i})$$

where Q is uniformly random. We now use the security of F to change $e'_i = g^{b_i}\cdot F_{s_{\mathsf{bf}}}(r_i)$ for a uniformly random elements e_i. In fact, a successful distinguisher D for `ViewReal` with e'_i and uniformly random e_i can be used to distinguish F from a uniformly random function. The distinguisher $D_{\mathsf{PRF}}^{O(\cdot)}$ for F is as follows: Sample uniformly random $\{r_i\}$, Q, g^β, and computing $e_i = g^{b_i} \cdot O(r_i)$, then output $b \leftarrow D(\{e_i, r_i\}, g^\beta, Q, H(Q^{-1} \cdot \prod e_i \cdot g^{h-\sum b_i}))$. Note that if oracle is $F_{s_{\mathsf{bf}}}(r_i)$, then the input produced to D is identically distributed to `ViewReal`'. On the other hand, if oracle is a random function f, then e_is are random as long as r_i does not repeat, but this happens only with negligible probability.

We conclude that if F is a PRF, DDH holds in the prime-order group, H is random function, and g is kept secret to the client, then `ViewReal` is computationally indistinguishable from the view produced by the PPT simulator S.

■

6.5 Implementation

We implemented a prototype of our system in Java that consisted of approximately 4K lines of code. We next list some choices we made during the development of our prototype.

Cryptographic Primitives. We used Java Cryptography Architecture to implement the required symmetric key operation in our system. We used SHA-256 as the underlying hash function. ORAM elements where symmetrically encrypted using AES-128. The group operations were instantiated using the subgroup of quadratic residues modulo $p = 2q + 1$, where p, q are both prime. The bit-length of p was chosen to be 1024. Large integers and operations were instantiated using `java.math.BigInteger` class.

Optimization. A common Bloom-filter-based index optimization packs together bits stored at the same position across different Bloom filters. In our case, whenever we visit a node, we need to check all its siblings Bloom filters at the exact same positions. With this consideration we packed the encrypted bits at the same position of all siblings nodes in a single ORAM block, reducing the number of ORAM read operations.

ORAM. We implemented the simple and efficient Path ORAM of [SvDS+13]. Each ORAM bucket (nodes in the ORAM tree) contained four elements. The ORAM stash was maintained entirely on the client. The lowest-level position map was set to maintain 1000 indices at the client side. Each element was between 1-2KB, containing "siblings" encrypted bits as explained above.

Private Set Membership Queries. At the end of each node evaluation, C and IS engage in a private set membership query, where IS inputs a set of all possible matching keys for the node, and C inputs the key computed. We implemented this protocol by having the server hash all the keys, randomly permute them, and send them to the client.

6.6 Evaluation

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

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

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

Conjunctions vs. Disjunctions. Figure 6.5 shows, in $\log_{10}$ scale, the latency time for conjunctions and disjunctions of sizes 1, 2, 3, 4, and 5 on a 100 K record database. We observe that while conjunctive queries run in a few seconds, disjunctive queries are exponentially more expensive. It is interesting to note that the number of ORAM queries performed by both types of queries is exactly the same; hence, the latency time is dominated by cryptographic operations and data transfer of the matching keys set. In the case of disjunctive queries, we also note that the use of multiple cores does not reduce the latency significantly (at most a factor of two for 24 cores). In the case of conjunctions, the evaluation is entirely sequential and the use of multiple cores has no effect.

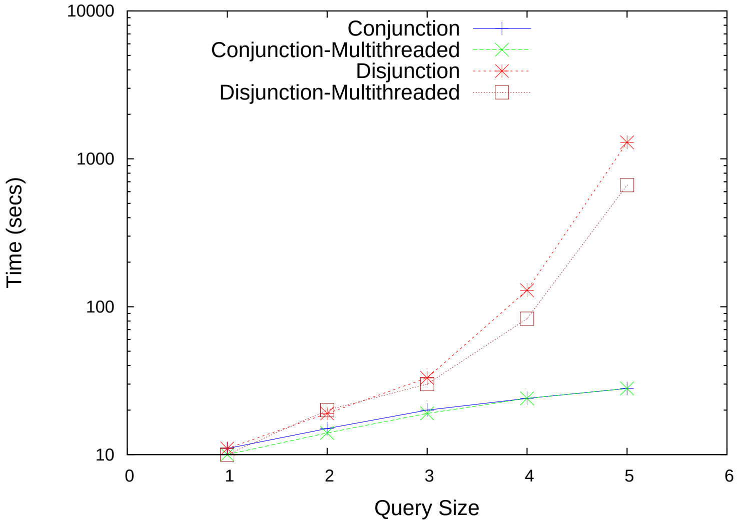


Figure 6.5: Latency of conjunctions and disjunctions of sizes 1, 2, 3, 4 and 5 for 100K records DB.

Varying Database Size. Figure 6.6 shows the latency for different DNF queries across databases of sizes 1K, 10K, and 100K. The difference between running a query on databases of varying sizes is captured in the number of nodes to be evaluated and the potentially larger ORAM size for larger databases. We observe the sub-linearity of our system's running time; increasing the database size by a factor of 10, increases the running time by a comparatively small amount, which is due to a single extra evaluation node and a larger ORAM structure.

ORAM vs. node evaluation. In table 6.1, the second and third columns illustrate the time our prototype spent in ORAM read queries and node evaluation, once ORAM queries have been performed. Since same-size queries require the same number of ORAM operation, the ORAM time is identical for same-size queries. Disjunctive queries, however, exhibit a much more expensive node evaluation execution, since they involve an exponentially large number of possible matching keys, which IS has to compute and hash individually. Moreover, the fourth column indicates that the network usage increases significantly with bigger disjunctions. The reason is that IS also needs to send the set of possible matching keys to C. In particular, for size-4 3-DNF, the network usage raises to 1GB, and we can

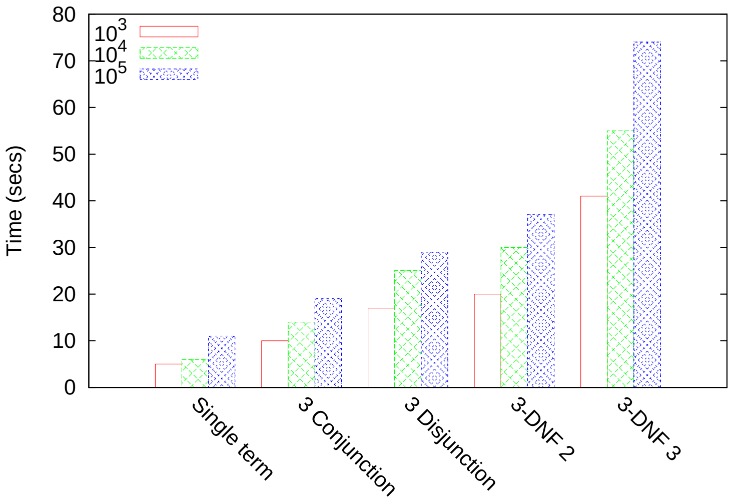


Figure 6.6: Query latency time for different-size DNF queries for databases of sizes 1K, 10K and 100K records.

infer that for these queries the index traversal will dominate the running time, considering queries that also return the records' payload.

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

Asymptotic Performance Analysis. The cost of our search procedure is proportional to the number of Bloom filter node visited. Each node evaluation consists of a stage of ORAM lookups and stage of matching keys calculation. Consider a general DNF formula consisting of k clauses, each of t_i terms. The number of ORAM lookups is $\sum_{i=1}^{k} t_i \cdot h$ each having $\mathsf{polylog}(D)$ index server-side lookups of size $\mathsf{polylog}(\lambda)$ each (by using Path-ORAM the $\mathsf{polylog}(D)$ term is of degree 2). At the same time, the number of possible matching keys to be computed is $\sum_{i=1}^{k} \prod_{j=1}^{i-1} t_j \cdot h \cdot \prod_{j=i+1}^{k} (t_j \cdot h + 1)$ ($< k \cdot (\mathtt{max}(t_i) \cdot h + 1)^{k-1}$), where each key can be computed by $2k$ exponentiations on the group [3]. Summarizing, the cost of

[3] Since keys have common terms, it is not necessary to compute $2k$ exponentiations for every key, and a much faster algorithm can be used.

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

Table 6.1: Latency in seconds of tasks in protocol and network usage per query on a 100K records DB.

each node evaluation is

$$C_{\mathsf{node}} = \mathsf{polylog}(\lambda)(\mathsf{polylog}(D) \cdot \sum_{i=1}^{k} t_i + \sum_{i=1}^{k} \prod_{j=1}^{i-1} t_j \cdot h \cdot \prod_{j=i+1}^{k} (t_j \cdot h + 1))$$

The number of nodes visited varies from query to query. For disjunctions, if a node matches the query, there must be *at least* a record in its subtree that matched the query. In this case, the worst case per record returned is when there is a single matching record, and hence the procedure needs to evaluate an entire path root-to-leaf to reach the record. Therefore, the worst-case per record cost $H \cdot C_{\mathsf{node}}$, where $H = \log_d(D)$ is height of the tree. On the other hand, it is specialy hard to analyse conjunctions since in the worst case, we can visit the entire tree, without finding a single matching record. For example, imagine a AND query of 2 terms, where the first term is present in half the records and the second term in the other half. However, as noted first in Chapter 4, the performance of AND queries is proportional to the least frequent term of the query in the database, which is optimal without an specialized index for conjunctions.

6.7 Conclusions

We proposed an encrypted search scheme that supports Boolean queries and enables delegated queries. Our system diminishes the leakage of existent solutions, preserving the sublinear search efficiency. Our construction integrates ORAM techniques with efficient search index structures leaking to the index server only the number of nodes visited in the search tree during the execution of a query. We also proposed a new protocol for oblivious product of pseudorandom function we called Masked-MOPRFthat allow us to securely evaluate Bloom filters. This protocol enables the delegated-query feature by disclosing only the match result. Finally, to hide the details of delegated queries from the data owner, we also used an oblivious search token generation.

We implemented our system prototype and ran it on a 100,000-record database. We showed that our system can handle conjunctive queries and small DNF formulas in 10-30 seconds. The sublinearity of our solution, also experimentally illustrated in Figure 6.6, allows us to extrapolate that queries on much larger databases (10^6, 10^7, and 10^8 records) will run in a few minutes. The cost of eliminating leakage is substantial; the Blind Seer and OXT-OSPIR systems manage to return records in less than a second for databases of size 10^8 records with a much larger number of searchable keywords. On the other hand, our system outperforms the secure single-keyword search of the HE-over-ORAM solution whose experimental results showed that their system answers a query in 30 minutes for 4×10^6 record databases. Therefore, our scheme provides a new tradeoff mark between privacy and efficiency.

To the best of our knowledge, our system is the first to apply ORAM techniques in private database search to reduce the leakage exhibited by other systems that allow Boolean queries. Our experimental results show that the use of ORAM *is not the bottleneck* for complex queries; hence, possible future work may focus on reducing the overhead of disjunction evaluation at each index node. In addition, we note that reducing the memory required to maintain our index is crucial for practical applications. This can be achieved by providing a better encoding scheme for the Bloom filter bits and optimizing the ORAM parameters. Another future direction is to augment our system with a private query authorization mechanism.

Part III

Final Remarks

Chapter 7

Related Work

7.1 Related Work

7.1.1 RAM based Secure Computation.

The first part of this thesis proposed a scheme for secure two-party computation that achieve sublinear time complexity, in an amortized sense. The main idea was to combine an ORAM scheme together with small generic secure computation steps.

This idea was first explored by Ostrovsky and Shoup [OS97] for the purpose of *private storage*. In their construction, there are three playes, a database owner and two non-colluding servers. One server simulates the ORAM server, and the ORAM state is shared between the two servers. An access to the data is done by having both servers execute an ORAM instruction using a secure computation protocol as we do. Although the construction of Chapter 3 follows the same paradigm of "ORAM + 2PC", the scenarios are quite different.

Another approach to using ORAM for secure computation was observed by Damgård et al. [DMN11], where the players share the the entire ORAM structure. Hence, all players require superlinear storage. In our construction, the client maintains only a polylogarithmic number of bits.

After the publication of our techniques [GKK$^+$12], RAM based secure computation has continued been estudied in [LO13b; LO13a; GGH$^+$13; GHL$^+$14]. In addition, the work of [KS14] implements several data structured using ORAM and uses them for secure compuation of specific functionalities. While all these schemes assumed semi-honest adversaries, the recent work of Afshar et al. [AHMR15] achieve malicious security in this model.

7.1.2 Encrypted Search

In the second part of this thesis we focused on the specific scenario of private DB querying.

The problem of private DBMS can be solved by general purpose secure computation schemes [GMW87; BGW88; Yao82; Yao86; LP09]. 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.

Private Information Retrieval protocols (PIR) [CGKS98] 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 [GIKM00] 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. [CGN97] 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 [Gen09] showed that FHE is theoretically possible. Despite this breakthrough and many follow up works, current constructions are too slow for practical use. For example, [GHS12] showed an implementation that homomorphically compute 720 AES blocks in two and a half days.

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 [PRV$^+$11; RVBM09] 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. A more efficient solution towards this end was proposed in [CLT11]. 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 systems, 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) [SWP00; Goh03; CM05; CGKO06; MS13; CJJ$^+$13; CK10]. Note that, although many of the techniques used in SSE schemes can be used in our scenario, the SSE setting focuses just on data outsourcing, and does not considering delegating search capabilities. That is, in SSE the data owner is the client, and so no privacy against the client is required. Additionally, SSE solutions often offer either a linear time search over the number of database records [SWP00; CM05; MS13], or a restricted type of client's queries [CGKO06; KP13], namely single keyword search or conjunctions. One exception is the recent SSE scheme of [CJJ$^+$13], which extended the approach of [CGKO06] 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, all of the solutions presented here 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 [ABC$^+$08; BBO07; BW07; BCOP04; SBC$^+$07]. Here, many users can use the server public key to encrypt their own data and send it to the server.

The CryptDB system [PRZB12] 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 [SWP00].

The closest to our setting is an extension of the OXT scheme [CJJ+13], called OSPIR-OXT [JJK+13], supporting the class of functions as OXT. A comparison between our work and OSPIR-OXT was described in Chapter 4, Section 4.1.2 and Chapter 6, Section 6.1.2.

Chapter 8

Conclusions

This thesis focused on achieving efficiency in secure computation protocols for the use in application involving high volumes of data. Hence, the primary goal was to overcome the linear time lower bound explained in the introducing chapter of this work.

By taking advantage of key properties in specific scenarios, and making use of modern cryptographic techniques we provided ad-hoc protocols achieving various level of security, efficiency and functionality.

We noted first that in many scenarios the function to be computed applies repeatedly over same or slightly modified data. This allowed us to preprocess the data so that future function evaluations over it can be computed securely, achieving sublinearity in an amortized sense. We gain further efficiency in some settings by making use of the help of a semi-honest third party. Furthermore, in some settings allowing some controlled privacy leakage is worth the extra efficiency that the protocol can achieve.

In addition to several novel techniques, we made extensive use of Oblivious RAM protocols and Yao's Garbled Circuits. ORAM was used to hide the access pattern to the data (hence protecting the data as well as the queries from the server holding it). To additionally protect data from a client, we evaluated privacy critical small subroutines using state-of-the-art Yao's Garbled Circuit construction. ORAM was used in Chapters 3 and 6, and Yao's Garbled Circuit technique was used in the protocols of Chapters 3, 4, and 5.

8.1 Summary of contributions

Part I . In the first part of this work we studied the problem from a theoretical perspective. We asked whether a two player functionality that can be computed insecurely over a Random Access Machine in *sublinear* time, can also be computed efficiently, hence overcoming the linear time complexity lower bound. We answered this positively in Chapter 3 by composing an Oblivious RAM (ORAM) protocol with small generic secure computation steps. The construction compiles a RAM program on virtual addresses into an ORAM program on physical, server-side addresses in which the "next instruction" of the program is computed securely via some generic Secure Computation protocol. Since the "next instruction" is usually a small program, and Oblivious RAM incurs a polylogarithmic overhead, the solution guarantees that if the original RAM program uses space s and requires t steps, then the compiled program runs in time $t \cdot \mathsf{polylog} s$, and uses $\mathsf{poly(s)}$ space.

We went further and provided an optimized protocol that takes advantage of an specific construction of ORAM, and uses Yao's' GC technique for the small secure computation steps. We provided a prototype of such scheme and showed its efficiency experimentally. We compared our protocol executing a binary search against a linear time secure search using Yao's protocol, and showed that our solution outperforms the trivial scan for dataset of 2^{19} elements. We note that since the development of Chapter 3 there have been many improvements over the ORAM scheme used. Since the same techniques we use can be applied to these newer schemes, we can infer that the construction would outperforms the linear scan for even smaller datasets.

Part II . In the second part of this theses we focused on the specific application of Private Database Search. In this setting, we aimed at protecting the client's query as well the server's data set. However, instead of looking at the problem from a theoretical perspective like in Part I, we searched for practical systems that still achieve strong security guarantees. We proposed 2 systems achieving different levels of efficiency and privacy.

The first system, called Blind Seer, was studied in Chapters 4 and 5. Blind Seer achieves extremely high efficiency while still protecting the query and data by making use of a third party assumed not to collude with server or client. In terms of functionality Blind Seer allows for:

- *Arbitrary* Boolean queries over the terms, where the only requisite is that negations are pushed down to the input level of the formula.
- Ranges queries.
- Stemming and free-text search.
- Ranking of results.
- Private access control over the queries.
- Indistinguishability of small result sets queries.

A prototype of the system was developed and tested with anonymized census data. The data set consisted of 100,000,000 records each composed by 70 searchable keywords and a payload of 100 KB. We compared the system against plaintext MySQL database system. Experimental results shows a performance multiplicative overhead of 1.2-3 over MySQL for many interesting SELECT id queries.

In addition, we formally showed that the system protects the client's query, the access control policy, and the server's data up to some limited specified leakage to the players. The privacy leakage primary correspond to access patterns to the database records and the traversed index, as well as query circuit and access control policy topology.

In Chapter 4 we showed the basic architecture and construction of Blind Seer, assuming that all players follows the protocol specification. However, we noted a weakness in the protocol that would allow a malicious client to easily circumvent the access control mechanism with no possible detection. In Chapter 5 we solved this issue by cryptographically binding the query to the access control mechanism and the traversal of the database index. In addition, our solution involve virtually no overhead by using a novel protocol that allow to securely evaluate a topology-aware universal circuit with no additional cost over evaluating the original circuit.

We presented a new prototype of Blind Seer that implemented the protection against malicious client behavior. We compared this new prototype against MySQL and the original

Blind Seer. The new Blind Seer manage to be only 2-3 times slower than MySQL when using multiple cores on both systems. Against the original Blind Seer, we showed that the system in fact introduce no significant overhead. The new implementation also allows for parallel traversal of the index, achieving up to ×15 improvement when using 15 cores.

In the next and final chapter we asked whether we can avoid the leakage to the index server and still achieve the level of efficiency reached with Blind Seer. The Blind Seer leakage mainly correspond to access pattern to the database records and index structure, as well search pattern in the query. We proposed a new scheme that stores the index and the records using ORAM structures to hide access and search patterns to the index server. Instead of secretly sharing the state of the ORAM between the players (as done in Chapter 3), we gave the client the entire ORAM parameters. However, we encrypt the ORAM data from the client in a special way and developed a novel protocol we called Masked-MOPRF, that we use to obliviously evaluate Bloom filter evaluation. Therefore, a client can traverse the index knowing *only* the result of the query in each index node, and nothing more. The new system allows for Boolean queries represented as DNF formulas over keywords. To our knowledge this is the first work that uses ORAM in the specific scenario of Boolean Private Search.

Our experimental analysis showed that the cost incurred is significant, both in terms of space required to hold the index and query search time. While Blind Seer can return a single records in a few milliseconds, the new system takes seconds on much smaller dataset. Although a straight forward comparison against the sublinear secure computation construction of Chapter 3 is not fair (since different ORAM schemes and dataset were used), we can infer that using that such a solution would be much more inefficient, since several look-ups would need to be performed to return records satisfying the Boolean formula, and the "next instruction" would require a complex circuit to compute.

We also noted that for DNF queries the bottleneck of the scheme is *not* the use of ORAM, but the oblivious Bloom filter evaluation protocol, whose complexity increases exponentially with the number of disjunctions in the query. Hence, future work may focus on a more efficient oblivious Bloom Filter evaluation procedure.

In Table 8.1 we summarize the trade-off space position of the schemes presented in this

System	Functionality	Efficiency	Leakage
Sublinear 2PC	General	Seconds per look-up	Amount of work
Blind Seer	Boolean DB Search	Practical Milliseconds per record	access pattern to index and records
Low-Leakage Secure Search	DNF DB Search	Practical for small formulas Seconds per record	work on index # records returned tree traversal to client

Table 8.1: Comparison between the protocols

Thesis. The first scheme for sublinear secure two-party computation potentially allow for any two party functionality, however every look-up perform in each evaluation runs in several seconds. The leakage is minimal and only reveals the running time of the computation. The Blind Seer system is the most efficient scheme presented here, and its also rich in functionality. Records satisfying a Boolean query can be returned within a Milliseconds. However, the leakage incurred include access pattern, search patterns, and index traversal pattern. The last scheme, allows for Boolean queries given as DNF. Is practical for small queries since records satisfying a small query can be returned in a few seconds. The leakage of such scheme is reduces only to the traversal pattern to the client, while the server learns only the amount of work performed.

8.2 Direction for future work.

Each of the system presented in this work can be improved in several ways. The use of better and simpler ORAM schemes would be of direct improvement over the solution of Chapters 3 and 6. In the same way, new secure two-party computation techniques can be used to improve efficiency in the schemes of Chapter 3 and Blind Seer. The Blind Seer system can be improved in terms of functionality by supporting more complex queries, and stronger query policies. Also, many real application make use relational multi-table database that Blind Seer can only supports naively (by adding keywords joining the tables). The last low-leakage version of Blind Seer can be enhanced by supporting access control, and multiple clients. As mentioned earlier, for large DNF formulas the ORAM running time stops being the bottle-neck of the search mechanism, and hence future work would concentrate in reducing the complexity of the oblivious Boom filter evaluation mechanism. This last scheme could potentially be modified to a two party setting (as in Chapter 3). This would require a much more expensive preprocessing step that result in the database owner holding its data in an ORAM structure for which he does not know the key. An important line of work is to develop efficient algorithms for such preprocessing, since otherwise a trusted third party is needed in any practical application.

We note that the range from complete privacy to best performance and functionality is wide, and this work only targets specific points within it. We see it as a step towards exploring several other trade-offs in this space. For example, it would be very interesting to develop an ORAM-like scheme that is specially designed for an application. A line of work in this direction is [WNL$^+$14] were the authors take advantage of a-priori known access pattern to the data structure to design a more efficient ORAM that leaks this access pattern. Another work is [GGH$^+$13] in which a single binary search is performed with one ORAM look-up.

An additional interesting line of research 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.

8.3 Publications

- The work of Chapter 3 paper has been presented at the ACM Conference of Computer Security (CCS) in 2012, and published in the corresponding proceedings [GKK$^+$12]. The paper was in joint effort with Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis.

- The first version of the Blind Seer system described in Chapter 4 is product of a 2 years of work together Vasilis Pappas, Binh Vo, Seung G. Choi, Vladimir Kolesnikov, Tal Malkin, Wesley George, Steve M. Bellovin, and Angelos Keromytis. A research paper presenting the contributions was presented at the IEEE Symposium on Security and Privacy 2014 (S&P) and published in the corresponding proceedings [PKV+14].

- The techniques of Chapter 5 that augment Blind Seer security against malicious-client behavior is product of work with Ben Fisch, Binh Vo, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steve M. Bellovin. A research paper presenting the contributions was presented at the IEEE Symposium on Security and Privacy 2015 (S&P) and published in the corresponding proceedings [FVK+15].

Part IV

Bibliography

Bibliography

[ABC$^+$08] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. *J. Cryptology*, 21(3):350–391, 2008.

[AHMR15] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 702–729, 2015.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, pages 535–552, 2007.

[BCOP04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 506–522, 2004.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, pages 97–109, 1995.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796, 2012.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM84] Manuel Blum and Silvio Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM J. Comput.*, 13(4):850–864, 1984.

[BW07] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 535–554, 2007.

[CGKO06] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security, CCS '06*, pages 79–88, New York, NY, USA, 2006.

[CGKS98] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, 1998.

[CGN97] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Technical Report TR-CS0917, Dept. of Computer Science, Technion, 1997.

[CJJ$^+$13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology -*

CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I, pages 353–373, 2013.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 577–594, 2010.

[CLT11] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. Efficient techniques for privacy-preserving sharing of sensitive information. In *Proceedings of Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011.*, pages 239–253, 2011.

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, pages 442–455, 2005.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 144–163, 2011.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[FVK$^+$15] Ben Fisch, Binh Vo, Fernando Krell, Vladimir Kolesnikov, Tal Malkin, Steven M. Bellovim, and Abishek Kumarasubramanian. Malicious-client security in blind seer: A private scalable DBMS. In *IEEE Symposium on Security and Privacy*, 2015.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GGH$^+$13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, pages 1–18, 2013.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. *Journal of the ACM*, 33(4):792–807, August 1986.

[GHJR15] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. In *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS), New York, June 2-5*, 2015.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 405–422, 2014.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 850–867, 2012.

[GIKM00] 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.

[GKK$^+$12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 576–587, 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 157–167, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[Goh03] Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications.* Cambridge University Press, 2004.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[HIKN08] Danny Harnik, Yuval Ishai, Eyal Kushilevitz, and Jesper Buus Nielsen. Ot-combiners via secure computation. In *Theory of Cryptography, Fifth Theory* *of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008.*, pages 393–411, 2008.

[HKK$^+$14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 458–475, 2014.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 145–161, 2003.

[JJK$^+$13] Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 875–888, 2013.

[JL10] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, pages 418–435, 2010.

[Kay12] Danielle M. Kays. Reasons to "friend" electronic discovery law. *Franchise Law Journal*, 32(1), 2012.

[KK12] Vladimir Kolesnikov and Ranjit Kumaresan. Improved secure two-party computation via information-theoretic garbled circuits. In *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*, pages 205–221, 2012.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156, 2012.

[KM08] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[Kol05] Vladimir Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, pages 136–155, 2005.

[KP13] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 258–274, 2013.

[KS08a] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 486–498, 2008.

[KS08b] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security, 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008, Revised Selected Papers*, pages 83–97, 2008.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung,* *Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 506–525, 2014.

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

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 1–17, 2013.

[LO13a] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 719–734, 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, pages 52–78, 2007.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 476–494, 2014.

[Mal11] Lior Malka. Vmcrypt: Modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 458–473, 2006.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[MS13] Tarik Moataz and Abdullatif Shikfa. Boolean symmetric searchable encryption. In *ASIACCS 2013: 8th ACM Symposium on Information, Computer and Communications Security*, 2013.

[Nie07] Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive*, 2007:215, 2007.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 681–700, 2012.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 448–457, 2001.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303, 1997.

[PI05] Jack E. Pace III. Testing the security blanket: An analysis of recent challenges to stipulated blanket protective orders. *Antitrust Magazine*, 19(3), 2005.

[PKV$^+$14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Steven Bellovin, and Angelos Keromytis. Blind seer: A private scalable DBMS. In *IEEE Symposium on Security and Privacy*, 2014.

[PR10a] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 502–519, 2010.

[PR10b] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 502–519, 2010.

[PRV$^+$11] Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M. Bellovin, and Tal Malkin. Private search in the real world. In *ACSAC '11*, pages 83–92, 2011.

[PRZB12] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.

[PSS09] Annika Paus, Ahmad-Reza Sadeghi, and Thomas Schneider. Practical secure evaluation of semi-private functions. In *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, pages 89–106, 2009.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 554–571, 2008.

[Rab81] Michael O. Rabin. How to exchange secrets by oblivious transfer. In *Technical Report TR-81.* Aiken Computation Laboratory, Harvard University, 1981.

[Rog91] Phillip Rogaway. *The round complexity of secure protocols*. PhD thesis, Massachusetts Institute of Technology, 1991.

[RVBM09] Mariana Raykova, Binh Vo, Steven Bellovin, and Tal Malkin. Secure anonymous database search. In *CCSW 2009.*, 2009.

[SBC$^+$07] Elaine Shi, John Bethencourt, Hubert T.-H. Chan, Dawn Xiaodong Song, and Adrian Perrig. Multi-dimensional range query over encrypted data. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 350–364, 2007.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4), October 1980.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 197–214, 2011.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

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

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian 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.

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *STOC*, pages 196–203, 1976.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226, 2014.

[WS08] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[WSC08] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 139–148, 2008.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposium on on Symbolic and Algebraic Computation*, EUROSAM '79, pages 216–226, 1979.