Private Large-Scale Databases with Distributed Searchable Symmetric Encryption

Yuval Ishai[1⋆], Eyal Kushilevitz[2⋆], Steve Lu[3⋆], and Rafail Ostrovsky[4⋆]

[1] Technion and UCLA, yuvali@cs.technion.ac.il
[2] Technion, eyalk@cs.technion.ac.il
[3] Stealth Software Technologies, Inc. steve@stealthsoftwareinc.com
[4] UCLA, rafail@cs.ucla.edu

Abstract. With the growing popularity of remote storage, the ability to outsource a large private database yet be able to search on this encrypted data is critical. Searchable symmetric encryption (SSE) is a practical method of encrypting data so that natural operations such as searching can be performed on this data. It can be viewed as an efficient private-key alternative to powerful tools such as fully homomorphic encryption, oblivious RAM, or secure multiparty computation. The main drawbacks of existing SSE schemes are the limited types of search available to them and their leakage. In this paper, we present a construction of a private outsourced database in the two-server model (e.g. two cloud services) which can be thought of as an SSE scheme on a B-tree that allows for a wide variety of search features such as range queries, substring queries, and more. Our solution can hide all leakage due to access patterns ("metadata") between queries and features a tunable parameter that provides a smooth tradeoff between privacy and efficiency. This allows us to implement a solution that supports databases which are terabytes in size and contain millions of records with only a 5× slowdown compared to MySQL when the query result size is around 10% of the database, though the fixed costs dominate smaller queries resulting in over 100× relative slowdown (under 1 second actual).

In addition, our solution also provides a mechanism for allowing data owners to set filters that prevent prohibited queries from returning any results, without revealing the filtering terms. Finally, we also present the benchmarks of our prototype implementation. This is the full version of the extended abstract to appear at CT-RSA 2016.

Keywords: Searchable Symmetric Encryption, Secure Databases, Private Cloud Computing

1 Introduction

In order to protect a large database (e.g. for cloud storage), one would like to apply encryption on the database so that only those with the proper keys can decrypt. However, for ordinary semantically secure encryption, this precludes any ability to perform useful operations on this data other than decryption. The ability to perform limited searches or other operations on ciphertexts would greatly enhance the utility of the encrypted database. This topic has motivated researchers to study the problem from many different angles, and has lead to cryptographic solutions such as Private Information Retrieval (PIR) [10,23], Oblivious RAM [17,26,27,19], Encrypted Keyword Search [4,28,14], Deterministic and Order-preserving encryption [1,3,2], Fully Homomorphic Encryption [15,5], and more.

One of the promising approaches for searching on encrypted data is known as Searchable Symmetric Encryption (SSE). This approach has been the subject of a long line of research starting with Song et al. [29]. An SSE scheme allows the data to be encrypted using only private-key primitives that allow it to be searched upon at a very low cost, while attempting to minimize the correlation between queries. The latter information is commonly referred to as *query leakage* or *access pattern leakage*. An important improvement of obtaining a sublinear time solution was introduced in Curtmola et al. [11]

[⋆] Work done while consulting for Stealth Software Technologies, Inc. Supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20199 and ENTACT subcontract through MIT Lincoln Laboratory. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation therein. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

and the notion of SSE was subsequently generalized to Structured Encryption by Chase and Kamara [9]. Recent works including that of Cash et al. [7] and Fisch et al. [13] present highly scalable SSE schemes supporting exact match queries and keyword searches, and also more complex Boolean formulas of these queries, and extended query types such as range queries.

Our motivation of building a large, scalable SSE scheme is similar to that of [7,13], but our approach and conclusions diverge from these works. Our aim is to build a light-weight solution that supports a variety of natural string-search queries. However, unlike their work, we insist on eliminating all leakage about the access pattern except an upper bound on the size of the individual matches, which must be leaked regardless of any efficiency requirements. Our solution builds on a B-tree data structure whose choice is natural as B-trees are ubiquitous, serve a variety of queries, and are more suitable for our cryptographic subprotocols compared to other string data structures like tries or n-grams.

We state a high level summary of our secure construction. At the heart of our construction is the ability for a client to privately search on a remotely held, encrypted B-tree such that 1) the client learns only the matching indices and nothing else about the entries in the tree, and 2) neither the client nor the remote parties learn which path was taken. Consider how a tree is travesed in the clear: starting from the root, a node is fetched, then the query is compared to the contents of the node which results in the pointer to a node in the next level, and this repeats until the leaf level is reached. We create cryptographic parallels to be able to perform this traversal while satisfying our security requirements.

In order to privately fetch a node from a level, PIR or even Symmetric PIR (SPIR, where the client does not learn anything beyond the query) does not fully guarantee our needs. There are two reasons for this: PIR still returns the node in the clear to the client, and the client must input a location to fetch. However, since the client should not learn which path was taken, nor the contents of the nodes, this information must be hidden. In order to account for this, we introduce a functionality known as shared-input-shared-output-SPIR or SisoSPIR that takes as input secret-shared values between the client and remote parties, and outputs a secret-shared node. This way, nodes can be fetched without the client learning the location or contents of the node. We will see later that the construction is reminiscent of the “indirect indexing” techniques due to Naor and Nissim [25]. Then, in order to compute on the secret-shared node against the query, we employ lightweight MPC that effectively computes a b-way comparison gate, where b is the branching factor of the tree, and returns a secret-shared result.

With this idea in mind, we are then able to build securely queryable B-trees, which then leads to range queries, substring queries, and more. Our paper takes a formal treatment of these concepts as a composition of cryptographic functionalities, each of which is easy to analyze, and their combined security follows from standard secure composition theorems (e.g. Canetti [6]). We propose realiziations to these functionalities, and also implement them and benchmark our results. Our code has been independently tested to scale to terabytes in size and millions of records, and we present our own timings that show that our solution is around 5× slower compared to MySQL when querying around 10% of the database, though the fixed costs dominate smaller queries resulting in over 100× relative slowdown (under 1 second actual).

1.1 Related Work

As noted above, the problem of searchable encryption, and that of private database management in general, can be solved using powerful general techniques such as Oblivious RAM, secure multiparty computation, and FHE. Our aim is to focus on practical solutions that have as little overhead as possible compared to an insecure solution. One of the interesting aspects of our construction is that we use highly efficient variants of Oblivious RAM, PIR, and MPC and apply them as sub-protocols only on *dramatically smaller portions* of the database.

There is a rich literature on searchable symmetric encryption (see for example [29,16,8,11,9,22,20,21,7,13]), and these works are highly relevant to the task at hand. Furthermore, recent works such as [24,32,12] have considered combining PIR with ORAM for efficiency reasons. While these schemes are more efficient than generic tools, they are limited in search functionality and possibly leak too much access pattern information. The most relevant work is that of Cash et al. [7], and we highlight the main differences between this work and ours. Indeed, our model uses two “servers” and a client, and the servers are assumed not to collude, as the two-server setting typically lends itself to more efficient instantiations. We also do not necessarily assume the data owner is the same as the client, which is the case for typical SSE schemes. This allows us to work in different settings, such as the example of a data owner delegating sensitive data to a semi-untrusted cloud, and still allowing a client (who is not the data owner themselves) to query against it while guaranteeing no unqueried information is leaked. If we assume that the client does own the data, then the client can play the role of both the client and the data owner, $\mathsf{S1}$, in which case non-collusion is for free (of course, this would mean the client would have to store the index data that would have been held by the primary server, but this is less data than what is held by the "helper" server that has the encrypted payloads). We obtain different string-type searches as opposed to boolean formulas on exact matches obtained by [7], and our leakage definitions are similar to those of [11,9,7] (though the type of leakage allowed by our solution is much more limited).

We do pay a price in the non-collusion assumption and efficiency compared to existing schemes, but we believe this tradeoff provides an interesting contrast since we achieve less leakage and offer an alternative construction in achieving these types of search queries like those in existing SSE schemes while maintaining a practical level of efficiency.

1.2 Our Contributions

In this work, we introduce the notion of *distributed searchable symmetric encryption.* We define it in terms of an ideal three-party functionality, where there is a querying client, a data owner, and a helper server.

We outline our main result as follows: there is a data owner $\mathsf{S1}$ that holds a database D that wants to outsource the bulk of the work of querying to a helper server $\mathsf{S2}$ such that a client C can perform queries q against D by interacting with $\mathsf{S1}$ and $\mathsf{S2}$ (but mostly $\mathsf{S2}$). The data owner wants the guarantee that only the results of the query is revealed to the C and no additional information about D, and only queries that satisfy the query policy list $\mathcal{P}$ will return any results. On the other hand, C does not want any additional information to be revealed about q to either $\mathsf{S1}$ or $\mathsf{S2}$. We can define a functionality $\mathcal{F}_{SSE}$ with two phases: Setup and Query such that during the setup phase, $\mathsf{S1}$ inputs D and $\mathcal{P}$ to $\mathcal{F}_{SSE}$, which returns a leakage profile $\mathcal{L}^i_{Setup}$ to party $i \in \mathsf{S2}, \mathsf{C}, \mathsf{S1}$. During the query phase, C inputs a query q (range, substring, etc.) to $\mathcal{F}_{SSE}$ and the functionality checks that q satisfies $\mathcal{P}$ and returns the results to C if it conforms, while sending a leakage profile $\mathcal{L}^i_{Query}$ to player $i \in \mathsf{S2}, \mathsf{C}, \mathsf{S1}$.

Main Theorem (Informal). *There is a sublinear communication protocol realizing the above SSE functionality $\mathcal{F}_{SSE}$ where the leakage profiles only reveal minimal size information (no information about access patterns and intersection of queries or results across multiple queries). The protocol achieves 1-privacy in the semi-honest (honest-but-curious) model, i.e. any adversary corrupting a single party in the protocol can be simulated in the ideal model, and uses a logarithmic number of communication rounds in the size of the database.*

In order to construct an efficient realization of this ideal functionality, we define and construct a few intermediate sub-protocols that may be of independent interest. One new concept is that of privacy preserving data structures, which can be thought of as a more general variant of Oblivious Data Structures [30]. Other concepts include efficient realizations of shared-input-shared-output variants of cryptographic primitives such as pseudorandom functions and private information retrieval.

1.3 Roadmap

In Section 2 we describe background and our model. In Section 3 we provide a high-level overview of our new scheme and provide the detailed construction and proofs for our main technical functionality $\mathsf{SisoSPIR}$ in Section 4. We construct a full-fledged distributed SSE using this functionality in Section 5. We show how to reduce various query types into range queries in Appendix C.

We describe our benchmark results in Appendix B. Finally, Appendix D discusses possible applications of our solution.

2 Background and Model

We consider a system of three parties: the client C, the server $\mathsf{S1}$, and "helper server" $\mathsf{S2}$. When considering adversarial behavior, we restrict our attention to the case of semi-honest (honest-but-curious) adversaries with the presence of an honest majority, i.e. only one party may be corrupted. Due to the low communication complexity, we automatically have some guarantees even against a malicious C. The assumption that the data owner server and the helper server are semi-honest and do not collude are reasonable if, for example, the helper server is a neutral cloud service.

We consider a simplified model of a database D, which we take to be a single table of records of the following form. D is a set of records indexed by t different fields $A_1, \ldots, A_t$, where each field A_i may take on some set of allowed values (e.g. string, date, enum, etc.). Each record $r \in D$ then takes the form $r = (x_1, \ldots, x_t, y)$ with each $x_i \in A_i$ denoting a searchable field value, and $y \in \{0,1\}^\ell$ (for some length parameter ℓ) being the payload. We make the simplifying assumptions that there is only one payload field (WLOG), the database schema is known to all parties, as well as the total number of records. All fields and records are padded up to the same length, and we assume A_1 to be a unique ID field, denoted by $id(r)$ for record r.

A *range* query q on a field A_i is of the form $x \prec b$ or $a \prec x$ or $a \prec x \prec b$, where $\prec$ can be either $<$ or $\leq$. The query returns all records r satisfying the inequality on field i. We focus on range queries and describe other query types and how to reduce them to range queries in Appendix C.

We also consider simple query authorization policies p that take as input a query q and output 0 or 1. As long as p is efficiently computable via a Boolean formula, we can use general MPC to evaluate and enforce only queries satisfying p applied to q is 1 in our system. For example, our current implementation allows us to deny queries that are not of a particular query type, or column, or value.

3 Overview of Our Construction

In this section we include a high level overview of our solution. A formal description of the various sub-protocols and their security proofs will be given in Section 4 and Section 5. In this section only, for the sake of simplicity, we focus our description on just performing a range query on a binary tree. We first consider the scenario where we do not need to hide the data owner's information from the client C. Recall that protocols such as PIR or ORAM allow queries of the form "fetch location i" from a data array D to obtain $D[i]$ to be performed in a randomized fashion without leaking any access pattern information: even identical repeated queries look the same to everyone but the querier.

First, let us focus on a single column (say, 'Name') with entries $x_1, \ldots, x_n$ (with duplicity). During initialization, these are stored in a balanced B-tree T, and let T_i denote the i-th level of the tree, and $T_i[j]$ denote the j-th node on that level. On the leaves, we additionally store pointers (along with the x_i) that point back to the original rows of the DB. In order to perform a range query (say, fetch all records where 'Name'>'Bob'), the client C uses fetches the node in root T_0 of the tree. If the value in the node is larger than 'Bob' the client wants to go right, otherwise left. This determines which node j_1 to traverse to in level T_1 of the tree. C then uses a private fetching algorithm (such as PIR or ORAM) to fetch the node $T_1[j_1]$, and then determines whether to go left or right again, which will result in j_2 for level 2 of the tree. This proceeds until C reaches a leaf, whereupon it will also privately fetch all subsequent leaves (since this is a > query). Since these leaves contain pointers $i_1, \ldots, i_k$ to the original DB, C can also privately fetch these pointers.

In our full solution, much of the complexity arises when we do not want the client C to learn the contents of the database *not* returned by the query. We therefore introduce a secret-shared variant $\mathsf{SisoSPIR}$ to ensure the location and node are secret shared, and then apply secure multiparty computation to determine whether to go left or right, where the choice is also secret-shared. We explain at a high level how this is done. Whenever C is about to receive a result of privately fetching a node, the server $\mathsf{S1}$ will mask it with a random value R_{node}. This renders the result node hidden, since now C cannot use this randomly masked value to determine whether to go left or right. Now, to determine which way to go, C invokes an MPC protocol with $\mathsf{S1}$ that computes *query$\geq$ value ? right:left.* We do not want C to know where it is exactly in the tree, so 'left' and 'right' are absolute pointers that are blinded. A common technique for this is to virtually shift the array by some random amount r, and offset the pointer by r. In order to handle policies, we incorporate a "killswitch" into the MPC where a non-compliant query will always lead the client down to a "no results found" leaf.

4 Formal Description

In this section we formally define and analyze the building blocks of our solution. All functionalities and protocols involve 3 parties: Server $\mathsf{S1}$, Client C, and Helper Server $\mathsf{S2}$.

Functionalities. We treat functionalities as picking their internal secret randomness. To model leakage, we use "leak x to P" to specify ideal functionality leakage which only affects the security requirement and not correctness, whereas "return y to P" is used to specify an actual output which affects both correctness and security. We treat the "Query phase" of functionalities as receiving a single query, with the implicit understanding that multiple queries are handled by repeating the Query phase sequentially for each query. We will sometimes invoke multiple sessions of the same protocol in parallel on different sets of inputs. Since we only consider security against semi-honest adversaries, parallel composition holds in general (we can run many simulators in parallel since the inputs cannot be modified by a semi-honest adversary to depend on the transcript). We define the main functionality we are trying to achieve, the distributed SSE functionality $\mathcal{F}_{SSE}$ in Figure 1.

Functionality $\mathcal{F}_{\mathsf{SSE}}$

Setup. $\mathsf{S1}$ inputs a database D and policy $\mathcal{P}$ to $\mathcal{F}_{SSE}$. Leak $\mathcal{L}^i_{Setup}$ (which is implementation defined) to party $i \in \mathsf{S2}, \mathsf{C}, \mathsf{S1}$.

Query. C inputs a query q. Checks that q satisfies $\mathcal{P}$ and returns the results of the query to C if it conforms. Leak a leakage profile $\mathcal{L}^i_{Query}$ to player $i \in \mathsf{S2}, \mathsf{C}, \mathsf{S1}$.

Fig. 1. The Privacy Preserving Data Structure functionality.

Protocols. To simplify the presentation of the protocols, we do not explicitly describe the authentication mechanism used for preventing attacks by the network. Security against the network is achieved via a standard use of encryption and MACs. This does not affect the security of the protocols against semi-honest insiders. We also simplify notation by letting parties pick their own randomness. We follow the standard convention of including in the *view* of each party only its internal randomness and the *incoming* messages. The outgoing messages are determined by the inputs, randomness, and incoming messages. Finally, we omit "Done" messages in the end of protocols, under the understanding that whenever a party finishes its role in a (sub)protocol, it sends a "Done" message to all other parties.

Security. We consider asymptotic (vs. concrete) security parameterized by a security parameter k. Security is defined with respect to families of polynomial-size circuits. Whenever we use a pseudorandom function (PRF) or a pseudorandom generator (PRG) we will instantiate these primitives using a standard block cipher such as AES with seed size equal to the standard key length of the block cipher. The correctness of some protocols assumes that the number of queries is smaller than 2^k. Concretely, the number of queries scheduled is polynomial in k, and the correctness requirement should hold for all sufficiently large k. We use the real/ideal simulation paradigm when discussing security of our protocols. Namely, we use the following standard definition for security (see e.g. Canetti [6] or Goldreich's Book [18]):

Definition 1. *We say a protocol π* 1-privately realizes $\mathcal{F}$ in the semi-honest model *if for every semi-honest (honest-but-curious) PPT adversary A corrupting a party in a real protocol π, there exists a PPT simulator S playing the role of A that only interacts with the ideal $\mathcal{F}$, such that on all inputs, S produces a simulated transcript that is computationally indistinguishable from the view of A. The view of A includes the transcript of messages that A sees during the execution of the protocol as well as its internal randomness.*
We say that the protocol has perfect correctness if the output of π always matches the output of $\mathcal{F}$.

4.1 Technical Overview

We provide a technical overview of our construction at a high level. The goal of our construction is to build a protocol that 1-privately realizes the functionality $\mathcal{F}_{\mathsf{SSE}}$. In order to build an efficient protocol, we look toward data structures that support fast evaluations of the queries we want (in particular, range queries). However, because the ideal functionality reveals nothing about the query except the so-called "leakage profile", we want to minimize this surface. If the data structure has vastly different number of lookups for best and worst-case queries, this would require our ideal functionality to reveal this information, otherwise no simulator could correctly guess how many lookups to simulate without knowledge of the data. Thus, as a tradeoff, we work only with privacy preserving data structures (which we introduce below) which roughly states that the access to the data structure is data independent. This is a very reasonable tradeoff as many real-world data structures already satisfy this property, in particular B-trees. After we introduce this notion, we focus just on the B-tree case, though our scheme extends to support any PPDS.

In our solution, the way a client performs a query is done roughly in two parts: first, the client interacts with S1 and S2 to traverse a B-tree to retrieve indexes matching the query, then interacts with S2 to retrieve the actual records at those indices. For the latter part, we introduce a primitive called weak distributed oblivious permutation Symmetric Private Information Retrieval or wSPIR for short, and its range-query variant rSPIR, that does the following: given a set of indices, the client can look them up from the S2 without revealing anything about the set of indices nor learning anything beyond that set of indices. This is accomplished by having the data randomly permuted and the client learning only the permuted indices.

The drawbacks of wSPIR is that once an element is looked up, it must be cached, and a more subtle point is that the indices must be known. During the traversal of the B-tree, we do not want any party to learn the path traversed by the query, and so this alone is insufficient. Therefore, we introduce another primitive, shared-input-shared-output SPIR, SisoSPIR which is a gadget that the input is a secret sharing of an index to an array (between the client and S2) and the output is a secret sharing of the indexed array element. We give two instantiations of SisoSPIR, a simple linear-time instantiation SisoLinSPIR and a more complex sublinear-time instantiation SisoSublinSPIR that we describe in Appendix A The simplicity of the linear-time instantiation makes it faster than the sublinear-time version in the implementation for most realistic database sizes, though it is slower asymptotically.

Finally, the last ingredient is a general secure multiparty computation (MPC) scheme. The way we then combine all of our ingredients is as follows. The data owner S1 sets up a PPDS B-tree to store the index data, which points to the records of the actual database, then treats each level of the B-tree as an array to be used for SisoSPIR and the main database will be set up to be used for rSPIR. When the client wants to make a query, it starts at the root where it has a trivial secret sharing with S2 and invokes SisoSPIR to obtain a secret shared version of the root node (which is different each time a query is made). It then uses general MPC to compute comparisons to obtain a secret sharing of the index to the next level of the B-tree. With this, it can then invoke SisoSPIR for the next level, and continues down until the leaf level. Then S2 sends the leaf shares to the client whereupon it can reconstruct the index information, and then uses rSPIR to retrieve the records corresponding to the query.

4.2 Privacy Preserving Data Structures (PPDS)

We can think of a (static) data structure for some data set D (consisting of $(key, value)$ pairs) as being two algorithms $\mathcal{DS} = (\mathsf{Setup}, \mathsf{Query})$. The setup algorithm takes as input some dataset D and outputs the initial state and sizes of the memory arrays $M_1, \ldots, M_k$. The query algorithm takes as input some query x and produces a sequence of memory probes of the form $q_\ell = (i, j)$ and gets the j-th entry of M_i, i.e. $M_i[j]$. The sequence can be adaptive in the sense that $q_{\ell+1}$ may depend on $q_1, \ldots, q_\ell$ as well as all the $M_i[j]$ for all $q_k = (i, j)$.

We take a modular approach and say that since PIR can hide the actual j within a memory array M_i, a PPDS need only "hide" the access pattern across the memory arrays. That is to say, there exists a simulator that can simulate the sequence of memory arrays being accessed (though it need not simulate which element in that memory array). Note that in the extreme case where each memory array is treated as a single element, the definition flattens into that of oblivious data structures as defined in [30]. We formalize this concept as a functionality $\mathcal{F}^{\mathcal{DS}}_{\mathsf{PPDS}}$, relative to some data structure $\mathcal{DS} = (\mathsf{Setup}, \mathsf{Query})$, that leaks to S2 only the sizes of the memory arrays in Figure 2.

Functionality $\mathcal{F}_{\mathsf{PPDS}}$

Setup. The functionality receives as input $(\textsc{Setup}, D)$, where D is some dataset, from the server. The functionality runs Setup on D and outputs to the server a sequence $M_1, \ldots, M_k$, where the length of the data stored in M_i is ℓ_i. It outputs $\{|M_i|\}, k, \{\ell_i\}$ to the client C and S2.

Query. The functionality receives from the client as input $(\textsc{query}, x)$, where x is a query. The functionality runs Query and returns the probe results $M_i[j]$ and locations $p_m = (i, j)$ to the client. It outputs to S2 only the is corresponding to the p_m probes.

Fig. 2. The Privacy Preserving Data Structure functionality.

Given a data structure, we define the three-party protocol $\pi^{\mathcal{DS}}$ to be: the server sets up the data structure, and the client sends its query to the server, the server processes the query and sends back the result to the client and "leaks" the memory array locations i to $\mathsf{S2}$. We say that some data structure is privacy preserving if $\pi^{\mathcal{DS}}$ is a 1-private (against a dishonest $\mathsf{S2}$) implementation of the functionality $\mathcal{F}^{\mathcal{DS}}_{\mathsf{PPDS}}$.

Observe that many data structures are well-suited for privacy-preserving data structures. Hash tables, Bloom filters, trees, and sorted arrays with binary search can all be converted to privacy-preserving ones. For the remainder of the paper, we will fix balanced B-trees as our PPDS, and focus on building a secure way to search on these B-trees.

4.3 General MPC

Some of our protocols will employ general secure multiparty computation (MPC) for simple functionalities with short inputs. In particular, the circuit complexity of functionalities we realize via general MPC will always be sublinear in the database size N. To abstract away the details of the underlying general MPC protocol we use, we will cast protocols that invoke it in the *MPC-hybrid model*. That is, we will assume the availability of a trusted oracle which receives inputs and delivers the outputs defined by the functionality. We will similarly use other hybrid models that invoke specific functionalities which we have already shown how to realize.

The implementation Π_{MPC} of an MPC oracle will use an efficient implementation of Yao's protocol [31] applied to a boolean circuit representing the functionality. To efficiently implement each 1-2 String OT in Yao's protocol, we use the 3 parties as follows: In an offline phase, $\mathsf{S1}$ generates a random OT pair (s_0, s_1) and (b, s_b), sends (s_0, s_1) to $\mathsf{S2}$ (acting as OT sender) and (b, s_b) to C (acting as OT receiver). In the online phase, we consume the precomputed random OTs via a standard, perfectly secure reduction from OT to random OT. Thus, the entire implementation of Π_{MPC} uses an arbitrary PRF as a black box, and does not require the use of public-key primitives. We omit further details about the implementation of Π_{MPC} and treat it from here on as a black box. Finally, we will use $\mathsf{sisoMPC}$ to denote a shared-input-shared-output variant of MPC, where the inputs and outputs are secret-shared between the parties (typically C and $\mathsf{S2}$).

4.4 Weak distributed oblivious permutation SPIR

We define our lowest level ideal functionality, which we refer to as Weak-distributed-oblivious-permutation-SPIR (wSPIR). This functionality, described in Figure 3, is used for allowing C to efficiently retrieve entries in an array generated by $\mathsf{S1}$. In higher level protocols which call wSPIR, the array will be randomly permuted by $\mathsf{S1}$. The main difficulty is in ensuring that $\mathsf{S2}$ does not learn the access pattern of repeated queries made by C. This is captured by leaking to $\mathsf{S2}$ a random unqueried location whenever a query repeats itself. When the database is randomly permuted, $\mathsf{S2}$ learns nothing at all about the queries made by C and C only learns the access pattern but not the actual (non-permuted) locations.

Functionality wSPIR

Init. Given an array $A \in (\{0,1\}^K)^N$ from $\mathsf{S1}$:

1. Store N, K, A.
2. Initialize an empty index list $I = \emptyset$.
3. Leak N and K to C and $\mathsf{S2}$.

Query. Given input $i \in [N]$ from C, do the following:

1. If $I = [N]$ skip to Step 4.
2. If $i \notin I$ set $i' = i$, else pick i' uniformly at random from $[N] \setminus I$.
3. Insert i' into I and leak i' to $\mathsf{S2}$.
4. Return $A[i]$ to C.

Fig. 3. Ideal functionality for Weak-distributed-oblivious-permutation-SPIR (wSPIR)

Figures 4 and 5 describe (respectively) the initialization phase and query phase of a protocol Π_{wSPIR} realizing wSPIR.

Protocol $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}$

Global parameters and functions.

- Computational security parameter 1^k.
- Pseudorandom generator (PRG) G with seed length k. The output length of G will be determined by the context.
- Pseudorandom function (PRF) $F_r : \{0,1\}^k \rightarrow \{0,1\}^k$, where $r \in \{0,1\}^k$.
- Notation: Write $F_{rsr}(x)$ to denote $F_r(F_s(F_r(x)))$ where x is padded to a length-k string.

Init.S1. On input (N, K, A), the Server $\mathsf{S1}$ does the following:

1. Pick two random and independent PRF keys $r, s \in \{0,1\}^k$.
2. Let B be the array defined by $B[i] = A[i] \oplus G(F_{rsr}(i))$.
3. Send N, K, B, s to $\mathsf{S2}$ and N, K, r to C.

Init.C. Store the values N, K, r received from $\mathsf{S1}$, and initialize an index history $IH = \emptyset$ and PRF history $PH = \emptyset$.
Init.S2. Store the values of N, K, B, s received from $\mathsf{S1}$.

Fig. 4. The initialization phase $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}$ for the functionality wSPIR

Protocol $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}$

Query.C. On input $i \in [N]$

1. If $|IH| = N$, skip to Step 4.
2. If i does not occur in IH set $i' = i$, else pick i' uniformly at random from $[N] \setminus I$.
3. Send i' to $\mathsf{S2}$ and get back $B[i']$. Insert $(i', B[i'])$ into IH.
4. Set $msg = F_r(i)$.
5. If msg does not occur in PH set $msg' = msg$, else pick msg' at random from $\{0,1\}^k$.
6. Send msg' to $\mathsf{S2}$ and get back $F_s(msg')$. Insert $(msg', F_s(msg'))$ into PH.
7. Fetch (i, b) from IH and (msg, z) from PH.
8. Return $output = b \oplus G(F_r(z))$.

Query.S2. Upon receiving i' from C, send back $B[i']$. Upon receiving msg' from C, send back $F_s(msg')$.

Fig. 5. The query phase $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}$ for the functionality wSPIR

Lemma 1. *Protocol $\mathit{\Pi}_{\mathsf{wSPIR}}$ realizes wSPIR with perfect correctness (i.e. the output of the protocol always matches the output of the functionality) and with computational security against a single semi-honest party.*

Proof Sketch. For correctness, consider an invocation of $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}$ on input i. In the end of Step 3, the pair $(i, B[i])$ must appear in IH (this is also the case when $|IH| = N$). In the end of Step 6, the pair $(msg = F_r(i), F_s(msg))$ must be in PH. Thus, after Step 7 we have $b = B[i]$ and $z = F_s(F_r(i))$, and the output computed in Step 8 satisfies $output = b \oplus G(F_r(z)) = B[i] \oplus G(F_{rsr}(i)) = A[i]$ as required. Next, we present a simulator for each party in an invocation of $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}$ and subsequent invocations of $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}$.

Simulator for S1. Since $\mathsf{S1}$ does not receive any messages its simulation is trivial.

Simulator for S2. To simulate $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init.S2}$, obtain N, K from the ideal leakage, pick B uniformly at random from $(\{0,1\}^K)^N$, and s uniformly at random from $\{0,1\}^k$ and execute $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init.S2}$ with these values of N, K, B, s. This simulation is computationally indistinguishable from the real execution of $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init.S2}$ because $\mathsf{S2}$ does not have F_r. Queries from the client invocations of $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query.C}$ are perfectly simulated by repeatedly invoking the query phase of wSPIR and observing the leakage to $\mathsf{S2}$. The m messages msg'_j received from C in Step 6 are simulated by picking uniformly random and independent strings from $\{0,1\}^k$.

Simulator for C. To simulate $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init.C}$, invoke $\mathsf{wSPIR.Init}$, obtain N, K from the leakage, pick r uniformly at random from $\{0,1\}^k$, and execute $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init.C}$ with these values of N, K, r.

Let $I = (i_1, \dots, i_m)$ be the sequence of inputs of C in these invocations, and let $O = (o_1, \dots, o_m)$ be the corresponding outputs of wSPIR. (Note that the indices i_j are not necessarily distinct.) We start by simulating the sequence of m messages $z_1, \dots, z_m$ received from S2 in Step 6. Since the messages msg'_j sent to S2 in Step 6 are all distinct (except with at most $m^2/2^k$ failure probability), and since s is unknown to C, we can simulate all these messages $z_j = F_s(msg'_j)$ by picking uniformly random and independent $\hat{z}_j \in \{0,1\}^k$. The remaining messages in Step 3 can be simulated similarly, keeping track of what C has seen. □

4.5 Shared-Input Shared-Output SPIR

A disadvantage of wSPIR is that it requires C to know the query locations i_j, and in particular learn when a query is repeated. However, when these queries are obtained by traversing a data structure (rather than originating from C), it is desirable to hide the query locations and query results from C. To this end we define and implement a stronger primitive which receives the query locations i_j in a secret-shared form and produces the output in a secret-shared form. We refer to this functionality as *shared-input shared-output SPIR* (SisoSPIR).

We present two flavors of SisoSPIR. The first has linear computational complexity in the size of the database for each query, but it is relatively lean and simple. The second implementation achieves sublinear computation via a light form of oblivious RAM. We will use the first implementation to fetch entries from the top levels of the tree-based data structure and the second to fetch entries from the bottom-most (and largest) level.

Both variants will use the following non-reactive *shared-input shared-output PRF* (SisoPRF) functionality. Loosely speaking, this functionality computes $f_r[x+y]$ and secret shares it as Q and $Q \oplus f_r[x+y]$, where r is a secret key to a PRF f and x and y are a secret sharing of an input, and Q is a random mask. See Figure 6 for a description. We will use two different implementations of this functionality: in the linear solution we will realize it via a 2-server PIR protocol applied to a precomputed table of function values, and in the sublinear solution we will implement it via the general MPC protocol π_{MPC} applied to a circuit representation of F.

Functionality SisoPRF

Global parameters and functions.

- Computational security parameter 1^k.
- Positive integers N, K received from S1 and leaked to all parties.
- Pseudorandom function $F_r : Z_N \to \{0,1\}^K$, where $r \in \{0,1\}^k$.

Functionality. On input $r \in \{0,1\}^k$ from S1, $x \in Z_N$ from C, and $y \in Z_N$ from S2:

1. Pick a random $Q \in \{0,1\}^K$.
2. Return Q to C.
3. Return $Q \oplus f_r[x+y]$ to S2.

Fig. 6. Ideal functionality for shared-input-shared-output-PRF (SisoPRF)

4.6 Linear implementation

Figure 7 defines the functionality realized by the linear implementation of SisoSPIR, referred to as SisoLinSPIR, and Figures 8 and 9 describe (respectively) the initialization phase and query phase of a protocol $\Pi_{\mathsf{SisoLinSPIR}}$ realizing SisoLinSPIR. Note that the functionality leaks the input y of S2 to S1. This leakage is harmless, because in the higher level protocols y will always be random and independent of the inputs.

Lemma 2 (Main Technical Construction of Linear SisoSPIR). *Protocol $\Pi_{\mathsf{SisoLinSPIR}}$ realizes SisoLinSPIR in the SisoPRF-hybrid model with perfect correctness and computational security against any single semi-honest party.*

Functionality SisoLinSPIR

Init. Given an array $A \in (\{0,1\}^K)^N$ from S1:

1. Store N, K, A.
   The entries of A will be indexed by the elements of the cyclic group Z_N.
2. Leak N and K to C and S2.

Query. Given input $x \in Z_N$ from C and $y \in Z_N$ from S2 do the following:

1. Leak y to S1.
2. Pick a random $R \in \{0,1\}^K$.
3. Return R to C.
4. Return $R \oplus A[x+y]$ to S2.

Fig. 7. Ideal functionality for linear shared-input-shared-output-SPIR (SisoLinSPIR)

Protocol $\Pi_{\mathsf{SisoLinSPIR}}$.Init

Global parameters and functions.

– Computational security parameter 1^k.
– Pseudorandom function $F_r : \{0,1\}^* \to \{0,1\}^*$, where $r \in \{0,1\}^k$. The input and output length will be understood from the context.

Init.S1. On input (N, K, A), the Server S1 does the following:

1. Pick a random PRF key $r \in \{0,1\}^k$.
2. Generate the masked array B defined by $B[i] = A[i] \oplus F_r(i)$ for $i \in Z_N$.
3. Send N, K, B to S2 and N, K to C.

Init.C. Store the values N, K received from S1.
Init.S2. Store the values of N, K, B received from S1.

Fig. 8. The initialization phase $\Pi_{\mathsf{SisoLinSPIR}}$.Init for the functionality SisoLinSPIR

Protocol $\Pi_{\mathsf{SisoLinSPIR}}$.Query

1. S2 sends y to S1.
2. S2 and S1 locally generate a virtual database $B^{\leftarrow y}$ defined by $B^{\leftarrow y}[i] = B[i+y]$.
3. C picks a random $R \in \{0,1\}^K$.
4. C picks a random subset $T_{\mathsf{S1}} \subseteq Z_N$ and lets $T_{\mathsf{S2}} = T_{\mathsf{S1}} \oplus \{x\}$.
5. C sends R and T_{S1} to S1 and T_{S2} to S2.
6. S1 locally computes $Z_{\mathsf{S1}} = \bigoplus_{i \in T_{\mathsf{S1}}} B^{\leftarrow y}[i]$ and S2 computes $Z_{\mathsf{S2}} = \bigoplus_{i \in T_{\mathsf{S2}}} B^{\leftarrow y}[i]$.
7. S1 sends to S2 the string $Z'_{\mathsf{S1}} = Z_{\mathsf{S1}} \oplus R$.
8. Parties invoke the SisoPRF oracle with inputs (N, K, r) from S1, input x from C, and input y from S2. Let Y_{C} and Y_{S2} denote the outputs.
9. C outputs $R \oplus Y_{\mathsf{C}}$ and S2 outputs $Z_{\mathsf{S2}} \oplus Z'_{\mathsf{S1}} \oplus Y_{\mathsf{S2}}$.

Fig. 9. The query phase $\Pi_{\mathsf{SisoLinSPIR}}$.Query for the functionality SisoLinSPIR in the SisoPRF-hybrid model

Proof. Recall from Figure 9 the definition of $T_{\mathsf{S1}}, T_{\mathsf{S2}}$ and $Z_{\mathsf{S1}}, Z_{\mathsf{S2}}$. Since the only difference between T_{S1} and T_{S2} is that x is a member of one and not of the other, we have $Z_{\mathsf{S1}} \oplus Z_{\mathsf{S2}} = B^{\leftarrow y}[x] = B[x+y]$. By the definition of SisoPRF we have $Y_{\mathsf{C}} \oplus Y_{\mathsf{S2}} = F_r(x+y)$. Hence $Y_{\mathsf{C}} \oplus (Z_{\mathsf{S2}} \oplus Z_{\mathsf{S1}} \oplus Y_{\mathsf{S2}}) = B[x+y] \oplus F_r(x+y) = A[x+y]$ and the pair of outputs $(R \oplus Y_{\mathsf{C}}, Z_{\mathsf{S2}} \oplus Z_{\mathsf{S1}} \oplus R \oplus Y_{\mathsf{S2}})$ is distributed identically to $(R, A[x+y] \oplus R)$, as required by the functionality.
We describe a simulator for each of the 3 parties.

Simulator for S1. To simulate the view of $\mathsf{S1}$, pick a random PRF key $r \in \{0,1\}^k$ (randomness of $\mathsf{S1}$ in $\Pi_{\mathsf{SisoLinSPIR}}.\mathsf{Init}$), a random mask $R \in \{0,1\}^K$, and a random subset $T_{\mathsf{S1}} \subseteq Z_N$ (messages from C in Step 5 of $\Pi_{\mathsf{SisoLinSPIR}}.\mathsf{Query}$). Together with the leaked y, this perfectly simulates the view of $\mathsf{S1}$, even when considered jointly with the outputs of C and $\mathsf{S2}$. This follows from the fact that the output of C in the protocol is masked with a random Q (from $\mathsf{SisoPRF}$) which is picked independently of the view of $\mathsf{S1}$. Thus, conditioned on the view of $\mathsf{S1}$ in the real protocol the output of C is uniformly distributed, as in the ideal world. Since the output of $\mathsf{S2}$ is *determined* by the inputs and the output of C, the simulation is perfect.

Simulator for C. The simulator is given leakage N, K and the output O_{C}. It simulates the randomness of C by picking $R \in \{0,1\}^K$ and $T_{\mathsf{S1}} \subseteq Z_N$ at random. It simulates the message received from $\mathsf{SisoPRF}$ by letting $Y_{\mathsf{C}} = R \oplus O_{\mathsf{C}}$. Since R is independent of the randomness Q used by $\mathsf{SisoPRF}$, the value of O_{C} in the real protocol is independent of R. Thus, the simulated view conditioned on O_{C} is distributed as in the real protocol. Finally, since the output of $\mathsf{S2}$ is determined by the inputs and O_{C}, the simulation is perfect also when considered jointly with the output of $\mathsf{S2}$ (recall that $\mathsf{S1}$ has no output).

Simulator for S2. The simulator is given leakage N, K and the output O_{S2}. It simulates the message B from $\mathsf{S1}$ by picking a random $B \in (\{0,1\}^K)^N$. It simulates the message T_{S2} by picking a random subset of Z_N and the message Z'_S by picking a random string from $\{0,1\}^K$. Finally, it simulates the message from $\mathsf{SisoPRF}$ by $Y_{\mathsf{S2}} = O_{\mathsf{S2}} \oplus Z_{\mathsf{S2}} \oplus Z'_{\mathsf{S1}}$ where Z_{S2} is computed from the simulated B and T_{S2} as in the protocol. To argue the correctness of the simulator, consider a hybrid experiment in which the protocol is executed with a truly random function F_r. This experiment is computationally indistinguishable from the real experiment by the pseudorandomness of F. In this case, the values $B, T_{\mathsf{S2}}, Z'_{\mathsf{S1}}, Y_{\mathsf{S2}}$ in the real protocol are uniformly and independently distributed, and $O_{\mathsf{S2}} = Z_{\mathsf{S2}} \oplus Z'_{\mathsf{S1}} \oplus Y_{\mathsf{S2}}$. Thus, the joint distribution of the view of $\mathsf{S2}$ and its output in the hybrid experiment is identical to the joint distribution of the simulated view and the output of $\mathsf{S2}$ in the ideal model. Since the output of C is determined by O_{S2} and the inputs, the simulator emulates the real protocol even when considering the output of C. □

5 Full SSE and Range Queries

5.1 Weak distributed oblivious permutation Range SPIR

Figure 10 defines our next (low-level) ideal functionality, referred to as Weak-Distributed-oblivious-permutation-Range-SPIR functionality (rSPIR). This functionality allows C to retrieve a range of entries from a permuted array generated by S1. Namely, C will obtain the entries $\alpha_0, \alpha_0+1, \ldots, \alpha_n$ of the array, corresponding to two permuted endpoints: $i = \pi(\alpha_0)$ and $j = \pi(\alpha_n)$. This is done without revealing the permutation π to C or S2. Moreover, S2 should be unable to observe overlaps between the retrieved ranges across queries. Such an overlap should look for him as accessing a random fresh location in the array which, when the database is randomly permuted, ensures that S2 learns nothing about the access pattern of C.
Figures 11 and 12 describe (respectively) the initialization phase and query phase of a protocol Π_{rSPIR} which realizes rSPIR with security against a single semi-honest party.

Lemma 3. *Protocol Π_{rSPIR} realizes rSPIR with perfect correctness and computational security against a single semi-honest party.*

Proof Sketch. The proof proceeds similarly to the wSPIR proof.

5.2 FindEndpoints

Our goal will be to use the above protocols to retrieve a range of records, once we found the relevant endpoints. For this we use, for each searchable field and each type of query, a "helper" array which is sorted according to the field value (or, sometimes, tokens) and contain pointers to the actual records.

Example. Suppose we have 2 records: (John, Smith, Blue, "See Alice Run") and (Bob, Jones, Green, "Hello World"). Then, for example, the helper table for the first column and range queries will look like ((Bob,1),(John,0)) (i.e., first names are sorted and each comes with the corresponding record number). Similarly, we have helper arrays for the second and third columns. For the fourth column, the helper table will look like ((Hello World,1),(See Alice Run,0)), for *range queries*, and ((Alice,0),(Hello,1),(Run,0),(See,0), (World,1)), for *keyword search queries* (where the values are tokenized into keywords).

Functionality rSPIR

```
Init. Given an array $A \in (\{0,1\}^K)^N$ and a permutation $\pi : [N] \to [N]$ from S1:

 1. Store $N, K, A, \pi$.
 2. Initialize an empty index list $I = \emptyset$.
 3. Leak $N$ and $K$ to all parties.

Query. Given input $i, j \in [N]$ from C, such that $\pi^{-1}(i) \le \pi^{-1}(j)$, do the following:

 1. Set $\alpha_{\mathsf{first}} = \pi^{-1}(i), \alpha_{\mathsf{last}} = \pi^{-1}(j)$ and $\Delta = \alpha_{\mathsf{last}} - \alpha_{\mathsf{first}}$.
 2. For $t = 0, \ldots, \Delta$ do:
          Let $\alpha_t = \alpha_{\mathsf{first}} + t$ and $\beta_t = \pi(\alpha_t)$.
          If $I = [N]$ skip to Step 3.
          If $\beta_t \notin I$ set $\beta'_t = \beta_t$, else pick $\beta'_t$ uniformly at random from $[N] \setminus I$.
          Insert $\beta'_t$ into $I$, leak $\beta'_t$ to S2 and $\beta_t$ to C.
    EndFor
 3. Return $A[\alpha_0], \ldots, A[\alpha_\Delta]$ to C.
```

Fig. 10. Ideal functionality for Weak-Distributed-oblivious-permutation-Range-SPIR (rSPIR)

Protocol $\mathit{\Pi}_{\mathsf{rSPIR}}.\mathsf{Init}$

```
Global parameters and functions.

 – Computational security parameter $1^k$.
 – Pseudorandom function (PRF) $F_r : \{0,1\}^k \to \{0,1\}^k$.

Init.S1. On input $(N, K, A, \pi)$, the Server S1 does the following:

 1. Pick two random and independent PRF keys $r, s \in \{0,1\}^k$.
 2. Let $B$ be the array defined by $B[i] = (A[i], \pi(i+1) \oplus F_{rsr}(\pi(i)))$ and $B' = \pi(B)$.
 3. Invoke $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}(N, K, B')$.
 4. Send $s$ to S2 and $r$ to C.

Init.S2. Participate in $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}$. Store the value $s$ received from S1.
Init.C. Participate in $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Init}$. Store the value $s$ received from S1, and initialize a PRF history $PH = \emptyset$.
```

Fig. 11. The initialization phase $\mathit{\Pi}_{\mathsf{rSPIR}}.\mathsf{Init}$ for the functionality rSPIR

Figure 13 defines the ideal functionality FindEndpoints. This functionality allows C to find, given a query on field field of type type, the two endpoints of the range of matches inside an array $L^{\mathsf{field},\mathsf{type}}$. The exact content of these arrays will be defined shortly.

The implementation of the FindEndpoints functionality is based on B-tree data-structures. S1, given each of the sorted arrays $L = L^{\mathsf{field},\mathsf{type}}$ (for each supported pair (field,type)), builds a B-tree with branching factor b as follows: in the leaf layer, partition the elements of L into groups of b elements each (in order). That is, for each such group, there is a leaf node (i.e., the i-th element of the array L belongs to the $\lfloor i/b \rfloor$) leaf node). We will also need to append the value i to the i-th element in the leafs. Finally, we create a leftmost "trap" node that contains $-\infty$ and a rightmost "trap" node which contains $+\infty$ values. Non-leaf nodes will contain b elements of the form $(i_{\mathrm{l-value}}, i_{\mathrm{r-value}}, ptr)$, where the value of all elements inside the subtree pointed to by ptr is in the (closed) interval $[i_{\mathrm{l-value}}, i_{\mathrm{r-value}}]$. Again, each of these internal layers (excluding the root layer) will contain a leftmost and rightmost "trap" nodes. Finally, in the initialization we invoke SisoLinSPIR.Init for each such layer (which results in S2 having an "encrypted" form of the layer).

The leaf level needs to be treated with a bit more care. This is because, when FindEndpoints.Query is applied, its final output is an actual pointer into the array. Moreover, to allow for an extra efficiency in the leaf level, we add to the standard entries of the array some δ dummy entries (those are not actually pointed to in our B-tree data-structure but are used, by SisoSublinSPIR, to hide access patterns). That is, we will take our array L add to it δ dummy arguments and permute it using some random

Protocol $\mathit{\Pi}_{\mathsf{rSPIR}}.\mathsf{Query}$

Query.C. On input $i, j \in [N]$

```
1. $ptr = i, t = 0$
2. While $ptr \neq j$
       Invoke $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}(ptr)$ to obtain $B'[ptr] = B[\pi^{-1}(ptr)] = (A[\alpha_t], \gamma_{t+1})$.
       Set $msg = F_r(ptr)$.
       If $\exists(msg, z) \in PH$ pick random $msg' \in_R \{0,1\}^k$. Else, set $msg' = msg$.
       Send $msg'$ to S2 and get back $F_s(msg')$. Insert $(msg', F_s(msg'))$ into $PH$.
       Fetch $(msg, z)$ from $PH$.
       Compute $ptr = \gamma_{t+1} \oplus F_r(z)$.
       $t \leftarrow t + 1$
   End While
3. Invoke $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}(j)$. Obtain $(A[\alpha_t], \gamma_{t+1})$
4. Output $A[\alpha_0], ..., A[\alpha_t]$
```

Query.S2. Participate in $\mathit{\Pi}_{\mathsf{wSPIR}}.\mathsf{Query}$. Upon receiving msg' from C, send back $F_s(msg')$.

Fig. 12. The query phase $\mathit{\Pi}_{\mathsf{rSPIR}}.\mathsf{Query}$ for the functionality rSPIR

Functionality FindEndpoints

Global parameters. List of pairs (field, type) such that S1 supports queries of type type to field field.
Init. For each pair (field, type) in the list, S1 provides an array $L^{\mathsf{field,type}} \in (\{0,1\}^K)^N$.

1. Store each array $L^{\mathsf{field,type}}$.
2. Leak the corresponding N and K to all parties.

Query. Given query q from C, do the following:

1. Let i_{left} be the minimal element of $L^{\mathsf{field,type}}$ that matches q, or $i_{\text{left}} = +\infty$ if no such element exists.
2. Let i_{right} be the maximal element of $L^{\mathsf{field,type}}$ that matches q, or $i_{\text{right}} = -\infty$ if no such element exists.
3. Return $i_{\text{left}}, i_{\text{right}}$ to C.

Fig. 13. Ideal functionality for FindEndpoints

permutation $\tau : [N + \delta] \to [N + \delta]$ (which in particular hides the information on which of the entries is a dummy entry and which is a real entry). Moreover, we will update according to τ all the pointers into the array, to reflect the new random order (and the increased size). We will apply SisoSublinSPIR.Init to this array, with $\mathit{\Gamma} = \{\tau(N), \ldots, \tau(N + \delta - 1)\}$.
Figure 14 describes the initialization phase of protocol $\mathit{\Pi}_{\mathsf{FindEndpoints}}$ which realizes FindEndpoints with security against a single semi-honest party.

Protocol $\mathit{\Pi}_{\mathsf{FindEndpoints}}.\mathsf{Init}$

Global parameters and functions. List of pairs (field, type) such that S1 supports queries of type type to field field.
Init.S1. For each pair (field, type) in the list, S1 is given an input array $L^{\mathsf{field,type}} \in (\{0,1\}^K)^{N^{\mathsf{field,type}}}$. It does the following:

1. Construct a B-tree $T^{\mathsf{field,type}}$, as described above, for the (field, type) pair.
2. Send $N^{\mathsf{field,type}}, K, T^{\mathsf{field,type}}$ to S2 and $N^{\mathsf{field,type}}, K$ to C.

Init.S2. Participate in SisoSublinSPIR.Init. Store the values $N^{\mathsf{field,type}}, K, T^{\mathsf{field,type}}$ received from S1.
Init.C. Participate in SisoSublinSPIR.Init. Store the values $N^{\mathsf{field,type}}, K$ received from S1.

Fig. 14. The initialization phase $\mathit{\Pi}_{\mathsf{FindEndpoints}}.\mathsf{Init}$ for the functionality FindEndpoints

Protocol $\mathit{\Pi}_{\mathsf{FindEndpoints}}$.Query

Global parameters and functions. MPC Protocols for the following functionalities:

- The functionality find-left gets as input (additive) shares for the content of the current node v in the B-tree (C-node and S2-node), a query q from C and a pointer left-trap for a trap node. It returns shares of a pointer (C-ptr, S2-ptr) to the leftmost (direct) child of v that satisfies the query q, or to left-trap if no such child exists. The functionality find-right is defined similarly for finding the rightmost child that satisfies q.
- The functionality ExtractEndpoints gets as input shares C-l-leaf, S2-l-leaf of the leftmost node satisfying q and C-r-leaf, S2-r-leaf of the rightmost node satisfying q. The first node is of the form $(x, i_{\mathrm{left}}, i_{\mathrm{realleft}})$ and the second is $(x', i_{\mathrm{right}}, i_{\mathrm{realright}})$. The functionality returns $i_{\mathrm{left}}, i_{\mathrm{right}}$ to C, except if $i_{\mathrm{realleft}} = +\infty$ or $i_{\mathrm{realright}} = -\infty$ or $i_{\mathrm{realleft}} > i_{\mathrm{realright}}$ or q does not satisfy the policy; in all of these cases return $i_{\mathrm{left}} = +\infty, i_{\mathrm{right}} = -\infty$.

Query. On input $q \in \{0,1\}^K$ for C:

```
1. $depth = \lceil \log_b N \rceil$, C-ptr $= root$, S2-ptr $= 0$
2. Do $depth$ times:
        Invoke $\Pi_{\mathsf{SisoLinSPIR}}$(C-ptr, S2-ptr) to obtain C-node, S2-node.
        Invoke MPC oracle for find-left(C-node, S2-node, $q$, left-trap) to obtain C-ptr, S2-ptr.
   End Do
3. Invoke $\Pi_{\mathsf{SisoSublinSPIR}}$(C-ptr, S2-ptr) to obtain shares of left leaf C-l-leaf, S2-l-leaf.
4. C-ptr $= root$, S2-ptr $= 0$
5. Do $depth$ times:
        Invoke $\Pi_{\mathsf{SisoLinSPIR}}$(C-ptr, S2-ptr) to obtain C-node, S2-node.
        Invoke MPC oracle for find-right(C-node, S2-node, $q$, right-trap). Obtain C-ptr, S2-ptr.
   End Do
6. Invoke $\Pi_{\mathsf{SisoSublinSPIR}}$(C-ptr, S2-ptr) to obtain shares of right leaf C-r-leaf, S2-r-leaf.

7. Invoke MPC oracle for ExtractEndpoints(C-l-leaf, S2-l-leaf, C-r-leaf, S2-r-leaf).
```

Fig. 15. The query phase $\mathit{\Pi}_{\mathsf{FindEndpoints}}$.Query for the functionality FindEndpoints in the (SisoPRF,MPC)-hybrid model

Next, we describe how FindEndpoints.Query is actually implemented. The idea is to traverse the B-tree searching for the given value in the tree. This starts from the root, and at each step proceeds from the current node to one of its b children, maintaining the invariant that the value that we search for is always between $i_{\mathrm{l-value}}$ and $i_{\mathrm{r-value}}$ of the current node (this includes the possibility of these values being $-\infty$ or $+\infty$, in case that we search for a value that is smaller or bigger, respectively, than all values in L). This search is performed jointly between S2 who knows the B-tree, and the client C who knows the query q. At each stage, they hold shares for the pointer to the current node (C-ptr, S2-ptr, respectively) as well as shares for the content of that node (C-node, S2-node, respectively). They compute a shared value of the next pointer via a MPC protocol that allows them maintaining all their information secret, followed by a shared-input shared-output SPIR that allows them to obtain the shares of the content. Recall that we have several versions of SisoSPIR; we employ the linear version for the internal nodes of the B-tree and the sub-linear version for the leafs level which is, of course, the largest. The latter is more efficient but its security limited to δ invocations; see discussion earlier (also, the shares for SisoSublinSPIR are in $Z_{N+\delta}$ rather than just Z_N in the standard SisoSPIR, where N stands for the size of the relevant array). Finally, the leaf nodes contain triplets of the form $(x_i, \pi^{\mathsf{field,type}}(i), i)$, from which the endpoints will be extracted given that some basic requirements are met (e.g., that q satisfies the policy). Figure 15 describes the query phase of protocol $\mathit{\Pi}_{\mathsf{FindEndpoints}}$.

Next, we consider the following outer protocol OuterFindEndpoints, that serves as an interface to FindEndpoints by applying a few additional permutations. Its setup phase consists of preparing all the arrays that FindEndpoints will utilize. It turns out that rather than keeping one array (for each field and query type), with entries of the form (x_i, ptr_i), it is convenient to keep two arrays $L[i] = (x_i, i)$ and $B[i] = ptr_i$ (all sorted according to x_i). Then, those arrays will be appropriately mixed, by applying to them some random permutations, in order to hide unnecessary information. Finally, in the query phase OuterFindEndpoints again invokes FindEndpoints, appropriately dealing with the random permutations. The ideal functionality OuterFindEndpoints is described in Figure 16. The implementation is straightforward as it is essentially obtained by replacing the various ideal functionalities by their actual implementations.

Functionality OuterFindEndpoints

Init. S1 is given database $D \in (\{0,1\}^K)^N$.
It picks a random permutation $\sigma : [N] \to [N]$. Then, for each field field and query type type that S1 supports, it does the following:

1. Compute arrays $L^{\mathsf{field,type}}[i] = (x_i, i)$, $B^{\mathsf{field,type}}[i] = ptr_i$ (of length $N^{\mathsf{field,type}}$).
2. Pick a random permutation $\pi^{\mathsf{field,type}} : [N^{\mathsf{field,type}}] \to [N^{\mathsf{field,type}}]$.
3. Let $L'^{\ \mathsf{field,type}}[i] = (x_i, \pi^{\mathsf{field,type}}(i))$ and $B'^{\ \mathsf{field,type}}[i] = \sigma(ptr_i)$.
4. Invoke rSPIR.Init using $B'^{\mathsf{field,type}}$ and $\pi^{\mathsf{field,type}}$ as inputs.
5. Invoke FindEndpoints.Init using $L'^{\mathsf{field,type}}$ as input.

Invoke wSPIR.Init using $D' = \sigma(D)$ (a randomly permuted version of database D) as input.
Query. Given query q of type type to field field from C, do the following:

1. Invoke FindEndpoints.Query using array $L'^{\mathsf{field,type}}$ and query q.
   Obtain i_{left} and i_{right} which are equal $\pi^{\mathsf{field,type}}(i_{\text{realleft}})$ and $\pi^{\mathsf{field,type}}(i_{\text{realright}})$, respectively. If $i_{\text{left}} = +\infty$ then Return $\emptyset$ to C.
2. Invoke rSPIR.Query using i_{left} and i_{right} as inputs.
   Obtain all elements in $B'[(\pi^{\mathsf{field,type}})^{-1}(i_{\text{left}}), \ldots, (\pi^{\mathsf{field,type}})^{-1}(i_{\text{right}})]$ (that is, $B'[i_{\text{realleft}}, \ldots, i_{\text{realright}}]$).
3. Each of these values is of the form $\sigma(ptr_j)$, for some ptr_j which is a pointer to a record that actually matches the query q. C invokes the functionality Π_{wSPIR}.Query on each value $\sigma(ptr_j)$ to obtain the records $D'(\sigma(ptr_j)) = D[\sigma^{-1}(\sigma(ptr_j))] = D[ptr_j]$.
   Return records to C.

Fig. 16. Ideal functionality for OuterFindEndpoints

Remark. We discuss how to handle query policies: we augment the FindEndpoints functionality to take as input a policy from the server, and if it is not satisfied by the policy, it sets $i_{\text{left}} = +\infty$ and $i_{\text{right}} = -\infty$.

5.3 Putting it All Together

Theorem 1 (Main Theorem.). *The OuterFindEndpoints protocol is a sublinear communication protocol realizing the distributed SSE functionality $\mathcal{F}_{SSE}$ where the leakage profiles only reveal the sizes of the objects (no information about access patterns and intersection of queries or results across multiple queries). The protocol achieves 1-privacy in the semi-honest model and uses a logarithmic number of communication rounds in the size of the database.*

Proof. We describe how to construct a simulator for all the parties in the protocol implementing OuterFindEndpoints.

Simulator for S1. For the initialization phase, the simulator mimics the server S1 proceeds as it does in the protocol, except it uses the simulator for rSPIR.Init and FindEndpoints.Init for all fields and query types, and finally wSPIR.Init for the main database. This generates the transcript for the initialization step.
During a query, the simulator generates the view for S1 by running the simulator for FindEndpoints.Query followed by the simulator for rSPIR.Query. Finally, it invokes the wSPIR.Query simulator (which is trivially empty for S1).

Simulator for S2. For the initialization phase, the simulator for S2 requires invoking the simulator for rSPIR.Init and FindEndpoints.Init for all the fields and query types. Finally, it runs the simulator for wSPIR.Init for the main database.
During a query, the simulator generates the view for S2 by running the simulator for FindEndpoints.Query followed by the simulator for rSPIR.Query. Finally, it invokes the wSPIR.Query simulator.

Simulator for C. For the initialization phase, simulating the client C is similar to simulating S2: we run the simulator for rSPIR.Init and FindEndpoints.Init for all the fields and query types. Finally, it runs the simulator for wSPIR.Init for the main database.
During a query, the simulator generates the view for C by running the simulator for FindEndpoints.Query followed by the simulator for rSPIR.Query. Since it knows the result of the query, it can generate the view for the client for i_{left} and i_{right} as follows. If no results are returned, i_{left} and i_{right} are set to their

trap representations, otherwise they are set to the range to be returned (as the result of the query). The simulator also maintains a list of previously seen pointers and chooses a random pointer for each never-before-seen location, or reusing an old one previously seen. Finally, it invokes the wSPIR.Query simulator which just returns the result to C. □

6 Conclusion

In this paper, we presented a solution for large-scale private database outsourcing via an SSE-style construction on B-trees. We formalized a model for our two-server SSE, and provided an abstract scheme along with an efficient realization of the scheme as our solution. The solution has sublinear overhead and leaks no access pattern information up to δ queries. Finally, we implemented a prototype and provided benchmarked results for our solution, which is only 5× slower compared to MySQL when querying around 10% of the database, with smaller queries resulting in over 100× relative slowdown due to fixed costs.

References

1. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *SIGMOD Conference*, pages 563–574, 2004.
2. Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.
3. Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. *IACR Cryptology ePrint Archive*, 2012:624, 2012.
4. Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.
5. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, pages 97–106, 2011.
6. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
7. David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO (1)*, pages 353–373, 2013.
8. Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455, 2005.
9. Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, pages 577–594, 2010.
10. Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.
11. Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS*, pages 79–88, 2006.
12. Jonathan Dautrich and Chinya Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *ACM CODASPY*, pages 289–296, 2015.
13. Ben Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in blind seer: A scalable private dbms. *IACR Cryptology ePrint Archive*, 2014:963, 2014.
14. Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324, 2005.
15. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
16. Eu-Jin Goh. Secure indexes, 2003.
17. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.
18. Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, Cambridge, UK, 2001.
19. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
20. Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *ACM CCS*, pages 875–888, 2013.

21. Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography*, pages 258–274, 2013.
22. Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography*, pages 285–298, 2012.
23. Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.
24. Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*, 2014.
25. Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.
26. Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.
27. Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology, 1992., Dept. of Electrical Engineering and Computer Science, June 1992.
28. Rafail Ostrovsky and William E. Skeith III. Private searching on streaming data. In *CRYPTO*.
29. Dawn Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
30. Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM CCS*, pages 215–226, 2014.
31. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.
32. Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. KT-ORAM: A bandwidth-efficient ORAM built on k-ary tree of PIR nodes. *IACR Cryptology ePrint Archive*, 2014:624, 2014.

A Sublinear Realization of SisoSPIR

The sublinear implementation of SisoSPIR will apply light oblivious RAM machinery for making the computation sublinear without revealing the access pattern to either C or S2. Specifically, the implementation is similar to that of wSPIR except that the inputs and outputs of the PRFs need to be secret-shared between C and S2 (as captured by the SisoPRF functionality) and the history lists also need to be secret-shared between C and S2. Since realizing SisoPRF is expensive and may form an efficiency bottleneck, we would like to minimize the number of calls to this functionality. To this end we add dummy entries to the database whose (pseudorandom) locations are known to C. In each subsequent invocation, a new dummy entry is consumed in order to conceal the access pattern from S2. We use a parameter δ, set by S1, to determine the number of dummy entries. After δ invocations of the protocol, when all the dummy entries are consumed, one can re-initialize the database with new dummy entries, yielding a solution with amortized $\tilde{O}(\sqrt{N})$ complexity. Alternatively, one can start revealing the access pattern to S2 once the dummy entries are consumed. In the current implementation we opt for the latter option.

Figure 17 defines the functionality realized by the sublinear implementation of SisoSPIR, referred to as SisoSublinSPIR, and Figures 18 and 19 describe (respectively) the initialization phase and query phase of a protocol $\Pi_{\mathsf{SisoSublinSPIR}}$ realizing SisoSublinSPIR. Note that the functionality leaks the positions of the dummy entries to C and the position of the queried entry to S2, where repeated entries are replaced by dummy entries. This leakage will be harmless, because the higher level protocol will randomly permute the concatenation of the actual database A with the δ dummy entries. We also note that in the actual usage of SisoSublinSPIR, the shares x and y of the queried position will never point to a dummy entry. We therefore obtain the following lemma.

Lemma 4. *Protocol $\Pi_{\mathsf{SisoSublinSPIR}}$ realizes SisoSublinSPIR in the (SisoPRF, MPC)-hybrid model with perfect correctness and computational security against any single semi-honest party.*

Proof. In the first δ queries, Step 4 creates secret sharing between C and S2 of a flag *new* which is set to 1 if and only if this is the first occurrence of the index $i = x + y$. (Note that, given the promise that $x + y \notin \Gamma$, the flag *new* is always set to 1 upon the first occurrence of i.) Step 4 also creates a sharing of the fetched string $b = B[i]$ in case i has occurred before. Step 6 uses the flag *new* to select between the fetched b and the value $b' = B[i']$ known to S2, yielding a sharing of $\beta = B[i]$. Finally, β is unmasked by $F_r[i]$ in Step 8 to yield output shares of $A'[x + y]$, as required.

After more than δ queries, Step 1 guarantees that the value β shared between C and S2 is equal to the masked database entry $B[x + y]$, which in Step 8 is unmasked with $F_r[x + y]$ to yield output shares of $A'[x + y]$, as required.

Due to the similarity to previous protocols, we briefly argue security against each party.

Functionality SisoSublinSPIR

Init. Given positive integers N, K, δ, an array $A' \in (\{0,1\}^K)^{N+\delta}$, and a list of δ distinct dummy positions $\Gamma = (d_1, d_2, \ldots, d_\delta) \in Z_{N+\delta}^{\delta}$ from S1:

1. Store N, K, A', Γ, δ.
   The entries of A' will be indexed by the elements of the cyclic group $Z_{N+\delta}$.
2. Initialize a counter q to 0 and an index history list I to $\emptyset$.
3. Leak N, K, Γ to C and N, K, δ to S2.

Query. Given input $x \in Z_{N+\delta}$ from C and $y \in Z_{N+\delta}$ from S2, with the promise that $x + y \notin \Gamma$, do the following:

1. Let $i = x + y$.
2. If $q \geq \delta$, leak i to S2 and skip to Step 5; otherwise increment q.
3. If $i \notin I$ let $i' = i$, else let $i' = d_q$.
4. Insert i into I and leak i' to S2.
5. Pick a random $R \in \{0,1\}^K$, and return R to C, and return $R \oplus A'[i]$ to S2.

Fig. 17. Ideal functionality for sublinear shared-input-shared-output-SPIR (SisoSublinSPIR)

Protocol $\Pi_{\mathsf{SisoSublinSPIR}}$.Init

Global parameters and functions.

- Computational security parameter 1^k.
- Pseudorandom function $F_r : \{0,1\}^* \to \{0,1\}^*$, where $r \in \{0,1\}^k$. The input and output length will be understood from the context.

Init.S1. On input $(N, K, \delta, A', \Gamma)$, the Server S1 does the following:

1. Pick a random PRF key $r \in \{0,1\}^k$.
2. Generate the masked array B defined by $B[i] = A'[i] \oplus F_r(i)$ for $i \in Z_{N+\delta}$.
3. Send N, K, δ, B to S2 and N, K, δ, Γ to C.

Init.C. Store the values N, K, δ, Γ received from S1. Initialize a counter q to 0.
Init.S2. Store the values of N, K, δ, B received from S1. Initialize list IH to $\emptyset$.

Fig. 18. The initialization phase $\Pi_{\mathsf{SisoSublinSPIR}}$.Init for the functionality SisoSublinSPIR

Simulator for S1. Since S1 does not get any messages (in the hybrid model), simulating S1 is trivial.

Simulator for C. Other than learning N, K, Γ (which are allowed by the leakage), C learns nothing: the message it receives from the SisoPRF oracle in Step 7 is uniformly distributed, independently of its input and randomness.

Simulator for S2. In the initialization phase, S2 gets a PRF-masked copy of the database A'. Given the leakage N, K, δ this can be simulated (up to computational indistinguishability) by picking a random database. In the q-th invocation of the query phase, $1 \leq q \leq \delta$, S2 learns in Step 4 an index i' which either satisfies $i' = x + y$ if $x + y$ occurs for the first time as the sum of the inputs, or $i = d_q$ otherwise. This is precisely the leakage of the ideal functionality. Starting from invocation δ, Step 1 reveals to S2 the input x of C, which is also allowed by the leakage of the ideal functionality. All other messages received by S2 in Steps 4,6,7 are random. □

B Implementation and Benchmarking

We implemented our protocol in C and C++ targeting a POSIX environment. In our implementation, we transmit all information over TLS, thus reducing the leakage to the network to just the *size* of

```
Protocol $\Pi_{\mathsf{SisoSublinSPIR}}.\mathsf{Query}$

1. If $q \geq \delta$:
   – $\mathsf{C}$ lets $\beta_{\mathsf{C}} = 0^K$ and sends $(\mathsf{BufferFull}, x)$ to $\mathsf{S2}$.
   – $\mathsf{S2}$ lets $\beta_{\mathsf{S2}} = B[x + y]$.
   – Jump to Step 7.
2. $\mathsf{C}$ increments $q$.
3. $\mathsf{C}$ picks a random bit $new_{\mathsf{C}}$ and random strings $b_{\mathsf{C}}, \beta_{\mathsf{C}} \in \{0,1\}^K$.
4. Parties invoke the $\mathsf{MPC}$ oracle for the following computation on inputs $(x, IH)$ from $\mathsf{S2}$ and $(y, d_q, new_{\mathsf{C}}, b_{\mathsf{C}})$ from $\mathsf{C}$:
   – Let $i = x + y$.
   – If $i$ does not occur in $IH$ let $i' = i$ and $b = 0^K$, and set the flag $new$ to 1;
     else let $i' = d_q$, fetch $(i, b)$ from $IH$ and set the flag $new$ to 0.
   – Return to $\mathsf{S2}$ the index $i'$, the bit $new_{\mathsf{S2}} = new_{\mathsf{C}} \oplus new$, and the string $b_{\mathsf{S2}} = b_{\mathsf{C}} \oplus b$.
5. $\mathsf{S2}$ lets $b' = B[i']$ and inserts $(i', b')$ into $IH$.
6. Parties invoke the $\mathsf{MPC}$ oracle for the following computation on inputs $(new_{\mathsf{S2}}, b', b_{\mathsf{S2}})$ from $\mathsf{S2}$ and $(new_{\mathsf{C}}, b_{\mathsf{C}}, \beta_{\mathsf{C}})$ from $\mathsf{C}$:
   – If $new_{\mathsf{S2}} \oplus new_{\mathsf{C}} = 1$ let $\beta = b'$ else let $\beta = b_{\mathsf{S2}} \oplus b_{\mathsf{C}}$.
   – Return to $\mathsf{S2}$ the string $\beta_{\mathsf{S2}} = \beta_{\mathsf{C}} \oplus \beta$.
7. Parties invoke $\mathsf{SisoPRF}$ with inputs $(N + \delta, K, r)$ from $\mathsf{S1}$, input $x$ from $\mathsf{C}$, and input $y$ from $\mathsf{S2}$. Let $Y_{\mathsf{C}}$ and $Y_{\mathsf{S2}}$ denote the outputs.
8. $\mathsf{C}$ outputs $Y_{\mathsf{C}} \oplus \beta_{\mathsf{C}}$ and $\mathsf{S2}$ outputs $Y_{\mathsf{S2}} \oplus \beta_{\mathsf{S2}}$.
```

Fig. 19. The query phase $\Pi_{\mathsf{SisoSublinSPIR}}.\mathsf{Query}$ for the functionality $\mathsf{SisoSublinSPIR}$ in the ($\mathsf{SisoPRF}$,MPC)-hybrid model

communication. Our tests were run on a desktop machine running inside a Ubuntu 12.04 LTS virtual machine with 8GB of RAM and 4 cores of an Intel i7-2600K 3.4GHz CPU assigned to it. Our testing focused on testing each of the cryptographic components of our solution, as well as the overall solution. We tested the main components: Oblivious Transfer, Shared-input-shared-output PIR, and secure node search, in both their online and offline (when applicable) phases. We also tested our setup phase for building the main database. We summarize our results as follows:

Processing a database from CSV to our format takes roughly 6ms per record, and then another 6ms to encrypt. Building the PPDS B-tree is much quicker, taking around 1 microsecond per record for each indexable column.

For online processing, we find that the dominant cost is in the usage of PIR. The generation of PIR instances can be done independent of the data size, only the number of elements, but the consumption of PIR depends on the data size. For 64-bit data and 64-bit pointers, a single instance PIR takes roughly 0.9 seconds when accessing something on the order of the leaf layer of a B-tree (100 million divided by the branching factor), though generating such an instance only takes 3.5 milliseconds (again, due to the fact that PIR generation is data-size independent). For OTs, since we only use them to transmit keys, we test them on fixed sizes of 128 bits though varying the number of instances. It takes a few microseconds to both produce and consume an instance of OT. Finally for secure nodesearch, we see that as the element size grows the nodesearch time grows linearly, corresponding to roughly 100 microseconds per bit of the data elements (and using 64-bit pointers). Overall it still takes much less than a second to evaluate for all tested data sizes.

B.1 Offline Processing

CSV to DB. We start with a flat table in CSV format using records with typical personal data (Name, DOB, etc.) and a medium sized payload field of 1 megabyte each and convert it into our internal representation in Figure 20.

Encryption of DB. Once we have a formatted database, we then encrypt each record so that we have an encrypted DB that will be sent to S2. Again, we run them on various collections of records in Figure 21.

Building of B-trees. We build privacy-preserving B-trees for our solution. Below in Figure 22 is our benchmarks in running time for generating a PPDS for data of 64-bit data, 64-bit pointers, branching factor 128.

B.2 Online Processing

Oblivious Transfer. We time the generation of random OTs to be consumed. These are 1-out-of-2 OTs of strings of length equal to the size of an encryption key (AES key). We generate N of them in batch at a time. These tests are performed over TLS over localhost, and presented in Figure 23. We also time the consumption of these OTs in Figure 24. Again, they are consumed in batches of size N.

Shared-input-shared-output PIR. We time the generation of random PIRs to be consumed. Note that these pre-generated PIRs do not depend on the sizes of the data elements, only the number of elements. We generate N of them in batch at a time. Again, these tests are performed over TLS, and presented in Figure 25.

We also time the consumption of these PIRs. We run PIR on elements of size equal to that of a node in a B-tree, which is equal to two data elements and one pointer per branching factor: $(2{\cdot}64{+}64){\cdot}128 = 24576$ bits. These are consumed one at a time, are performed over TLS, and presented in Figure 26.

Secure node search. We use an optimized Yao-style secure node search protocol. Here, we present the time for execution on instances of various data element sizes presented in Figure 27.

Actual Queries and Comparison to MySQL. We set up a database of 10 million records, where each record is roughly 0.5KB. We query the database using range queries that return roughly 1000, 10000, 50000, 100000, 250000, 500000, 750000, and 1 million records (which is 10% of the database). The raw times are presented in Figure 28. We consider the relative multiplicative overhead, which is presented in Figure 29. We see that although it starts off at over 100× due to fixed costs, it is approaching a factor of 5× for reasonably large queries, though it appears to have considerable overhead for smaller queries. It is apparent that this is due to our construction having additional fixed cost time that dominates small queries, and can easily be seen when plotting a trend between our times and the MySQL times. This is shown in Figure 30 along with the linear regression line showing a roughly 5× slope.

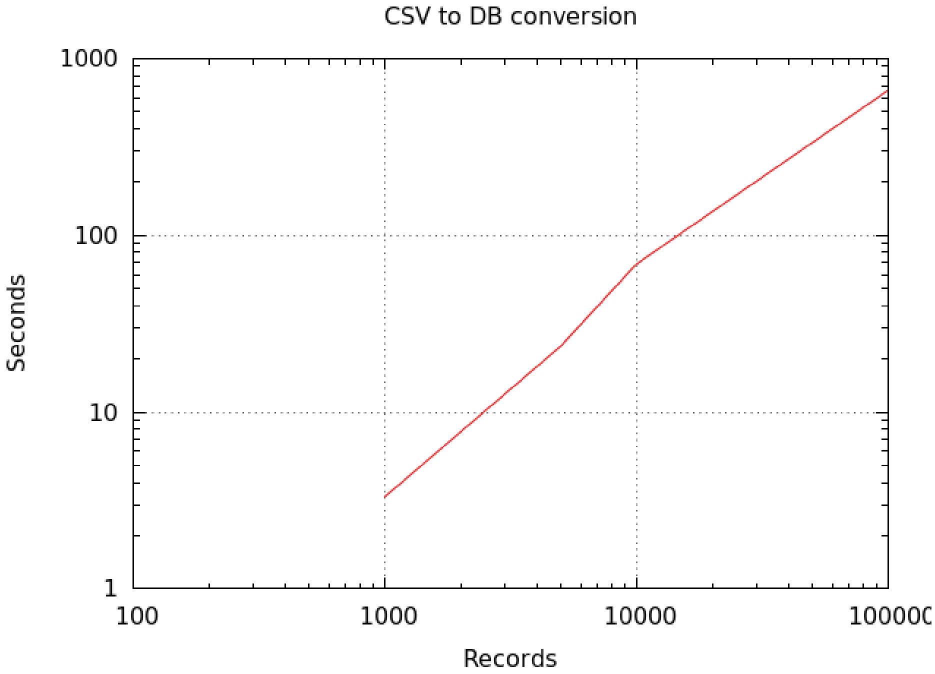

Fig. 20. CSV to Database

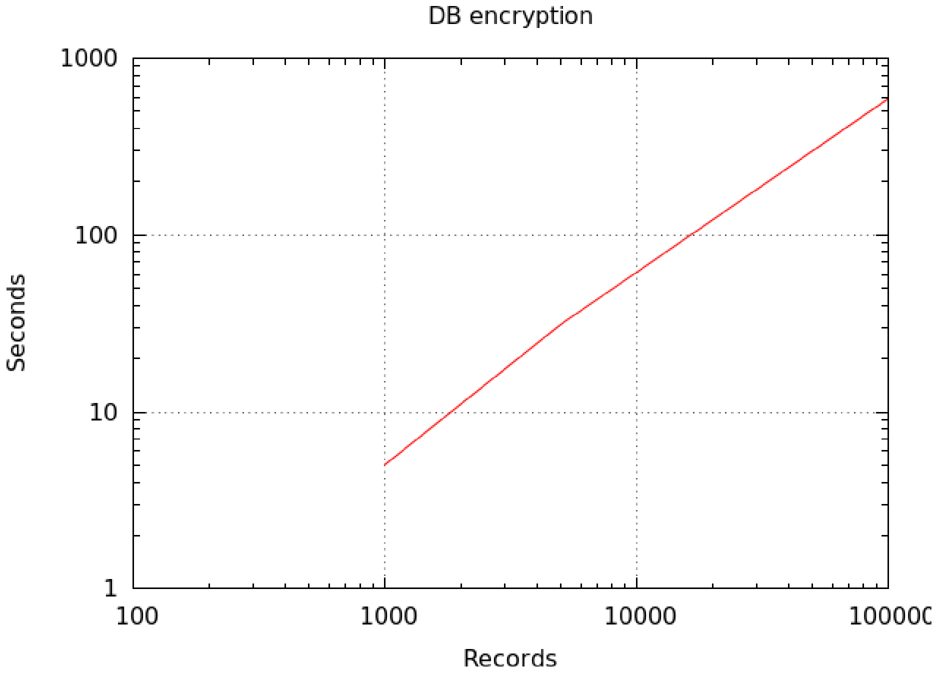

Fig. 21. Encrypt Database

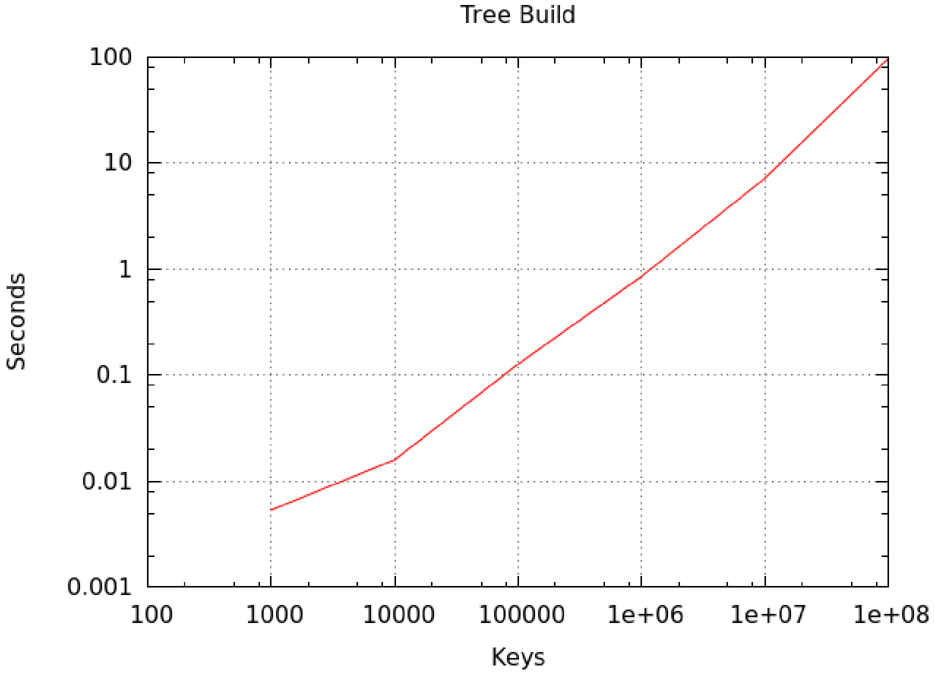

Fig. 22. Build Privacy Preserving B-tree

C Reduction to Range Queries

Other Query Types. A *keyword search* query q on a field A_i consists of a value s. The query returns all records r where its field element x_i contains s as a delimited keyword. A *stemming search* query q

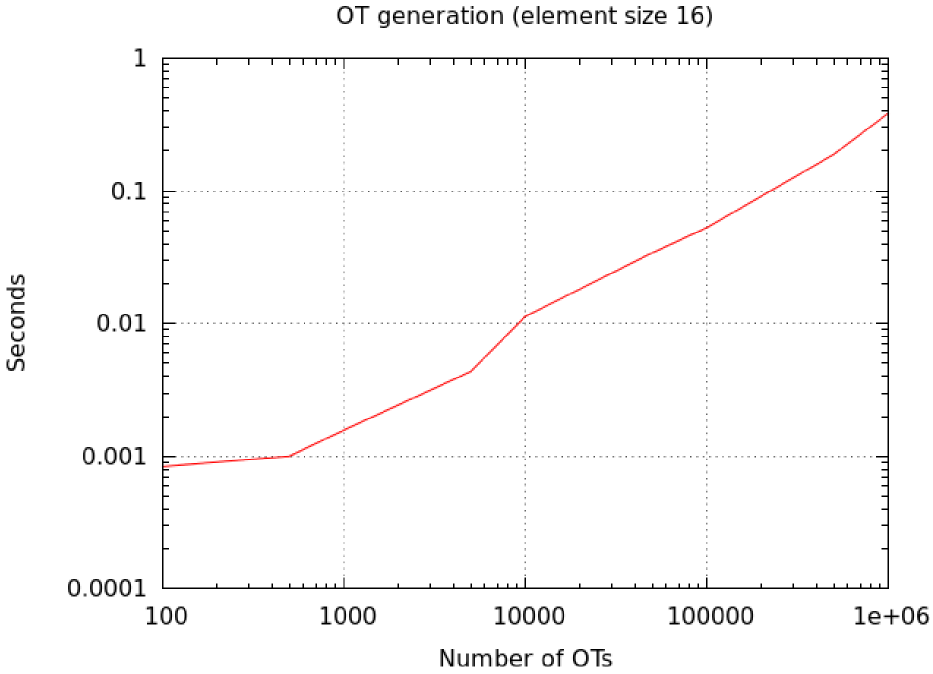

Fig. 23. OT Generation

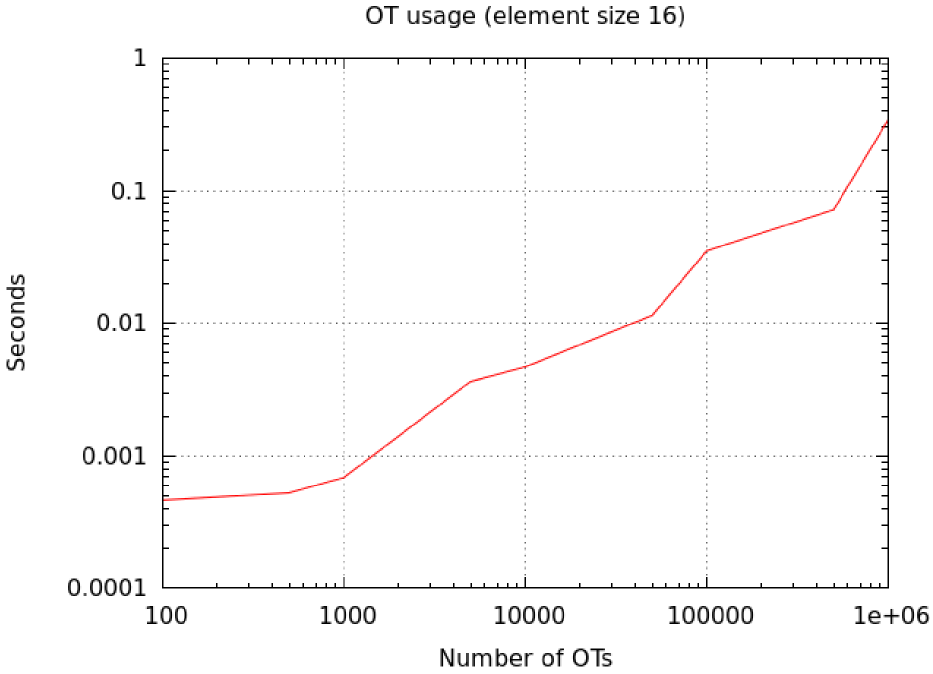

Fig. 24. OT Consumption

on a field A_i consists of a value s. The query returns all records r where its field element x_i contains some delimited keyword s' such that $STEM(s) = STEM(s')$, where $STEM$ is some publicly known stemming algorithm, e.g. a stemming dictionary. A *substring search* query q on a field A_i consists of

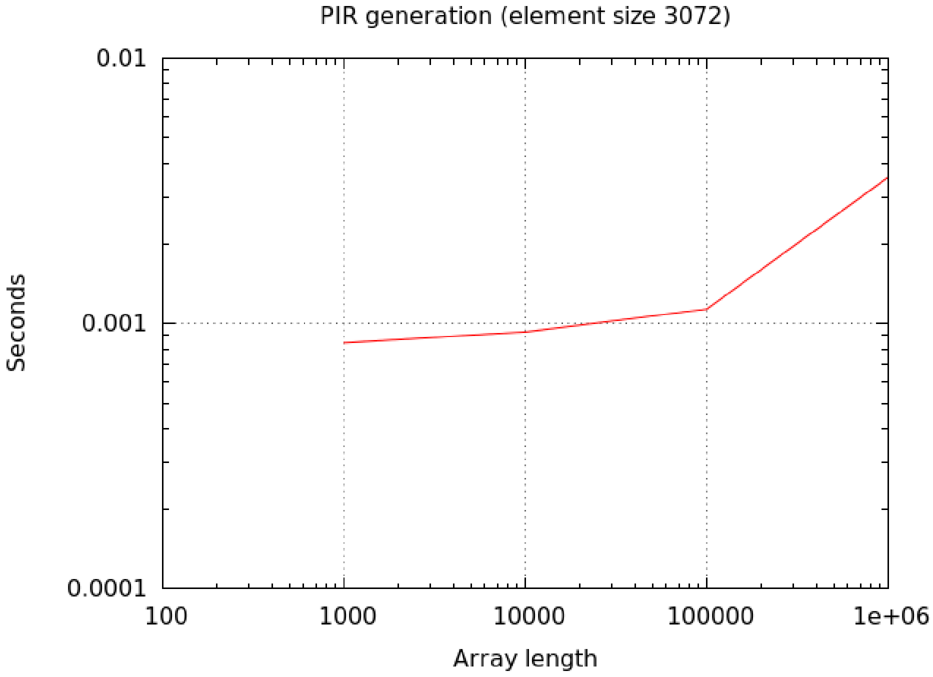

Fig. 25. PIR Generation

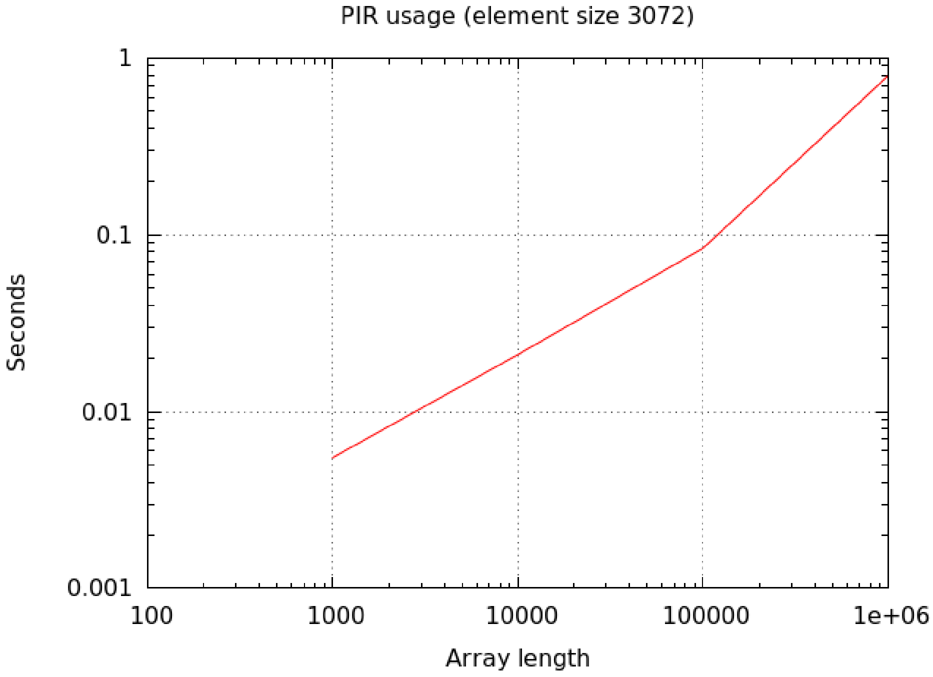

Fig. 26. PIR Consumption

a value s. The query returns all records r where its field element x_i contains s as an (initial, final, or intermediate) substring.

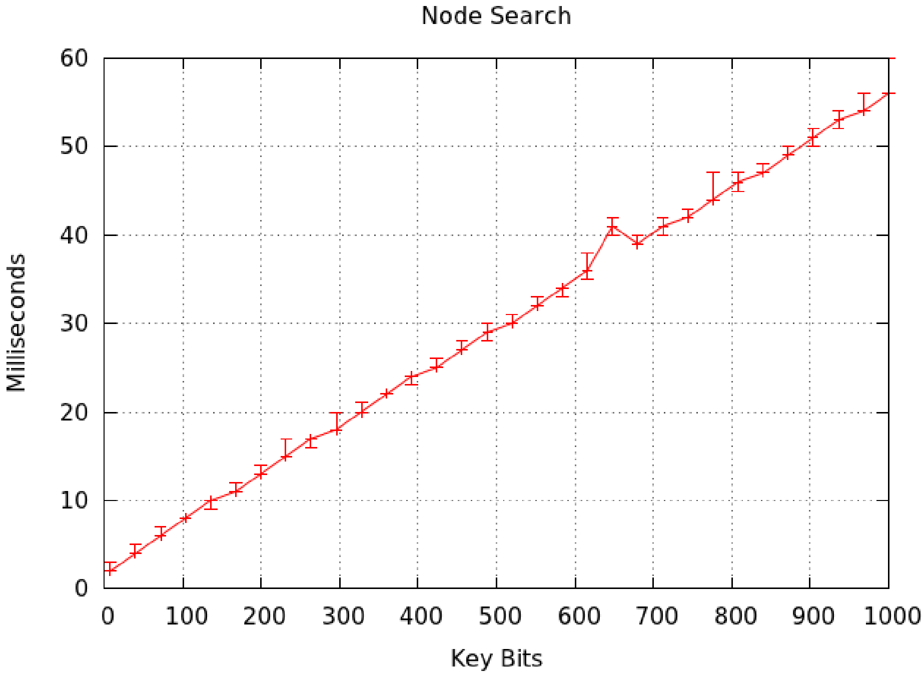

Fig. 27. Secure Node Search using Yao-based MPC

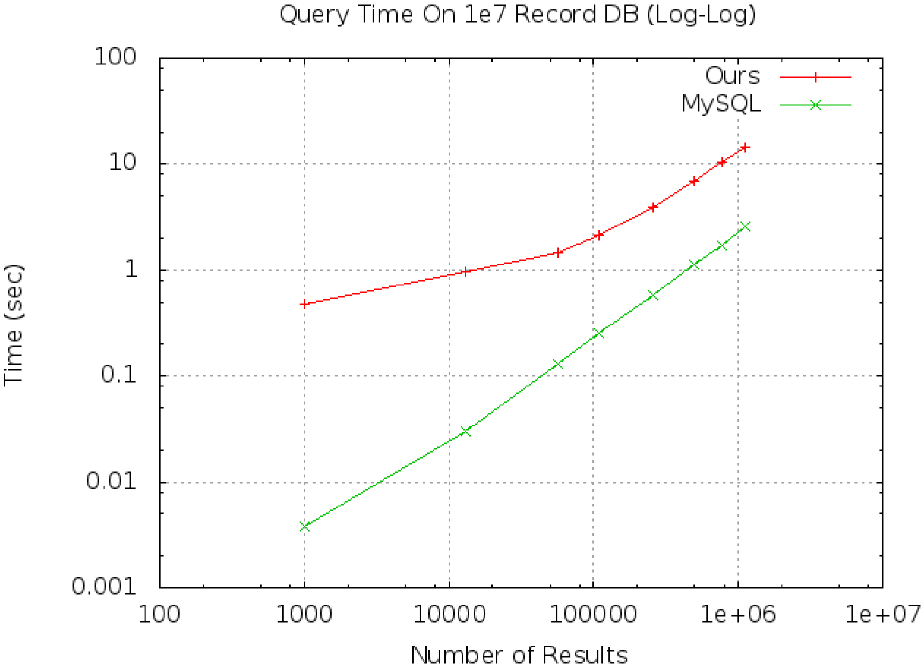

Fig. 28. Actual Query Times

Reducing Query Types to Range Search. We first describe how to reduce all of our proposed query types to range search: range/equality, keyword, stemming, and substring. To perform an equality search on some term q, we simply perform a range search on $[q, q]$. For the fields with keyword and stemming,

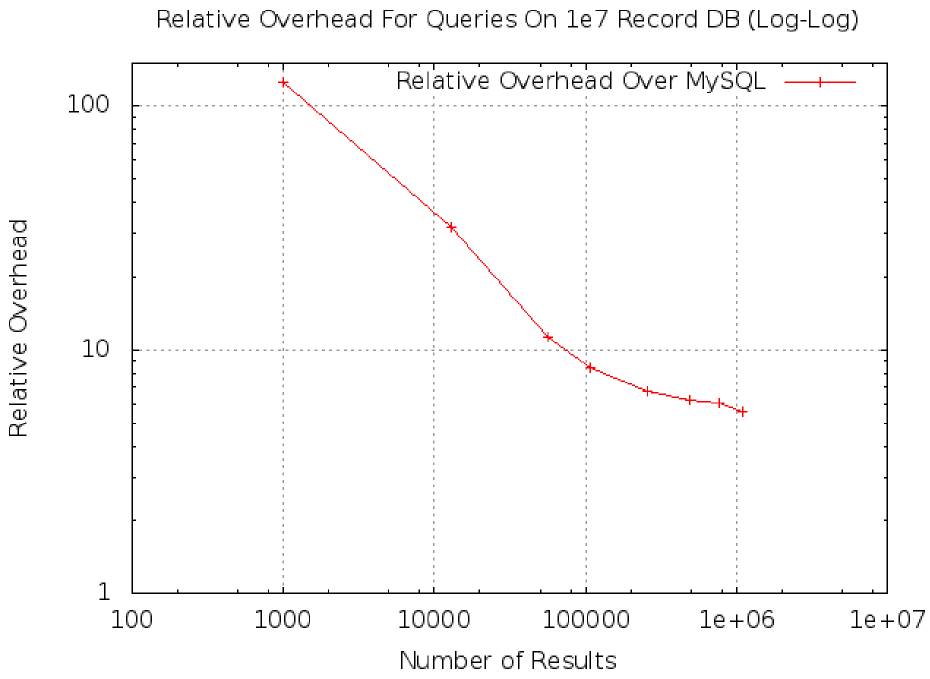

Fig. 29. Relative Query Times

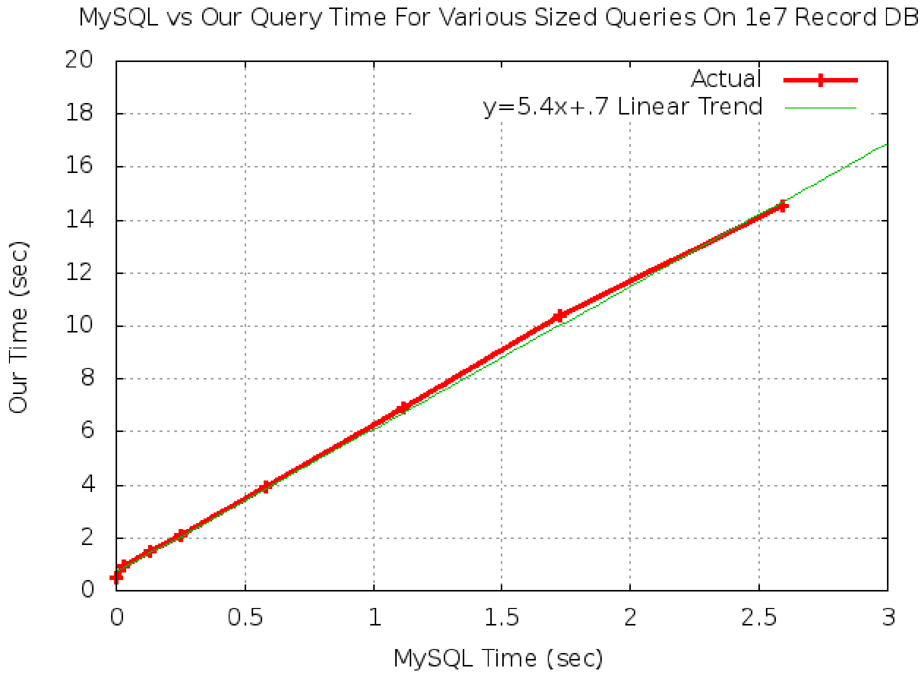

Fig. 30. Comparison To MySQL Trendline

we tokenize each word into keywords and stems (for stemming, we use the Porter stemming algorithm to demonstrate functionality, though any stemming algorithm/dictionary may be used in place of this),

then perform an equality search for the keyword or the stem of the keyword. For substring queries, we have three types of substrings: initial (q%), final q%, and intermediate substrings (%q%).
We have some substring length bound C, and given a term x, we append $C-1$ "initial" symbols and "terminal" symbols to the start and end of x. E.g. If the search term is "Alice", and our substring bound is 4 then we make the term "###Alice$$$". We then insert all size C substrings of that term into our data structure. Then, to perform an initial search on up to C characters on some query q, we pad q with sufficiently many # at the start to reach C characters and do an equality search. Similarly, for a terminal search, we append $ at the end of q and do an equality search. For an intermediate query, we perform a range search on $[q0\dots0, q1\dots1]$ which is just the string q appended with all zeroes or all ones at the end.

D Discussion of Applications

The solution is applicable in any scenario for secure and private database search that is in a three-party setting, where one party acts as a "semi-trusted" third party helper server. We summarize a few possible applications:

Secure Search. The main application is secure search, where one party holds sensitive data (hotel reservations, "no-fly" list, etc.) and another party has sensitive queries against it. The third party in this scenario is a neutral party agreed upon by both the data and query holders.

Private Forensics. Different entities may want to compare notes on data they collected from their own internal security team (bugs, vulnerabilities, attack patterns, etc.). The third party can be a repository for all the data, and each entity can query against it.

Data Collaboration. Different entities may want to collaboratively perform operations on their joint data. Again, the third party can be a repository for all the data, and each entity can query against it.

Secure Remote Database. With the growing popularity of storing data remotely, we want a way to do so privately when the data is sensitive. Here, the third party is the storage, and the Server can be the same entity as the Client.