Private Database Search with Sublinear Query Time

Keith B. Frikken, Boyang Li

Private Database Search with Sublinear Query Time

Keith B. Frikken and Boyang Li

Department of Computer Science and Software Engineering
Miami University, Oxford, OH 45056
frikkekb@muohio.edu, lib@muohio.edu

Abstract. The problem of private database search has been well studied. The notion of privacy considered is twofold: i) the querier only learns the result of the query (and things that can be deduced from it), and ii) the server learns nothing (in a computational sense) about the query. A fundamental drawback with prior approaches is that the query computation is linear in the dataset. We overcome this drawback by making the following assumption: the server has its dataset ahead of time and is able to perform linear precomputation for each query. This new model, which we call the precomputation model, is appropriate in circumstances where it is crucial that queries are answered efficiently once they become available. Our main contribution is a precomputed search protocol that requires linear precomputation time but that allows logarithmic search time. Using this protocol, we then show how to answer the following types of queries with sublinear query computation in this precomputation model: i) point existence queries, ii) rank queries, iii) lookup queries, and iv) one-dimensional range queries.

Keywords: Private Database Search, Secure Two-party Computation, and Precomputation

1 Introduction

There are many privacy/confidentiality concerns when querying a database about personal information. For example, if a user is querying information about a medical condition, religious beliefs, or political leanings then the query should remain private. Furthermore, a corporation asking a query may fear that revealing the query is a risk to their competitive advantage. Furthermore, even if the database owner is trusted, there is always the fear that a corrupt insider at the database owner's organization would leak the query information. The desire to protect the queries is not only a concern of the querying entity, but the database owner might not want this information due to liability concerns. One way to mitigate these concerns is to make the database publicly available, but this is not always an option. For example, if the database contains information about individuals, then revealing this information publicly may not be legal. Also, if the database owner wants to charge for queries, then revealing the information publicly is not a option. Example 1 gives a more detailed example that demonstrates the need for private querying.

Example 1. A federal agency wants the ability to query a transaction database to determine if a suspect (for which the agency has a warrant) is contained in the database.

Furthermore, the federal agency wants to keep the identity of the suspect private, because: i) revealing this information might compromise the investigation, and ii) to avoid possible litigation if the suspect is innocent. The owner of the database wants to help the agency, but does not want to violate the privacy rights of the other people in the database.

Any secure function evaluation/secure multi-party computation, such as [17], can be used to solve this private database querying problem. Unfortunately, these solutions require the server and the client to perform computation and communication that is linear in the size of the dataset. Overcoming this linear bound appears impossible. If the server doesn't "touch" every item in the dataset, then the server learns some information about the query. To overcome this linear bound, we introduce a new model, which we call the precomputation model. In this model, queries are divided into the following two phases:

1. *Precomputation Phase:* In this phase the server does computation on its dataset. Furthermore, the server generates a message that is sent to the client. This phase assumes that the server's information is known, but that the query is unknown, and this phase is allowed to require linear complexity. The precomputation message can be sent (perhaps on a DVD) to the client before the query is known.
2. *Query Phase:* After the query is made available, this phase captures the interaction between the client and the server.

The two main goals of a protocol in this precomputation model are: i) that the computation/communication of the interactive query phase and the client computation phase is sublinear in the size of the dataset, and ii) that the total computation/communication is "close" to that required by the general solutions. Returning to Example 1 the database owner may be willing to precompute information, so that when the query becomes available the federal agency will be able to obtain its result as quickly as possible. Furthermore, the owner might compute several messages and send the information to the federal agency before the query is being asked.

The rest of this manuscript is organized as follows: Section 2 introduces the problems which can be solved by our protocol and the contribution of this paper. Section 3 and 4 introduce the building blocks and new tools used in the remainder of the manuscript. In section 5, a protocol for the private database search problem in the precomputation model is given. Section 6 gives a sketch of the security analysis. In section 7, we present experiments and results of comparison between naive scheme and our protocol. Finally, Section 8 describes related work and Section 9 concludes the paper and gives future work.

2 Problem Definition and Contributions

We consider the following general database search problem
$SEARCH(s_0, \ldots, s_n, m_1, \ldots, m_{2n-1}\ ;\ q)$: where the server has a sorted sequence of points $s_0, s_1, \ldots, s_n$ and a sequence of messages $m_1, \ldots, m_{2n-1}$; furthermore a client has a query point q. Without loss of generality we assume $s_0 = -\infty$ and $s_n = \infty$

(this can be accomplished by padding the list with values that are smaller/larger than any q value). At the end of the protocol the client obtains message m_ℓ where: i) $\ell = 2i$ if $s_i = q$ and ii) $\ell = 2i - 1$ if $s_{i-1} < q < s_i$. That is, if the query point is in the server's dataset, then the client learns the corresponding message, and otherwise the client learns a message that is assigned to values between two search keys. The security requirement is that the server should learn nothing (computationally) about the query point, and the client should learn nothing other than the message. For example, the client should not learn if the query is an exact match or an in-between match, unless the messages reveals this information. Furthermore, the protocol should be secure against a semi-honest server and a malicious client.

This private database search problem can be used to solve the following types of common database queries in a private manner:

1. *Existence:* The server has a set $S = \{s_1, \ldots, s_{n-1}\}$ and the client has a query point q. The boundary of all elements in S and q is (s_0, s_n). At the end of the query the client should learn whether $q \in S$. Let Π be the permutation that sorts S and let $m_i = 1$ if i is even and otherwise let $m_i = 0$. The existence problem is solved by $SEARCH(s_0, s_{\Pi(1)}, \ldots, s_{\Pi(n-1)}, s_n, m_1, \ldots, m_{2n-1}\ ;\ q)$.
2. *Message Lookup:* Suppose that the server has a set of tuples where the first element is a key and the second element is a message associated with that key, i.e., $S = \{(s_1, m_1), \ldots, (s_{n-1}, m_{n-1})\}$, and that the client wants to lookup the message associated with key q. The boundary of the keys and q is (s_0, s_n). More formally, the client wants to learn m_i such that $s_i = q$ and if no such match exists then the client should learn $\perp$. Let Π be the permutation that sorts $\{s_1, \ldots, s_{n-1}\}$ and let $m_i = m_{\Pi(i)}$ if i is even and otherwise let $m_i = \perp$. The message lookup problem is solved by $SEARCH(s_0, s_{\Pi(1)}, \ldots, s_{\Pi(n-1)}, s_n, m_1, \ldots, m_{2n-1}\ ;\ q)$.
3. *Rank:* Another variation is for the client to learn the rank of its query in the set. That is, the server has a sorted sequence $s_1, \ldots, s_{n-1}$, the client has a query q, and the answer to the query is the value $|\{s_i : q > s_i\}|$. The boundary of all elements in S and q is (s_0, s_n). If we let $m_i = \lfloor \frac{i-1}{2} \rfloor$, then this is easily solved by $SEARCH(s_0, s_{\Pi(1)}, \ldots, s_{\Pi(n-1)}, s_n, m_1, \ldots, m_{2n-1}\ ;\ q)$.
4. *One-dimensional range query:* Suppose that the server has a set of points $S = \{s_1, \ldots, s_{n-1}\}$ and the client has a query interval $[a, b)$. Also, the boundary of a, b, and all elements in S is (s_0, s_n). The desired output of this protocol is $|\{s_i : a \leq s_i < b\}|$. This can be solved with two calls to search, but we postpone the discussion of this solution until section 3.4.

In this paper we are interested in solving the private database search problem in the precomputation model. In this model, the server is allowed to perform precomputation on the values for each query. Furthermore, the server is allowed to send the client a single message before the protocol begins. While this assumption is unreasonable in some environments, it is applicable in some situations (for example in the database search problems considered in the introduction). The goal of such a protocol is to minimize the time it takes to answer the query once the client's query is known. Moreover, the goals of the protocol are:

1. The precomputation phase should require at most linear computation.

2. The message should be at most linear in the size of the database.
3. The query phase of the protocol should require sublinear computation and communication. Also, the client should perform at most linear computation.

In this paper we introduce a new protocol for private database search that requires the server to perform $O(n)$ work in the precomputation phase and $O(1)$ modular exponentiations in the query phase. Furthermore, the size of the precomputation message is $O(n)$. The client and server perform $O(1)$ communication in the query phase and the client performs $O(1)$ modular exponentiations. Finally, the client performs only $O(\log n)$ computation during the query phase.

A related problem is that of keyword search [4]. This problem is identical to the message lookup protocol described above. While keyword search is less flexible, then the problem described above it is still useful to compare the efficiency of these two approaches for message lookup. In the keyword search protocol described in [4], the total communication is $O(polylogN)$ and the client performs only $O(\log N)$ modular exponentiations. However, in this protocol the server must perform $O(N)$ modular exponentiations and the query still requires $O(N)$ computation. Thus while the communication of this scheme is lower than the communication required by our scheme, our protocol has significantly more efficient query processing in the precomputation model. Table 1 compares the performance of the keyword search protocol for message lookup.

Table 1 also compares our scheme versus the standard naive scheme admitted by traditional SFE solutions (such as [17]). More specifically, this naive solution would be a circuit that performed $O(N)$ equality comparisons followed by a logical or of the results of theses comparisons. In this naive scheme, during the precomputation phase the server would compute the circuit and this garbled circuit would constitute the precomputation message. Note that the performance of these schemes is asymptotically the same in all aspects except client computation and query computation. Notice that our new scheme achieves a significant performance improvement in the query phase, which is the main motivation for the precomputation model.

Category	**Our Scheme**	**Naive Scheme**	**Keyword Search**
Server Comp	$O(N)$	$O(N)$	$O(N)$
Server Mod Exps	$O(1)$	$O(1)$	$O(N)$
Client Comp	$O(\log N)$	$O(N)$	$O(\log N)$
Client Mod Exps	$O(1)$	$O(1)$	$O(\log N)$
Precomp Message Size	$O(N)$	$O(N)$	0
Query Comm	$O(1)$	$O(1)$	$O(polylog(N))$
Query Phase Comp	$O(\log N)$	$O(N)$	$O(N)$

Table 1. Performance Comparison

3 Building Blocks

3.1 Notational Conventions

For $i \in \{0,1\}^b$, the binary representation of i is denoted by $i[b]i[b-1]\cdots i[1]$ where $i[b]$ is the most significant bit. The symbol $||$ is used to represent concatenation. When a value is used in a superscript and is surrounded by $()$ then this corresponds to a string label and not the value itself. When given a boolean value B, the value $\overline{B}$ is the complement of B. When specifying a protocol with two parties, the two parties inputs are separated by a semi-colon.

3.2 Oblivious Transfer

A well-known building block for privacy-preserving computations is chosen 1-out-of-k OT. In this protocol the sender inputs k values $v_0, \ldots, v_{k-1}$, the chooser inputs a choice $\sigma \in [0, k-1]$, and at the end of the protocol the chooser learns v_σ. Furthermore, the chooser should not learn anything about any values other than v_σ and the sender should not learn anything about σ. The OT functionality is defined as $((v_0, \ldots, v_{k-1}); \sigma) \mapsto (\perp \ ; \ v_\sigma)$ where $\perp$ is the empty string. In the remainder of this paper we only utilize the case where $k = 2$ and denote the OT protocol as $OT(v_0, v_1 \ ; \ \sigma)$. An efficient two-message protocol for OT was given in [13]. In this protocol the chooser and sender must perform $O(1)$ computation, modular exponentiations, and communication to achieve chosen 1-out-of-2 OT.

3.3 Permuted Encodings

A method, introduced in [17], for splitting a Boolean value, v, between two parties so that neither knows the value is as follows: one party chooses two encodings for the value w_0 and w_1 which are randomly chosen from a large domain[1]. The other party obtains the encoding w_v. The first party knows the meaning of the encodings, but does not know the actual value, and the second party knows the actual encoding value but does not know what it means. We use the variation, introduced in [14], which is: the first party chooses a permutation value λ, and the other party learns the values $v \oplus \lambda$ and w_v. This extra piece of information is useful to improve the efficiency of the underlying scheme. When given a b-bit value v, we use $ENCODE(v, \{(\lambda_i, w_0^i, w_1^i) : i \in [1, b]\})$ to denote the permuted encodings of each bit of v (i.e., $\{(v[i] \oplus \lambda_i, w^i_{v[i]}) : i \in [1, b]\}$). We use $EGEN(1^\kappa)$ to denote the process of generating a permuted encoding given a security parameter κ; that is, $EGEN(1^\kappa)$ produces a set of values $\{\lambda, e_0, e_1\}$ which are a permutation bit, a zero-encoding, and a one-encoding.

3.4 Scrambled Circuit Evaluation

Yao's scrambled circuit evaluation [17] allows for the computation of any function in a privacy-preserving manner. At a high level this approach works by creating a circuit that

[1] The domain must be large enough to prevent guessing

computes the desired function, and then one party, the generator, scrambles the circuit, and the other party, the evaluator, evaluates the scrambled circuit. The specific version of Yao's protocol that is used in this paper was described in [14]. This version of Yao's protocol was implemented in the Fairplay system [12] and was shown to be efficient for some problems. Recently, this technique has been proven secure in [11]. As this paper utilizes Yao's protocol extensively we review it next (we refer the reader to [14, 11] for a full description).

1. The generator creates the scrambled circuit as follows:
   (a) For each wire of the circuit, the generator chooses a permutation and two encodings (one for each possible value) of the wire.
   (b) For each gate of the circuit, the generator creates a PEGLT that will allow the user to obtain the permuted encoding of the output wire based on the permuted encodings of the two input wires.
2. The generator sends the gates' PEGLTs to the evaluator along with the permuted encoding values for all of the wires corresponding to generator inputs.
3. The generator and evaluator engage in a 1-out-of-2 OT protocol for each of the wires corresponding to evaluator inputs, where the evaluator learns the permuted encodings for these wires.
4. The evaluator uses the PEGLTs and the encodings of the input wires to obtain the permuted encodings for all wires of the circuit.
5. The result of the computation can either remain split or can be revealed to either participant. For example, to reveal the result to the evaluator, the generator simply sends the permutation bit for each output wire.

In the remainder of the manuscript we use the following notations:

- $CGEN_b(\circ, \{(\lambda_i, e_0^i, e_1^i) : i \in [1,b]\}, \{(\lambda_i', f_0^i, f_1^i) : i \in [1,b]\}, \{\lambda, g_0, g_1\})$ denotes the process of generating a circuit that compares to values with $x \circ y$ over b-bit values where $\circ \in \{=, \leq\}$. Furthermore, the permuted encodings for x and y are $\{(\lambda_i, e_0^i, e_1^i) : i \in [1,b]\}$ and $\{(\lambda_i', f_0^i, f_1^i) : i \in [1,b]\}$ respectively. Finally, the set of permuted encodings for the output wire is $\{\lambda, g_0, g_1\}$). As output this creates the gate gadgets (i.e., the PEGLTs) for the circuit computing operation $\circ$.
- $CEVAL_b(C, \{(x[i] \oplus \lambda_i, e_{x[i]}^i) : i \in [1,b]\}, \{(y[i] \oplus \lambda_i', f_{y[i]}^i) : i \in [1,b]\}$ evaluates the scrambled circuit C given the encodings for x and y. If C was generated for operation $\circ$, then the result of this evaluation is $r \oplus \lambda, g_r$ where r is the value of the predicate $x \circ y$. That is, this returns the permuted encoding of the result of the circuit.

Achieving One-dimensional Range Querying A tool for private database search can be used in conjunction with scrambled circuit evaluation to answer one-dimensional range queries. That is, suppose that the server has a set of points $S = \{s_1, \ldots, s_n\}$, the client has a query interval $[a, b)$, and the desired client output of this protocol is $|\{s_i : a \leq s_i < b\}|$. This can be computed by computing $rank_S(b) - rank_S(a)$ where $rank_S(x)$ is the rank of x in S. Thus to compute the result securely, the server can create a subtraction circuit, and then use two private database searches. In the first

search the client can obtain the encodings used in the subtraction circuit for $rank_S(a)$ and in the second search the client can learn the encodings in the subtraction circuit for $rank_S(b)$. Then the client can evaluate the circuit to obtain the desired result without revealing either $rank_S(a)$ or $rank_S(b)$.

4 New Tool: Chained PEGLTs

In this section we present a generalization of the gate gadget that is used in Yao's SFE [17] called Permuted Encrypted Garbled Lookup Tables (PEGLT). In this protocol the two parties have b bits split between them using permuted encodings. That is for Boolean values $v_1, \ldots, v_b$, the server has a set of encodings $\{(\lambda_i, e_0^i, e_1^i) : i \in [1, b]\}$ and the client has the corresponding values $\{(\lambda_i \oplus v_i, e_{v_i}^i) : i \in [1, b]\}$. Furthermore, the server has a set of messages $\{M_i : i \in \{0,1\}^b\}$ where each message is m bits long (in what follows we assume that $m = O(1)$). At the end of the protocol, the client should learn $M_{v_1 v_2 \cdots v_b}$ and nothing else, and the server should not learn anything about the split value.

Due to page constraints we do not give the protocol for a traditional PEGLT, but instead give a variation of PEGLTs where the client and server engage in n different PEGLTs where each PEGLT uses the same encodings as the previous PEGLT but has an additional encoding. That is the servers inputs in the successive protocols are $\{(\lambda_i, e_0^i, e_1^i) : i \in [1,1]\}, \{(\lambda_i, e_0^i, e_1^i) : i \in [1,2]\}, \ldots, \{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}$ and the clients inputs are $\{(\lambda_i \oplus v_i, e_{v_i}^i) : i \in [1,1]\}, \ldots, \{(\lambda_i \oplus v_i, e_{v_i}^i) : i \in [1,n]\}$. Furthermore, the server's messages are the sets $\{M_i^1 : i \in \{0,1\}^1\}, \{M_i^2 : i \in \{0,1\}^2\}, \ldots, \{M_i^n : i \in \{0,1\}^n\}$. Another requirement is that the server should be able to generate all of the lookup tables at the same time without interaction from the client, and the correctness and security requirements are the same as those in the previous section.

The main idea of this protocol is that server will choose $n+1$ sets of keys, denoted by $K_0, \ldots, K_n$ where $K_j = \{k_i^j : i \in \{0,1\}^j\}$. Now, these keys will be appended to the end of the messages used in the scheme; that is, for each $j \in [1,n]$ and $i \in \{0,1\}^j$, $\hat{M}_i^j = M_i^j || k_i^j$. At the jth PEGLT, the client will learn a single modified message from the set $\{\hat{M}_i^j : i \in \{0,1\}^j\}$, and thus will learn a single key from K_j. The key k_i^j will be used with the appropriate encoding to encrypt the messages $\hat{M}_{i0}^{j+1}$ and $\hat{M}_{i1}^{j+1}$. Essentially, key k_i^j is a compressed form of the encodings $e_{i_1}^1, \ldots, e_{i_j}^j$ in that client will be able to learn k_i^j if and only if it has $e_{i_1}^1, \ldots, e_{i_j}^j$. Thus the j PRF evaluations that were done for these encodings in the $(j+1)$st table can be replaced by a single PRF using this key.

In Figure 1 we describe the details of the table generation phase of chained PEGLT. In this scheme the server generates all n lookup tables, without interacting with the client.

To help clarify this protocol we do an example with $n = 2$. In this case the server has inputs $\{(\lambda_1, e_0^1, e_1^1), (\lambda_2, e_0^2, e_1^2), M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}\}$. For the sake of an example assume that $\lambda_1 = 0$ and $\lambda_2 = 1$. The server will generate three sets of keys from $\{0,1\}^\kappa$; denote these by $K_0 = \{k_\perp^0\}$, $K_1 = \{k_0^1, k_1^1\}$, and $K_2 =$

1. For $\ell = 0$ to n create a key set $K_\ell = \{k_i^\ell : i \in \{0,1\}^\ell\}$ where each k_i^ℓ is chosen uniformly from $\{0,1\}^\kappa$.
2. For $j = 1$ to n do the following steps:
   (a) For all $i = i_1 \cdots i_j \in \{0,1\}^j$, the server chooses $r_i \leftarrow \{0,1\}^\kappa$ and computes $i' = i'_1 \cdots i'_j = i_1 \oplus \lambda_1 || \ldots || i_j \oplus \lambda_i$. The server also creates a message $\hat{M}_i^j = M_{i'}^j || k_{i'}^j$. Then the server then computes $C_i^j = (r_i, F_{e_{i'_j}^j}(r_i) \oplus F_{k_h^{j-1}}(r_i) \oplus \hat{M}_i^j)$ where F is a pseudorandom function mapping $\{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^{m+\kappa}$ and $h = i'_1 \cdots i'_{j-1}$.
   (b) Create table $T_j = \{C_\ell^j : \ell \in \{0,1\}^j\}$.
3. Return the message $k_\perp^0, T_1, \ldots, T_n$.

Fig. 1. $GENTAB_n(\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}, 1^\kappa)$

$\{k_{00}^2, k_{01}^2, k_{10}^2, k_{11}^2\}$. Now the server creates two tables, the first of which is the ordered set $\{(r_0, F_{k_\perp^0}(r_0) \oplus F_{e_0^1}(r_0) \oplus (M_0^1 || k_0^1)), (r_1, F_{k_\perp^0}(r_1) \oplus F_{e_1^1}(r_1) \oplus (M_1^1 || k_1^1))\}$. The second table (which is the more interesting table) will be the ordered set (recall that $\lambda_1 = 0$ and $\lambda_2 = 1$):

$$\{(r_{00}, F_{k_0^1}(r_{00}) \oplus F_{e_1^2}(r_{00}) \oplus (M_{01}^2 || k_{01}^2)), (r_{01}, F_{k_0^1}(r_{01}) \oplus F_{e_0^2}(r_{01}) \oplus (M_{00}^2 || k_{00}^2)),$$
$$(r_{10}, F_{k_1^1}(r_{10}) \oplus F_{e_1^2}(r_{10}) \oplus (M_{11}^2 || k_{11}^2)), (r_{11}, F_{k_1^1}(r_{11}) \oplus F_{e_0^2}(r_{11}) \oplus (M_{10}^2 || k_{10}^2))\}$$

In the table lookup phase, the client will have the message $k_\perp^0, T_1, \ldots, T_n$ and it will sequentially obtain the permuted encodings for the value v. In Figure 2 we describe the details of the protocol for the jth lookup (where the user will learn a message and a key).

1. Let $\ell = (v_1 \oplus \lambda_1) || \cdots || (v_b \oplus \lambda_b)$ and lookup $C_\ell = (r_\ell, D_\ell)$ from table T_j.
2. Compute $\hat{M}_\ell = D_\ell \oplus F_{k_{v_1 \cdots v_{j-1}}^{j-1}}(r_\ell) \oplus F_{e_{v_j}^j}(r_\ell)$. Parse $\hat{M}_\ell$ into $M_{v_1 \ldots v_j}^j$ and $k_{v_1 \ldots v_j}^j$ and return these values.

Fig. 2. $LOOKUP_j(T_j, k_{v_1 \cdots v_{j-1}}^{j-1}, \{(v_\ell \oplus \lambda_\ell, e_{v_\ell}^\ell) : \ell \in [1,j]\})$

Returning to the example suppose that $v = 01$, and thus the client should obtain M_0^1 and M_{01}^2 from the first and second table lookup respectively. In the first table lookup the client has $k_\perp^0$, $v_1 \oplus \lambda_1 = 0$, and $e_{v_1}^1 = e_0^1$. The client takes entry 0 in T_1 (i.e., $(r_0, F_{k_\perp^0}(r_0) \oplus F_{e_0^1}(r_0) \oplus (M_0^1 || k_0^1))$) and computes $M_0^1 || k_0^1$, which is the correct message. Now in the second table lookup the client uses k_0^1 and $v_2 \oplus \lambda_2 = 0$ and $e_{v_2}^2 = e_1^2$ to decrypt entry 00 in the table T_2. That is, the client decrypts $(r_{00}, F_{k_0^1}(r_{00}) \oplus F_{e_1^2}(r_{00}) \oplus (M_{01}^2 || k_{01}^2))$ to obtain $M_{01}^2 || k_{01}^2$, which is what is expected.

In chained PEGLT server needs to perform only $O(1)$ pseudorandom functions per table entry. Since there are only $O(2^n)$ entries in all n tables, the server needs to perform

$O(2^n)$ computation. Furthermore, the client only performs $O(1)$ PRF evaluations per lookup, and thus performs only $O(n)$ computation. Finally, the efficiency of the above scheme can be improved slightly by removing the first encryption key and the last encryption key (i.e., keys is K_0 and K_n). However, this improvement does not change the asymptotic complexity of the protocol.

5 Private Database Search Protocol

In this section, the main result of this paper is presented. Specifically, a protocol for the private database search problem in the precomputation model is given that requires the client only perform sublinear (in the size of the database) computation and communication. In the private database search problem the server has a sorted sequence of points $s_1, s_2, \ldots, s_N$ (where each $s_i \in \{0,1\}^b$) and a sequence of messages $m_1, \ldots, m_{2N+1}$; furthermore the client has a query point $q \in \{0,1\}^b$. For ease of presentation we assume[2] that N is a power of 2 and denote $n = \log N$.

The main idea of this protocol is to use a standard binary search to achieve computation and communications that is logarithmic in the dataset size. We often refer to the binary search as a navigation through a complete binary search tree where the leaf nodes of the tree are the values in the server's set in sorted order. The difficulty with performing a private binary search is that the path of the search (i.e., whether the search goes left or right at a specific node) must be hidden from both the client and the server. This path cannot be revealed to the server because it would reveal a small range that contains the query, and the path cannot be revealed to the client, because this would reveal the rank of the query. Neither of these things are revealed by the result alone. To hide the search path, we utilize the well known technique of permuted encodings for these values. These permuted encodings are used in scrambled circuit evaluations to perform the comparison at each node on the search path, and chained PEGLTs are used to obtain the encodings for the nodes in the search tree. To make this discussion more concrete we present some formal notation.

We organize the values into a complete binary search tree, where the root node is denoted by T, and the intermediate nodes are denoted by $T_{i_1 i_2 \cdots i_m}$ where all i values are in $\{0,1\}$ and $m \in [1,n]$. The intermediate nodes are organized such that the left (resp. right) child of $T_{i_1 i_2 \cdots i_j}$ is $T_{i_1 i_2 \cdots i_j 0}$ (resp. $T_{i_1 i_2 \cdots i_j 1}$). The value of node $T_{i_1 i_2 \cdots i_j}$ is denoted by $v_{i_1 i_2 \cdots i_j}$. Note that the leaf nodes of the tree contain the values in set S is sorted order; that is, $v_i = s_i$ for all $i \in \{0,1\}^n$. The levels of the tree are denoted by $L_0, L_1, \ldots, L_n$ where L_0 is the root level of the tree and L_n is the leaf level. When performing a binary search on the tree, a comparison is made between q and a specific value at each level of the tree. We denote the result of the comparison at level L_j as R_j, where R_j is 0 (resp. 1) if q is less than or equal to (resp. greater than or equal to) the value at level L_j. Finally, note that the comparison at level L_j is between q and $v_{R_0 R_1 \cdots R_{j-1}}$.

The querier will obtain three types of permuted encodings in our scheme, including:

[2] It is straightforward to remove this assumption through padding.

1. *Querier's value:* These correspond to the permuted encodings for the value q. The permutation, zero encoding, and one encoding for bit $q[i]$ are denoted respectively by $\lambda^{(q),i}$, $e_0^{(q),i}$, and $e_1^{(q),i}$. The set of these values $\{(\lambda^{(q),i}, e_0^{(q),i}, e_1^{(q),i}) : i \in [1,b]\}$ is denoted by $PE^{(q)}$.
2. *Level L_j value:* These correspond to the permuted encodings for the server's value at the node for level L_j. The permutation, zero encoding, and one encoding for bit $L_j[i]$ are denoted respectively by $\lambda^{(L),j,i}$, $e_0^{(L),j,i}$, and $e_1^{(L),j,i}$. The set of values corresponding to level L_j, i.e., $\{(\lambda^{(L),j,i}, e_0^{(L),j,i}, e_1^{(L),j,i}) : i \in [1,b]\}$ is denoted by $PE^{(L),j}$.
3. *Comparison Results:* These correspond to the permuted encoding for R_j. The permutation, zero encoding, and one encoding for R_j are denoted respectively by $\lambda^{(R),j}$, $e_0^{(R),j}$, and $e_1^{(R),j}$. The set of these values corresponding to a specific level L_j, i.e., $\{\lambda^{(R),j}, e_0^{(R),j}, e_1^{(R),j}\}$, is denoted by as $PE^{(R),j}$.

Now that the notation has been defined, a more concrete view of our protocol is possible. The major steps of the protocol are as follows:

1. To bootstrap the system, the client and server engage in an OT protocol where the client learns the permuted encodings for q, the server sends the client the permuted encodings corresponding to v (the value at the root of the tree). These encodings are input into a scrambled comparison circuit which will split the value of R_0 in a permuted encoded format.
2. At each non-leaf level, L_j, a chained PEGLT is used to reveal the permuted encodings for $v_{R_0 R_1 \cdots R_{j-1}}$ to the client. These new encodings are then used in a scrambled comparison circuit to compare the value $v_{R_0 R_1 \cdots R_{j-1}}$ to q to obtain the permuted encoding for R_j.
3. When the leaf level is reached, a chained PEGLT is used to reveal the permuted encodings for $v_{R_0 R_1 \cdots R_{n-1}}$ to the client. These encodings are used along with the encodings for q in a scrambled circuit to reveal encodings for where q is less than, greater than, or equal to $v_{R_0 R_1 \cdots R_{n-1}}$. These two bits are used in the PEGLT to reveal the corresponding message.

5.1 Precomputation Phase

As input to the precomputation phase, the server inputs its set $S = \{s_1, \ldots, s_n\}$, a set of messages $m_1, \ldots, m_{2n+1}$, and a security parameter 1^κ. The full details of the precomputation phase are given in Figure 3.

5.2 Query Phase

In the query phase the client first performs a series of oblivious transfers that reveal to the client the permuted encodings of the query value q. Then, the client uses those encodings and the precomputation message to compute the result. Essentially, this phase uses the circuit at each level of the search tree to compare q and the current value of the node on the search path. The results of this comparison are used with the lookup tables from the precomputation phase to obtain the permuted encodings for the next node in the tree (see Figure 4).

1. *Choose permuted encodings/Setup:* Using $Egen(1^\kappa)$, the server chooses the following sets of permuted encodings: $\{(\lambda^{(q),i}, e_0^{(q),i}, e_1^{(q),i}) : i \in [1,b]\}$, $\{(\lambda^{(L),j,i}, e_0^{(L),j,i}, e_1^{(L),j,i}) : j \in [0,n], i \in [1,b]\}$ and $\{(\lambda^{(R),j}, e_0^{(R),j}, e_1^{(R),j}) : j \in [0,n]\}$. The server also creates a tree T that contains the values of S.
2. *Generate comparison circuits:* For $j \in [0, n-1]$ the server creates a scrambled comparison circuit that will be used to compare q and $v_{R_0 R_1 \cdots R_{j-1}}$ by using $C_j = CGEN(\leq, PE^{(q)}, PE^{(L),j}, PE^{(R),j})$.
3. The server creates a scrambled equality circuit for the last level of the tree that will be used to compare q and $v_{R_0 R_1 \cdots R_{n-1}}$ by using $C_n = CGEN((=), PE^{(q)}, PE^{(L),n}, PE^{(R),n})$.
4. *Generate PEGLTs:* Using chained PEGLT, the server creates a mechanism to use the R encodings to obtain the encodings for the values on the search path of q and for the server messages. That is for $j \in [1, n-1]$ let $M_i^j = v_{i_1 \ldots, i_j}$ and let $M_i^n = m_i$, then the server computes $LT = GENTAB_n(\{PE^{(R),i} : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}, 1^\kappa)$.
5. *Create message:* Form a message, we call it PM in the remainder of this section, consisting of the following elements and return it: $\{C_i : i \in [0,n]\}, LT, ENCODE(v, PE^{(L),0})$.

Fig. 3. Precomputation Phase(1^κ, S)

1. The client and server engage in b 1-out-of-2 OTs as follows : $OT(q[i] \; ; \; \lambda^{(q),i}||e_0^{(q),i}, \overline{\lambda^{(q),i}}||e_1^{(q),i})$ for all $i \in [1,b]$ where the client is the chooser and the server is the sender. The client gets $ENCODE(q, PR^{(q)})$.
2. For $i = 0$ to $n-1$ the client does the following:
   (a) $E_i = R_i \oplus \lambda^{(R),i}||e_{R_i}^{(R),i} = CEVAL(C_i, A, B_i)$ where $A = ENCODE(q, PR^{(q)})$ and $B_i = ENCODE(v_{R_0 \cdots R_{i-1}}, PE^{(L),i})$.
   (b) The client learns $ENCODE(v_{R_0 \cdots R_i}, PE^{(L),i})$ and $k_{v_1 \ldots v_{i+1}}^{i+1}$ by doing: $LOOKUP_{i+1}(T_{i+1}, k_{v_1 \ldots v_{i+1}}^i, E_i)$.
3. For the leaf level, the client does the following: $E_n = R_n \oplus \lambda^{(R),n}||e_{R_n}^{(R),n} = CEVAL(C_n, A, B_n)$ where $A = ENCODE(q, PR^{(q)})$ and $B_n = ENCODE(v_{R_0 \cdots R_{n-1}}, PE^{(L),n})$.
4. The client uses $R_n \oplus \lambda^{(R),n}$ and $\lambda^{(R),n}$ to learn the value of R_n which is the desired result.

Fig. 4. Query Phase($PM, q \; ; \; PE^{(q)}$))

5.3 Performance Analysis

Assume that $b = O(1)$, then in the precomputation phase the server creates $O(\log N)$ circuits each with size $O(1)$. The chained PEGLT will require $O(n)$ computation. In the interactive phase the scheme requires $O(1)$ 1-out-of-2 OTs each of which requires $O(1)$ computations/modular exponentiations. Hence, the query phase requires each

party to perform $O(1)$ modular exponentiations, and since these can be done in parallel, this phase requires $O(1)$ rounds. Finally, in the query phase the client has to evaluate $O(\log N)$ circuits each with $O(1)$ gates, and thus this requires $O(\log N)$ computation. The client also has to do the chained PEGLT lookup which requires $O(\log N)$ computation for all lookups. Thus the total query computation is $O(\log N)$. We summarize the performance of our scheme in Table 1 (in section 2).

6 Proof of Security

Due to page constraints we give only a sketch of the security analysis. This protocol is secure in the honest-but-curious adversary model[3]. The standard definition for security states that there should be a probabilistic polynomial time simulator that can produce a transcript that is computationally indistinguishable from the client's (resp server's) view of the real protocol when given the client's (resp. server's) input and output. For a formal definition see [7]. When proving the security of a protocol, the composition theorem of [2] is useful. This theorem states that if the protocol is proven secure when the protocol's building blocks are replaced by a version of those building blocks that utilize a trusted third party, then the protocol that results from the building blocks being replaced by secure implementations is also secure. Now, the server's view consists the results of the oblivious transfer during the interactive query phase where the server plays the part of the sender. In OT the sender does not have any output, and hence security against a dishonest server is straightforwad. To demonstrate client-side security, notice that all of the building blocks (OT, scrambled circuits, and PEGLT) reveal only permuted encodings to the client. Hence, these intermediate results are trivially simulateable.

7 Experiments

In this section, we present experiments and results of a comparison between the naive scheme and our protocol. The experiments are on a *Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz 2.67GHz* CPU and *2.00 GB* RAM. The operating system is Windows7 Enterprise (x64). The implementations are written in Java.

In this experiment, we implement the protocols solving the point existence queries problem, which is the server inputs a set of numbers S and the client inputs a number q to learn whether $q \in S$. We varied server's input size form 100 to 3000 in step of 100. For each input size we run each experiment 20 times and report the mean performance. The bit size of input number is 16.

Precomputation time (cf. Figure 5**)**. Our experiments shows the naive scheme costs linear time in the precomputation phase. Since only little time is needed in small input size by using our scheme, the performance for our scheme is not very obvious here. Clearly, our scheme is much faster than naive scheme in the precomputation phase. The reason is that our scheme generates less circuits than naive scheme and generating Chained-PEGLT is faster than generating circuits.

[3] Recall that an adversary is honest but curious if the adversary will follow the protocol, but will try to learn additional information

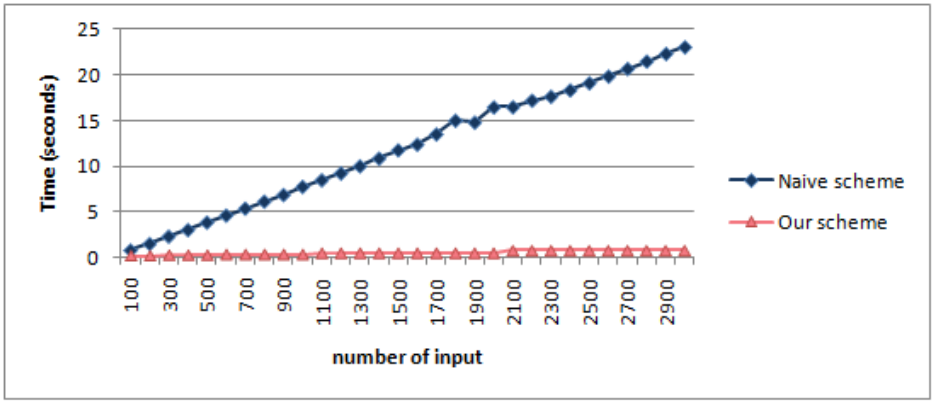

Fig. 5. Precomputation time

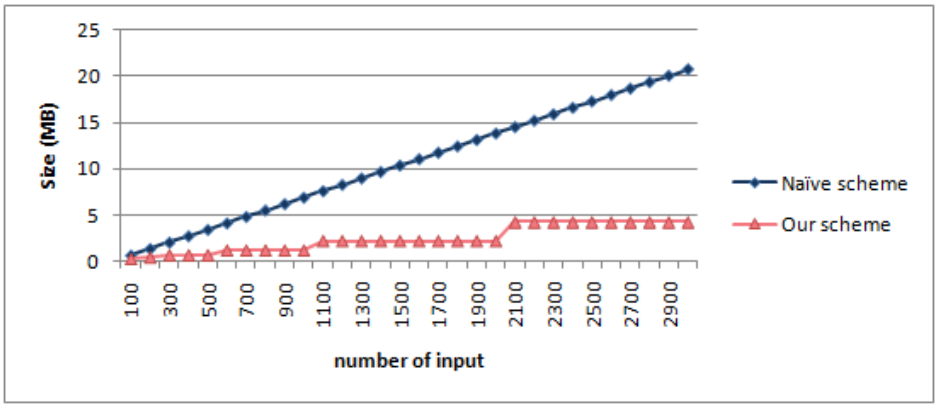

Fig. 6. Communication size

Communication size (cf. Figure 6). Both schemes require linear communication size. The data jumps in certain number of input for our scheme. That's because the size of message depends on the height of the search tree.

OT time. There is no difference between our scheme and naive scheme in OT time, because the client does same OT in both scheme for its input. Due to page constraints we do not provide the comparison figure.

Evaluation time (cf. Figure 7). Our scheme significantly improves the performance in evaluation time. In further experiments, the evaluation time for our solution is still under 0.002 seconds even server's input size increases to 50000.

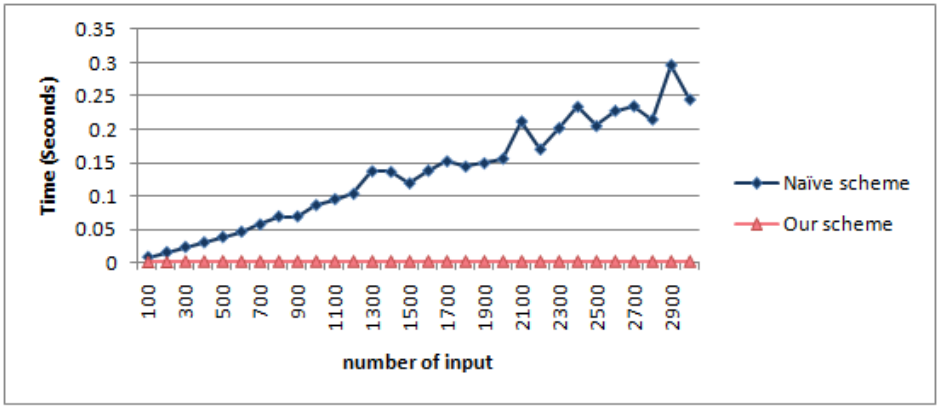

Fig. 7. Evaluation time

8 Related Work

Secure Multiparty Computation (SMC) is the problem of creating a privacy-preserving protocol for any function f[4]; that is, creating a protocol that computes f over distributed inputs while revealing only the result and inferences that can be made from this result. General results state that any function can be computed in such a secure manner. The first constructions for secure two-party SMC were given in [16, 17]; these assumed that the adversary of the protocol was honest-but-curious (HBC) in that the adversary will follow the protocol exactly but will attempt to make additional inferences. Later a construction was given for multiple parties [8] in the malicious adversary model (where the adversary deviates arbitrarily from the protocol) assuming that a majority of the participants are honest. There have also been many other papers attempting to improve the efficiency of these protocols to make the general results practical. However, to our knowledge all of these protocols require linear computation and/or communication when solving the private data querying problem.

An area that is related is private information retrieval (PIR) [3, 10, 1]. In PIR, the server has a sequence of bits $v_1, \ldots, v_N$ and the client has a specific index $i \in N$. The goal of PIR is that the client should learn v_i without revealing anything about the index to the serve, while requiring only sublinear communication. While PIR is related to accessing a database in a private manner, there are important differences between PIR and the work in this manuscript. First, it is not clear how PIR could be used to solve the problems solved in this manuscript. That is, PIR allows the client to access a specific bit, but this doesn't appear to solve problems like message lookup and range queries. Secondly, PIR requires linear computation, whereas the goal of this paper is to have sublinear computation in the query phase.

Another related problem is the area of oblivious RAM [9]. In oblivious RAM, a data owner wants to access a dataset but desires to hide the access pattern from an adversary that holds the data. Techniques have been developed which allow an access cost that

[4] We are assuming that f can be computed in polynomial time when given all of the inputs.

requires sublinear computation and communication (in an amortized sense). Furthermore, recent results [15] have shown that these schemes can be practical. However, the oblivious RAM model does not apply to the problems considered in this paper, because the Oblivious RAM model assumes that the accessing party has all of the data. In our case this would correspond to the client having all of the data and querying its own data.

9 Conclusions/Future Work

In summary, in this paper we introduce the precomputation model for privately accessing a database. In this model, the database owner performs linear precomputation on the dataset for each query, but this step can be completed without the query being fixed. We also present several protocols in this model where the query time is sublinear based on a new building block of a private database search. As future work we propose the following problems: A limitation of the current approach is that the precomputation must be done for each query. It would be interesting if the precomputation information could be shared for multiple queries. Perhaps the current techniques could be combined with the approach in [5] that uses fully homomorphic encryption [6]. Also, the current approach only works for the honest but curious model. An interesting extension would be to extend this to the malicious adversary model.

Acknowledgement

The authors would like to thank the anonymous reviewers for their comments and useful suggestions. Portions of this work were supported by Grant CNS-0915843 from the National Science Foundation.

References

1. Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. *Lecture Notes in Computer Science*, 1592:402–414, 1999.
2. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
3. Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.
4. Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer Berlin / Heidelberg, 2005.
5. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of CRYPTO*.
6. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.
7. Oded Goldreich. *Foundations of Cryptography: Volume II Basic Application*. Cambridge University Press, 2004.

8. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 218–229, May 1987.
9. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
10. Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, pages 364–373. IEEE Computer Society, 1997.
11. Yehuda Lindelll and Benny Pinkas. A proof of yao's protocol for secure two-party computation. In *Journal of Cryptology 22(2)*, pages 161–188, 2009.
12. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
13. Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.
14. Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139, New York, NY, USA, 1999. ACM.
15. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148, New York, NY, USA, 2008. ACM.
16. Andrew C. Yao. Protocols for secure computation. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.
17. Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.