ABSTRACT

# PRIVATE DATA QUERYING IN THE PRECOMPUTATION MODEL

by Boyang Li

Private data querying(PDQ) is about querying a database held by a server without: $i)$ revealing any information about the query to the server, and $ii)$ learning more information than the result of the query. Prior solutions to the PDQ problem require linear(in the size of the dataset) computation and communication, which is impractical for large datasets. In this thesis, we propose a new model for secure computation that separates PDQ protocols into two phases: a precomputation phase and a query phase. We introduce a scheme with sublinear computation and communication query cost under the assumption that the data owner can do a reasonable amount of computation at the precomputation phase. This assumption is reasonable in many environments when the data is known ahead of time and the queries are known at a later time. We introduce such protocols for the following database problems: existence problem, message lookup, the rank of query, one dimensional query, two dimensional range query, a small data change by the server.

# PRIVATE DATA QUERYING IN THE PRECOMPUTATION MODEL

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

Department of Computer Science and Software Engineering

by

Boyang Li

Miami University

Oxford, Ohio

2011

Advisor ______________________________

Dr. Keith Frikken

Committee Member ______________________________

Dr. William John Brinkman II

Committee Member ______________________________

Dr. Lukasz Opyrchal

# Contents

# List of Tables

# List of Figures

# Acknowledgement

I would like to express my sincere gratitude to my advisor, Dr.Keith Frikken, for his tremendous help. Throughout my two-year academic program, his guidance helped me in all the time of research and writing of this thesis. I appreciate it greatly.

I would like to thank my thesis committee members, Dr. William John Brinkman II and Dr. Lukasz Opyrchal for their suggestion and support. Also, I would like to thank all instructors and friends who have helped and inspired me during my study. Finally, I would like to thank my parents for supporting and encouraging me to pursue this degree.

## 1 Introduction

With the rapid development of computer technology, more and more organizations use computers to store large amounts of information. Furthermore, some of this information is private. Even though the information is private, the owners of the data may want to provide certain queriers with access to some of the data. However, in order to protect the privacy of the information, the data owner wants to provide only the results of the query; what complicates this matter further is that the querier does not want to reveal the content of the query. In the rest of this proposal, we will use "server" to denote the data owner and "client" to denote the data querier.

This technique is useful in many areas when a client queries a secret dataset and the query itself is also a secret. Consider the following situation: suppose that a federal agency (e.g., FBI, DHS, etc) wants to determine if a specific suspect has had a transaction with a bank. The federal agency should learn nothing beyond whether the transaction is in the system, because we assume that the federal agency only has authority (in the form of a warrant) to view information about the specific suspect. On the other hand, the bank should not know the request of the agency, including which records have been searched. The intent of this idea is to protect information from suspects. The reason that it is important to protect this information is two-fold: 1)the suspect is not necessarily guilty and revealing this information could damage an innocent suspect, and 2)an insider at the bank could leak information about a potential suspect. Hence our goal is that at the end of the interaction, the federal agency will only get the results of its query, and the bank learns nothing about the agency's query.

There are several papers such as [1] and [2] that give a solution for exchanging data in a private manner that can be used to solve this problem and similar problems. In these solutions, the communication and computation costs are linear in the size of the datasets. Clearly, these schemes are not scalable for large datasets. The goal of our proposal is to reduce the communication and computation costs required to run the protocol into something that is sublinear in the dataset.

Unfortunately, this goal is not achievable in the current framework, because the server has all the information of the dataset, and if the server does not "touch" every element in the dataset, then the server learns some of the elements which do not belong to the query. Leaking this information should not be allowed in the protocols, so the server must "touch" all elements in the dataset, hence the computation cost is at least linear in the size of the dataset.

Reconsider the previous example, the bank knows that the federal agency will ask a query about suspects at some point in time. The bank wants to supply the encryption data to the agency before the query, so that the query can be handled efficiently. Hence the goal is to minimize the cost to process the query once it becomes available. More formally, we separate the protocol into two phases: a precomputation phase and a query phase. In the precomputation phase, the server has the entire dataset and has the ability to do some precomputations. Also, the server can send a single message(perhaps it was stored in a DVD or CD) to the client ahead of time. During the query phase, the client uses the information from the precomputation phase and its input to query the database with sublinear computation/communication.

In this paper, we introduce such a new protocol and use the protocol to solve various problems. In addition, we provide a new tool, named Chained-PEGTT, which is a variation of Yao's scrambled circuit. We also implement the protocol and give the result of experiments to show the contribution.

The rest of this paper is organized as follows: Section 2 introduces the prior works for this problem and some related problems. Section 3 introduces the background technologies that we use in our protocol. Section 4 shows the protocol of Chained PEGTT. In section 5, a full protocol for the private database search problem in the precomputation model is given. We also give the varied protocols which can solve the other problems. Section 6 gives a formal proof of security for our protocol and the Chained PEGTT. In section 7, we present two experiments and the results, which are the comparison between naive solution and ours solution and the comparison for our scheme in different bit size. Section 8 concludes the paper and gives future work.

## 2 Prior Work

The two-party secure function evaluation is two parties $A$ and $B$ with private input values $x$ and $y$, jointly compute and respectively get the outputs $f_1(x, y)$ and $f_2(x, y)$. At the end of the protocol, the party $A$ only knows $x$ and $f_1(x, y)$ and the party $B$ knows $y$ and $f_2(x, y)$. More formally, the security means that no adversary can learn more knowledge other than input and output of the protocol and anything that can be computed from them in polynomial time. We assume that the adversary has semi-honest behavior, who follows the protocol but wants to compute information other than what he/she is authorized to get.

Yao's paper [1] gives protocol for secure function evaluation. Lindell and Pinkas summarized and proved the security of Yao's original protocol [3]. The basic idea of Yao's approach is to build a circuit for computing the function gate by gate, from the input wires to the output wires. The approach gave a technique that compute the results of any circuit without revealing any wire's values other than the output wires.

While the previous result is very general, several papers have attempted to achieve more efficient results for domain-specific problems. For example, [7] gives a protocol for set intersection. In addition, the papers [8] and [9] give protocols for the private comparison to determine which party has greater value. That is, two parties hold private input numbers $x$, $y$, respectively. At the end of the protocol, both parties can learn whether $x > y$ without revealing other informations.

Another problem which is similar to the problem that we were solving is Private Information Retrieval(PIR, [4], [15] and [5]). PIR is a problem that the server has a n-bit string $x = x_1 x_2 \dots x_n$ and the client tries to get a specific bit $x_i$ from the server. Furthermore, the server does not obtain any information about $i$, such as "$i \neq 67$" or "$i \leq 51$". The paper [4] presents several solutions to the problem by using $k$ copies of the dataset ($k \geq 2$) if the server has unbounded computing power. The paper also shows that models that can minimize the communication complexity into a sublinear dataset size when the server has computational limit. But in these models, the client learns more information than he/she should get. Although [4] then shows that the symmetric PIR can hide the private data from both parties, the computation complexity will still be linear. In summary, we can use none of the models represented by [4] to minimize the computation complexity. In addition, PIR is only a simple problem querying for one bit. Hence the solution for PIR is not useful to solve our problem. Furthermore, the paper [6] analyses that the PIR is not practical in the real world due to rapid improvement of hardware and bandwidth.

Before [2], *secure function evaluation(SFE)* was mainly considered a purely theoretical tool. With development of the cryptographic techniques and the speed up of hardware, [2] showed that SFE can be practical. They designed a high-level programming language, the *Secure Function Definition Language* (SFDL), and a lower-level language, the *Secure Hardware Definition Language* (SHDL), which can be used to generate Boolean circuits. They created a compiler that can translate a SFDL file into a SHDL file, and therefore, convert the higher-level program into a circuit. Later, the paper gives the implementation of a specific Two-Party SFE protocol based on the protocol suggested by [1]. Finally, the paper showed experimentally that SFE was practical for some problems.

# 3 Background

## 3.1 Notation

Given value $X \in \{0,1\}^n$, let $X^{(0)} \ldots X^{(n-1)}$ be binary representation of $X$ in big endian, and as a short hand notation let $X^{(i,j)}$ be the bit string $X^{(i)}\ X^{(i+1)}\ \ldots\ X^{(j)}$.The notation $X^{[\alpha,\beta]}$ denotes $X^{(\alpha\beta,(\alpha+1)\beta-1)}$. Let $\{k_i^j : i \in [a_1,a_2], j \in [b_1,b_2]\}$ denote a set with elements $k_i^j$ for each $i \in [a_1,a_2]$ and for each $j \in [b_1,b_2]$. For example, $\{k_i^j : i \in [0,1], j \in [0,1]\}$ indicates that the set has elements $k_0^0, k_0^1, k_1^0, k_1^1$. Let $[M : S]$ denote an array with subscript $S$. For example, $[M : \{0,1\}^m]$ represents an array of $2^m$ message from $m_{0^m}$ to $m_{1^m}$.

## 3.2 Oblivious Transfer

We will use oblivious transfer as a tool to initialize our protocol. The first paper that made the notion of “oblivious transfer”(OT) is Rabin’s [10]. There are many equivalent forms of it. In this proposal, we use 1-out-of-2 oblivious transfer ($OT_1^2$) which was given by Naor and Pinkas[12]. In the $OT_1^2$ protocol, the server inputs two string values $x_0$, $x_1$ and the client inputs a bit $\sigma \in \{0, 1\}$. At the end of the protocol, the client learns the string $x_\sigma$. What makes this oblivious is that the server learns nothing about the client’s input while the client learns nothing other than $x_\sigma$. The cost of $OT_1^2$ is $O(1)$ communication and $O(1)$ modular exponentiations. We use the notation $OT(x_0, x_1; \sigma)$ to denote oblivious transfer in the rest of this proposal.

## 3.3 Yao’s Protocol

In this section, we introduce Yao’s scrambled circuit, which allows us to do secure function evaluation. The circuits are constructed by a series of gate and the connection wires including the input wires, the output wires, and the intermediate wires. There are two parties involved in its whole evaluation process where one party is a circuit generator and other is a circuit evaluator.Lindell and Pinkas [3] proved this protocol is secure against semi-honest adversaries. The generator first computes the circuit and then computes a scrambled circuit. The evaluator is then given the scrambled circuit and evaluates it to obtain the result. Let’s assume that a circuit has $m$ wires inputs and $n$ gates where $m \leq n$. The complexity of the protocol is as following: the number of modular exponentiation is $O(m)$, the number of computation complexity is $O(n)$, the communication complexity is $O(n)$, and the number of round is $O(1)$.

As the protocols proposed in this manuscript rely heavily on the technical details of this protocol, we now describe these details. At first we clarify some notations about the circuit that we use. In this manuscript, we use capital letters represent sets and lower case represents individual elements. Also, the digit of letter’s subscript denotes the serial number of that element. We use the notation $C$ to denote a circuit. The notation $GT = \{gt_1 \dots gt_n\}$ to denote the gates of the circuit and the notation $W = \{w_1 \dots w_m\}$ to denote the wires in the circuit. Let $w_i.val$ denotes the value of the wire when the evaluator evaluates the circuit. The notation $c_i.w_j$ denotes the circuit $c_i$'s wire $w_j$. We also use the notation $gt_i.I$ to denote the $gt_i$'s input wires and $gt_i.O$ to denote its output wires. $|gt_i.I|$ denotes the number of inputs wires for $gt_i$ There are four types of wires in a circuit: the generator input's wires, the evaluator input's wires, the intermediate wires, and the output's wires. We use the notations $G$, $E$, $IN$, and $OU$ to denote these wires respectively and $I$ denotes the union of $G$ and $E$. During the generation phase, the generator chooses two random strings $k^0_{w_i}$ and $k^1_{w_i}$(we define them as "encodings") for each $w_i \in W$ - one for representing 0, the other for 1. For each wire, the evaluator is going to obtain only one of these encodings, but it will not know the real value of the encoding. Because $k^0_{w_i}$ and $k^1_{w_i}$ have identical distributions, the evaluator will not know what the encodings represent when it gets them. To be more clear, the Table 1 summarizes the notations above.

| **Notation** | **Description** |
|---|---|
| $C$ | circuit |
| $GT$ | Gates of the circuit |
| $W$ | Wires in the circuit |
| $w_i.val$ | Real value of the wire during the running phase |
| $c_i.w_j$ | The circuit $c_i$'s wire $w_j$ |
| $gt_i.I$ | The $gt_i$'s input wires |
| $gt_i.O$ | The $gt_i$'s output wires |
| $\|gt_i.I\|$ | The number of inputs wires for $gt_i$ |
| $G$ | The generator input's wires |
| $E$ | The evaluator input's wires |
| $IN$ | The intermediate wires |
| $IO$ | The output's wires |
| $I$ | The union of $G$ and $E$ |

Table 1: Notation

The main difficulty of Yao's scrambled circuit is how does the evaluator obtain the correct single encoding for each wire without learning the other encoding. Generally speaking, the techniques required depend on the type of wire. There are three cases: $i)$the generator input $G$, $ii)$the evaluator input $E$, and $iii$)the intermediate wires and output wires. For the generator input $G$, since the encodings is blind for the evaluator, the generator can simply send $G$ to the evaluator. For the evaluator input $E$, the evaluator can obtain $E$ by doing oblivious transfer with the generator. However, the third case is the most involved situation. In the next section, we describe how this is done.

### 3.3.1 Permuted-Encrypted-Garbeled-Truth-Table

The problem now becomes given the encodings for the input wires, how does the evaluator obtain the encodings for the other wires. We use Permuted-Encrypted-Garbeled-Truth-Table($PEGTT$) to achieve this. The generator creates a $PEGTT$ for each gate in the circuit. Using a $PEGTT$ for a gate and the encodings for the input wires to that gate, the evaluator can obtain the encoding of the output wire for that gate. Using the $PEGTT$s, the evaluator thus can obtain the encodings for all wires in the circuit. In order to explain $PEGTT$ in more detail, we now show how the $PEGTT$ evolves from a standard truth table using the example of an or-gate(see Figure 1).

| $w_1$ | $w_2$ | $w_3(w_1\|w_2)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Figure 1: The Standard Truth Table for an or-gate

Instead of using Boolean values in the truth table, the garbled truth table(see Figure 2) uses the wire encodings. Clearly, the $GTT$ does not solve the problem at hand, because it reveals all encodings to the evaluator.

The question thus become, how to let the evaluator obtain only a single encoding per wire. One way to prevent the evaluator from learning more than one encoding in the table is to encrypt each output encoding by using the corresponding input encoding. As an example of this see Figure 3. For each encoding of $w_3$, the table encrypts it by using x-or from all its entry encodings,

| $w_1$ | $w_2$ | $w_3(w_1\|w_2)$ |
|---|---|---|
| $k_{w_1}^0$ | $k_{w_2}^0$ | $k_{w_3}^0$ |
| $k_{w_1}^0$ | $k_{w_2}^1$ | $k_{w_3}^1$ |
| $k_{w_1}^1$ | $k_{w_2}^0$ | $k_{w_3}^1$ |
| $k_{w_1}^1$ | $k_{w_2}^1$ | $k_{w_3}^1$ |

Figure 2: The Garbled Truth Table for an or-gate

$Enc_{k_{w_1}^{w_1.val} \oplus k_{w_2}^{w_2.val}}(k_{w_3}^{w_3.val})$. In this case, the evaluator can get a single output encodings only if it has the entry encodings of the output. We call this version of the truth table an Encrypted-Garbled-Truth-Table($EGTT$).

| $w_1$ | $w_2$ | $w_3(w_1\|w_2)$ |
|---|---|---|
| $k_{w_1}^0$ | $k_{w_2}^0$ | $Enc_{k_{w_1}^0 \oplus k_{w_2}^0}(k_{w_3}^0)$ |
| $k_{w_1}^0$ | $k_{w_2}^1$ | $Enc_{k_{w_1}^0 \oplus k_{w_2}^1}(k_{w_3}^1)$ |
| $k_{w_1}^1$ | $k_{w_2}^0$ | $Enc_{k_{w_1}^1 \oplus k_{w_2}^0}(k_{w_3}^1)$ |
| $k_{w_1}^1$ | $k_{w_2}^1$ | $Enc_{k_{w_1}^1 \oplus k_{w_2}^1}(k_{w_3}^1)$ |

Figure 3: The Encrypted Garbled Truth Table for an or-gate

Given the input encodings the evaluator must know which value to decrypt. In order to do this, the table should connect the encoding with a bit that indicates the encoding. However, the problem is that the $EGTT$ will not hide the additional bit of the encoding in the table. In this case, the evaluator can learn what the encoding represents for according to the additional bit. The permutation mechanism is used to mitigate this problem. The generator first assigns a permutation number $\lambda_i \in \{0, 1\}$ for each wire $w_i$ in the table. Then, the generator creates a table called a Permuted-Encrypted-Garbeled-Truth-Table($PEGTT$). The table reveals the encodings of $W_i.val$ with a permutation position $W_i.val \oplus \lambda_i$ to the evaluator. The new table has $2^{|gt.I|}$ entries. Each entry is an arrangement of permutation positions for all input wires. The permutation bit will conceal the meaning of the additional bit from the evaluator, because it is permuted 50% of the time. To continue our example before, if we assume the permutation numbers are $\lambda_1 = 0, \lambda_2 = 1, \lambda_3 = 1$, we will get the table shows as Figure 4.

| $w_1$ | $w_2$ | $w_3(w_1\|w_2)$ |
|---|---|---|
| 0 | 0 | $Enc_{k^0_{w_1} \oplus k^1_{w_2}}(k^1_{w_3}\|\|0)$ |
| 0 | 1 | $Enc_{k^0_{w_1} \oplus k^0_{w_2}}(k^0_{w_3}\|\|1)$ |
| 1 | 0 | $Enc_{k^1_{w_1} \oplus k^1_{w_2}}(k^1_{w_3}\|\|0)$ |
| 1 | 1 | $Enc_{k^1_{w_1} \oplus k^0_{w_2}}(k^1_{w_3}\|\|0)$ |

Figure 4: The PEGTT for an or-gate

### 3.3.2 Getting the result

There are two situations after the evaluator evaluates the circuit. If the goal is for the evaluator to obtain the result, then the generator can reveal the permutation number for the output wires. On the other hand, if the goal is for the generator to learn the output, then the evaluator can send the encoding for the output wires to the generator.

### 3.3.3 Putting all of the details together

There are two parties involved in Yao's scrambled circuit: the generator which generates the circuit and the evaluator which evaluates it. $X \in \{0,1\}^n$ denotes the generator's input and $Y \in \{0,1\}^n$ denote the evaluator's input. The output of the circuit is a bit. Now we give the whole protocol below.

1)The generator secretly assigns two random strings $k^0_{w_i}$ and $k^1_{w_i}$ for each wire $w_i \in W$ and a permutation number $\lambda_{w_i}$.

2)The generator generates $PEGTT_i$ for each gate $\{gt_i : i \in [1,n]\}$ in the circuit. We use $[M_{gt_i} : \{0,1\}^{|gt_i.I|}]$ to denote the array of output encodings. For each entry of the table $\beta_1 \dots \beta_{\ell_{|gt_i.I|}} (\beta_i \in \{0,1\})$, the generator sets the output $Enc_{\bigoplus_{j=1}^{|gt_i.I|} k^{\beta_j - \lambda_j}_{gt_i.I_j}}(m_{\beta_1 - \lambda_1, \dots \beta_{\ell_i} - \lambda_{\ell_i}})$.

3) For each $g_i \in G$, the generator determines its input encoding as $k^{X^{(i)}}_{g_i}$. The generator sends the $\{PEGTT_i : i \in [1,n]\}$ and the input encodings $k^{X^{(i)}}_{g_i}$ to the evaluator.

4)The evaluator gets its input encodings $E$ from the generator by using $OT^1_2$. For each wire $e_i \in E$, the two parties run the protocol $OT(k^0_{e_i}, k^1_{e_i}; Y^{(i)})$. The evaluator will get the $\{k^{Y^{(i)}}_{e_i} : i \in$

$[0, |E| - 1]\}$ at the end of the protocol.

5)The evaluator evaluates the circuit gate by gate according to the $PEGTT$s and the input encodings $G$ and $E$. For each gate $gt_i$, the evaluator first obtains all input wire's encodings $\{gt_i.I_j : j \in [0, |gt_i.I| - 1]\}$ and then searches the $PEGTT_i$ to get the output encoding $gt_i.O$. Finally, the evaluator can get all of the wires' encodings in the circuit. The result of the circuit $OU$ is the output from the last gate.

## 3.4 Building Block

We have several building blocks using to generate our protocol. These are PEGTT generation function, PEGTT evaluation function, the circuit generation function, and the circuit evaluation function.

### 3.4.1 PEGTT generation function and evaluation function

By giving all input encodings of wires, the $PEGTT$ generation building block can generate a $PEGTT$. It is represented as $PEGTTGen(\Lambda, \{k^i_{w_j} : i \in [0, p-1], j \in [1, \ell]\}, \lambda_O, [M : \{0, p-1\}^\ell])$. $\Lambda \in \{0, p-1\}^\ell$ denotes a permutation number array of $\ell$ length. In addition, $\{k^i_{w_j} : i \in [0, p-1], j \in [1, \ell]\}$ denotes the input encodings of the table while $[M : \{0, p-1\}^\ell]$ denotes the array of output encodings. Furthermore, let $m$ denote an element of $M$. The table is created as following: for each entry of the table $\beta_1 \ldots \beta_\ell (\beta_i \in [0, p-1])$, the protocol gives a value $Enc_{\bigoplus_{i=1}^{\ell} k_{w_i}^{\beta_i - \lambda_i}}(m_{\beta_1-\lambda_1,\ldots\beta_\ell-\lambda_\ell} || m_{\beta_1-\lambda_1,\ldots\beta_\ell-\lambda_\ell}.val \oplus \lambda_O)$. At the end of the function, it returns a $PEGTT$ that has been generated.

We use the notation $PEGTTEval(tb; k_{w_i} || \{u_i : i \in [0, \ell]\})$ to denote the $PEGTT$ evaluation function for the table $tb$. For each $w_i$, $u_{w_i} \in [1, p-1]$ denotes permutation position that was revealed to the function and $\omega_i = (u_{w_i} + \lambda_i) mod\ p$. Given the table $tb$ and input encodings $\{k_{w_i} : i \in [0, \ell]\}$, the function outputs $m_{\omega_1,\ldots\omega_\ell} || m_{\omega_1,\ldots\omega_\ell}.val \oplus \lambda_O$.

We now show how to use these functions by giving an example to generate a $PEGTT$ for and-gate and evaluate the table. Assume there are two input wires and one output wire with two encodings for each of them, $k_0^0, k_0^1, k_1^0, k_1^1, k_2^0, k_2^1$. The permutation numbers are 1, 0, and 1. We create the and-gate $PEGTT$ $tb_{and}$ (see Figure 5) by using function $PEGTTGen(1, 0;\, k_0^0,\, k_0^1,\, k_1^0,\, k_1^1;\, 1;\, k_2^0,\, k_2^1)$.

| $w_0$ | $w_1$ | $w_2(w_0 and w_1)$ |
|---|---|---|
| 0 | 0 | $Enc_{k_0^1 \oplus k_1^0}(k_2^0 \| 1)$ |
| 0 | 1 | $Enc_{k_0^1 \oplus k_1^1}(k_2^1 \| 0)$ |
| 1 | 0 | $Enc_{k_0^0 \oplus k_1^0}(k_2^0 \| 1)$ |
| 1 | 1 | $Enc_{k_0^0 \oplus k_1^1}(k_2^0 \| 1)$ |

Figure 5: $tb_{and}$

The $tb_{and}$ can be evaluated by using function $PEGTTEval$. To help clarify how this function works we give an example of $PEGTTEval(\ tb_{and};\ k_0^1||0,\ k_1^1||1)$. The evaluator first gets two permute numbers 0, 1, which can indicate the evaluator to lookup the row $01$. Then, the evaluator decrypts the message that he got in the first step by using the keys $k_0^1$, $k_1^1$ and returns $k_2^1||0$ which is the result.

### 3.4.2 Circuit generation function and evaluation function

Given a type of circuit and the encodings for the input and output wires, the circuit generation function returns a scrambled circuit that uses these encodings. We use $CircuitGen(\text{TYPE}; I_0, I_1 \ldots I_n; \lambda_{I_0} \ldots \lambda_{I_n}; OU_0, OU_1 \ldots OU_r; \lambda_{OU_0} \ldots \lambda_{OU_r})$ to denote the circuit generation function. The parameter $TYPE$ corresponds to a description of the circuit, such as "less than", "greater than", "equal". $\lambda$ denotes permutation number. At the beginning of the protocol, CircuitGen creates all gates $GT$ and intermediate wires $IN$. Then, it chooses encodings $k_{IN}$ for all $IN$ and permutation numbers. Then, it calls $PEGTTGen$ to generate a $PEGTT$ for each $GT$. Finally, the function returns the whole circuit that was generated.

Given a scrambled circuit and encodings for the circuit's input wires, the circuit evaluation function returns encodings for the output wires. It is represented as $CircuitEval(circuit; k_{i_0}||i_0.val \oplus \lambda_{i_0}, \ldots k_{i_n}||i_n.val \oplus \lambda_{i_n})$ where $i_0, \ldots i_n$ are the input encodings. These input encodings should follow the patterns that were determined before the table had been generated.

## 4 New tool: Chained PEGTT

In this section, we introduce a variation of PEGTT to deal with a circumstance that the server and the client engage in $n$ PEGTTs where the later PEGTT uses the same encodings as the previous one but has an extra encoding. That is the server inputs $\{(\lambda, e_0^i, e_1^i) : i \in [1,1]\}$, $\{(\lambda, e_0^i, e_1^i) : i \in [1,2]\}$,..., $\{(\lambda, e_0^i, e_1^i) : i \in [1,n]\}$ and the message sets $M$, which are $[M^1 : \{0,1\}^1]$, $[M^2 : \{0,1\}^2]$,..., $[M^n : \{0,1\}^n]$; the client inputs $\{(\lambda \oplus v_i, e_{v_i}^i) : i \in [1,1]\}$,..., $\{(\lambda \oplus v_i, e_{v_i}^i) : i \in [1,n]\}$ to get the corresponding messages. Another requirement is the server must generate all look-up tables before it interacts with the client.

By using the protocol form the subsection 3.4.1 to deal with $n$ successive tables, the server's computational cost is $O(n2^n)$ and the client would have to perform $O(n^2)$ operations. However, by using the Chained PEGTT which is the new tool in this section, the server's computational cost can be reduced to $O(2^n)$ and the client's computational cost can be reduced to $O(n)$. We will give the full protocol and a brief analysis after the protocol is given. The main idea of Chained PEGTT is the server generates a $n+1$ set of keys $K = \{K_0, K_1, \ldots, K_n\}$, where $K_j = \{k_i^j : i \in \{0,1\}^j\}$ and then concatenates each $k_i^j$ with $M_i^j$ to generate the new messages $\hat{M}_i^j = M_i^j||k_i^j$. Also, the messages $\hat{M}_{i0}^{j+1}$ and $\hat{M}_{i1}^{j+1}$ have been encrypted by key $k_i^j$ with the appropriate encodings. After look up the $jth$ Chained PEGTT, the client can learn $\hat{M}_i^j$ which includes $k_i^j$. The key thing in here is $k_i^j$ is a compressed form of the encodings $e_{i(1)}^1, \ldots, e_{i(j)}^j$ in that the client will be able to learn $k_i^j$ if and only if it has $e_{i(1)}^1, \ldots, e_{i(j)}^j$. The client can do a single pseudorandom function evaluation in the $(j+1)$th table instead of doing the $j$ evaluations. Following the standard definition in [13], we have the pseudorandom function's definition " Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We say that $F$ is a $pseudorandom function$ if for all probabilistic polynomial-time distinguishers $D$, there exists a negligible function $negl$ such that: $|Pr[D^{F_k(\cdot)}(1^n) = 1] - Pr[D^{f(\cdot)}(1^n) = 1]| \leq negl(n)$ where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and $f$ is chosen uniformly at random from the set of functions mapping n-bit strings to n-bit strings."

In figure 6, we show the protocol to generate the Chained PEGTT. The input is encodings and permuted numbers with messages. After running the protocol, the server can generate all $n$ lookup tables.

1. For $\ell = 0$ to $n$ create a key set $K_\ell = \{k_i^\ell : i \in \{0,1\}^\ell\}$ where each $k_i^\ell$ 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 $C_i^j = (r_i, F_{e^j_{i'_j}}(r_i) \oplus F_{k_h^{j-1}}(r_i) \oplus \hat{M}_i^j)$ 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 $k_\perp^0, T_1, \ldots, T_n$.

Figure 6: $GENTAB_n(\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}, 1^\kappa)$

We will give an example to clarify the steps. Assume $n = 2$ and the server has input $\{(\lambda_1, e_0^1, e_1^1)$, $(\lambda_2, e_0^2, e_1^2)$, $M_0$, $M_1$, $M_{00}$, $M_{01}$, $M_{10}$, $M_{11}\}$. Also, assume that $\lambda_1 = 0$ and $\lambda_2 = 1$. At first, 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 $K_2 = \{k_{00}^2, k_{01}^2, k_{10}^2, k_{11}^2\}$. Then, the server create two tables. The first one is an 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 will be the ordered set $\{(r_{00}, F_{k_0^1}(r_{00}) \oplus F_{e_1^2}(r_{00}) \oplus (M_{01}^2 || k_{01}^2)), (r_{01}, F_{k_0^1}(r_{01}) \oplus F_{e_0^2}(r_{01}) \oplus (M_{00}^2 || k_{00}^2))$, $(r_{10}, F_{k_1^1}(r_{10}) \oplus F_{e_1^2}(r_{10}) \oplus (M_{11}^2 || k_{11}^2)), (r_{11}, F_{k_1^1}(r_{11}) \oplus F_{e_0^2}(r_{11}) \oplus (M_{10}^2 || k_{10}^2))\}$ (notice that the order has been permuted here, because $\lambda_2 = 1$).

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 messages. In Figure 7 we describe the details of the protocol for the $j$th 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 $\hat{M}_\ell = D_\ell \oplus F_{k^{j-1}_{v_1 \cdots v_{j-1}}}(r_\ell) \oplus F_{e^j_{v_j}}(r_\ell)$. Parse $\hat{M}_\ell$ into $M^j_{v_1 \ldots v_j}$ and $k^j_{v_1 \ldots v_j}$ and return these values.

Figure 7: $LOOKUP_j(T_j, k^{j-1}_{v_1 \cdots v_{j-1}}, \{(v_\ell \oplus \lambda_\ell, e^\ell_{v_\ell}) : \ell \in [1, j]\})$

Retuning to our example, suppose that $v = 01$, and thus the client should obtain $M^1_0$ and $M^2_{01}$ from the first and second table lookup respectively. In the first table lookup the client has $k^0_\perp$, $v_1 \oplus \lambda_1 = 0$, and $e^1_{v_1} = e^1_0$. The client takes entry 0 in $T_1$ (i.e., $(r_0, F_{k^0_\perp}(r_0) \oplus F_{e^1_0}(r_0) \oplus (M^1_0||k^1_0)))$ and computes $M^1_0||k^1_0$, which is the correct message. Now in the second table lookup the client uses $k^1_0$ and $v_2 \oplus \lambda_2 = 0$ and $e^2_{v_2} = e^2_1$ to decrypt entry $00$ in the table $T_2$. That is, the client decrypts $(r_{00}, F_{k^1_0}(r_{00}) \oplus F_{e^2_1}(r_{00}) \oplus (M^2_{01}||k^2_{01}))$ to obtain $M^2_{01}||k^2_{01}$, which is what is expected.

In Chained PEGTT server needs to perform only $O(1)$ PRF 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.

We will give a full proof of security for this primitive in the section 6.2.

## 5 Private Database Search Protocol

Consider the following problem: A client has a value $x \in \{0,1\}^b$ and a server has a set $S = \{s_1, \ldots, s_n\}$ where each $s_i \in \{0,1\}^b$. The goal of the protocol is for the client to learn whether $x \in S$. Furthermore, the server should not learn information about x, except whether $x \in S$ or $x \notin S$.

In this protocol, we assume that the server knows $S$ ahead of time and has the ability to do some precomputation before the query from the client. Also, we assume that the server can send one message to client before the query is available. Perhaps the server does precomputation for many protocols and sends the information to the client on a media such as a DVD. The naive solution for this problem is to create a comparison circuit for each record and then do a linear search over all circuits during the query time. The goal of the our protocol is to minimize the amount of time required to run the protocol after this precomputation.

Before describing the detailed protocol, we describe the intuition behind the protocol. The first step is to create a binary search tree whose leaf nodes store the values of the set $S$. The client and server engage in a standard search on this tree for the client's value. The difficulty is that the scheme must hide the search path from both the server and the client, because revealing this path to the server would reveal an interval that contains the query and revealing this path to the client would reveal the rank to the client. These pieces of information are not revealed by the result alone, and hence are extra information. At each level of the tree, the client and the server use a scrambled circuit to compute an intermediate result. Specifically at non-leaf level the circuit is a comparison circuit and at the leaf level it is an equality circuit. The client's input encodings are the same for each of these circuits, because the client's value in the search does not change. However, the encodings for the server's input depend upon the search path. For each level of the tree, the server stores all its input encodings into one PEGTT during the precomputation phase. In order to hide the searching path from the server, the server should not access any wire's encodings during the query phase. Hence the server sends all PEGTTs to the client before the query. On the other side, we can use the PEGTTs and circuits to hide real values and positions from the client.

One example has been showed in the figure 8. Assume the server has inputs 3, 7, 14, 22, 39, 43, 48, 51 and generates a binary search tree based on that. The permutation numbers are 1, 0, 0. The client's input is 22. For each level of the tree, the client use his input encoding compare with the server's input encoding getting from the previous PEGTT. It's a binary tree searching(the searching path has been showed in the figure), but encodings and permutation numbers help us hide a lot of information. Due to the full protocol is given next, we do not explain all the details in here.

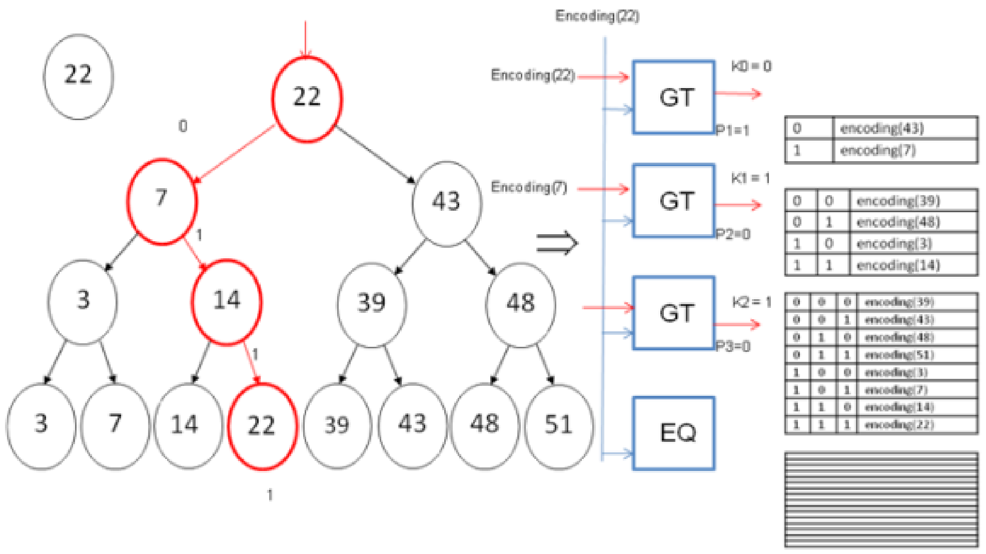

Figure 8: Example of our protocol

## 5.1 Protocol

The server inputs $[S : \{0,1\}^n]$ where each $s_i \in \{0,1\}^b$. On the other hand, the client inputs $x$. At the end of the protocol, the server outputs null and the client outputs 1 if $x \in S$, 0 otherwise.

**Precomputation phase:**

1. The server builds a binary search tree $T$ over the set S = $\{s_1, \ldots, s_n\}$, assuming $n = 2^h$. Let $t$ denote the root of tree $T$, and define $t_{\ell_1 \ldots \ell_k 0}$ to be the left child of $t_{\ell_1 \ldots \ell_k}$, and $t_{\ell_1 \ldots \ell_k 1}$ to be the right child. Also define $v_{\ell_1 \ldots \ell_k}$ to be the value at node $t_{\ell_1 \ldots \ell_k}$.

2.For $0 \leq i \leq h$ and $0 \leq j \leq b$, the server generates circuits $c_i$ and random bit strings $\delta_{e_j} \in \{0,1\}^{2\rho+1}$, $\delta_{c_i.g_j} \in \{0,1\}^{2\rho+1}$, and $\delta_{c_i.ou} \in \{0,1\}^{2\rho+1}$. Let $\delta.\lambda$ denote $\delta^{(2\rho)}$ and $\delta[i]$ denote $\delta^{[i,\rho]}$ where $i \in \{0,1\}$.

3.For each non-leaf level, $i$, of $T$, the server generates a less-than circuit using encodings and permutations from previous step. That is, for $i \in [0,h)$, $c_i$ = CircuitGen$(LT, \delta_{c_i.g_0}, \ldots, \delta_{c_i.g_b}, \delta_{e_0}, \ldots, \delta_{e_b}, \delta_{c_i.ou})$.

4.For the leaf level of $T$, the server creates an equality circuit using encodings/permutations from previous step, that is $c_h = CircuitGen(EQ,\, \delta_{c_h.g_0}, \ldots, \delta_{c_h.g_b},\, \delta_{e_0}, \ldots, \delta_{e_b},\, \delta_{c_h.ou})$.

5. For each non-leaf level, $i$, of $T$, the server generates transition tables, $TransT_i$, which allow the client to obtain the encodings of the server's inputs for the next circuit. Let $M_k \overset{\text{def}}{=} \{\delta_{c_k.g_0}[v^{(0)}_{\ell_1\ldots\ell_k}], \ldots, \delta_{c_k.g_b}[v^{(b)}_{\ell_1\ldots\ell_k}]\}$ denote the encoding array of the server's input at level $k$. The server encrypts the encodings with permutation positions by using all encodings on the search path. For level $i \in [1, h]$, the server calls $PEGTTGen(\delta_{c_0.ou}.\lambda, \ldots\, \delta_{c_{i-1}.ou}.\lambda, \delta_{c_0.ou}[0],\, \delta_{c_0.ou}[1], \ldots\, \delta_{c_{i-1}.ou}[0], \delta_{c_{i-1}.ou}[1],\, [M_i : \{0,1\}^i])$ to generate $TransT_i$.

6. The precomputation message is all transition tables $TransT$ and the circuits $C$. Also, the server sends its first circuit input encodings with permutation positions to the client, which is $\delta_{c_0.g_0}[v^{(0)}]||v^{(0)} \oplus \delta_{c_0.g_0}.\lambda, \ldots, \delta_{c_0.g_b}[v^{(b)}]||v^{(b)} \oplus \delta_{c_0.g_b}.\lambda$. As a short notation, we use $sq_i^j$ to denote the server's input encoding for the $jth$ wire in the circuit $i$ with permutation position during query phase. For example, the former formula can be written as $sq_0^0, \ldots, sq_0^{b-1}$.

**Query phase:**

1. The client gets encodings of its input by using $OT_2^1$. For each wire $e_i \in E$, the two parties run the protocol $OT(\delta_{e_i}[0]||0 \oplus \delta_{e_i}.\lambda,\, \delta_{e_i}[1]||1 \oplus \delta_{e_i}.\lambda;\, x^{(i)})$. The evaluator will get the $\{\delta_{e_i}[x^{(i)}]||0 \oplus \delta_{e_i}.\lambda : i \in [0, b-1]\}$ at the end of the protocol. As a short notation, let $cq^j$ denote the client's input encoding at the $jth$ wire concatenating with permutation position during the query phase, and also $oq_i$ denote the $c_i$'s output encoding concatenating with permutation position during the query phase.

2. For $0 \leq i < h$, the client calls the function $CircuitEval(c_i,\, sq_i^0, \ldots, sq_i^{b-1},\, cq^0, \ldots, cq^{b-1})$ to get output $oq_i$. Then, the client calls $PEGTTEval(TransT_{i+1}, oq_0, \ldots, oq_i)$ to get the server's input for $c_{i+1}$, which is $\{sq_{i+1}^0, \ldots, sq_{i+1}^{b-1}\}$.

3. For the level $h$, the client calls $CircuitEval(c_h,\, sq_h^0, \ldots, sq_h^{b-1},\, cq^0, \ldots, cq^{b-1})$. The output of the last circuit will be the final result.

Now we do a complexity analysis for our protocol. Recall that we have $n$ values in the dataset

$S$, and the number of bits for each value is $b$. For the server's operations in the precomputation phase, we have $O(\log n)$ circuits each of size $O(b)$ and $O(n)$ $PEGTT$ tables where each entry is of size $O(b)$. So in the precomputation phase, the computational complexity is $O(nb)$ and communication complexity is $O(nb)$. In the query phase, our protocol has $O(b)$ communication complexity and $O(b)$ modular exponentiation for the oblivious transfer. The computational complexity is $O(b \log n)$ by using binary search tree. The number of rounds is $O(1)$. Clearly, comparing with the runtime linear searching model, there is not big increase of the cost during the precomputation phase. However, the computational time for the client during the query phase has been reduce into sublinear and this is a significant improvement.

## 5.2 Other Problems

This protocol can be modified to solve some other interesting problems. In this subsection, we will define the problems and give some details to solve it based on the protocol in the subsection 5.1.

1. **Message lookup:** A client has a search term and a server has a set of (term, message) pairs. The intent of this protocol is that the client learns the message corresponding with his term. More formally, the client has a value $x \in \{0,1\}^b$ and the server has a set $S$ = ($s_1$, $m_1$), $\ldots$, ($s_n$, $m_n$) where each $s_i \in \{0,1\}^b$, and $m_i$ is plaintext. At the end of the protocol, the client gets $m_i$ if $x = s_i$, and gets nothing otherwise. Let $MG$ to denote the set of server's plaintext messages with $\perp$ in between, which is $\perp$, $m_1$, $\perp$, $m_2$, $\perp$, $\ldots$, $\perp$, $m_n$. The modification of this variation is to add the $PEGTT_{h+1}$ after the step 5 in the precomputation phase. That is, the server calls $PEGTTGen(\delta_{c_0.ou}.\lambda, \ldots\ \delta_{c_h.ou}.\lambda, \delta_{c_0.ou}[0], \delta_{c_0.ou}[1], \ldots\ \delta_{c_h.ou}[0], \delta_{c_h.ou}[1], MG)$.

2. **The rank of query:** A server has many values and a client wants to learn the rank of his query. More formally, the client has a value $x \in \{0,1\}^b$ and the server has a set $S = \{s_1, \ldots, s_n\}$ where each $s_i \in \{0,1\}^b$. The goal of the protocol is, at the end of the protocol, the client learns the rank of $x$ in $S$. The high level idea is that the server first sorts all his inputs and pads a $\infty$ after all the numbers. Then, the server ranks all the numbers and duplicate the ranks to get $R$. For

example, if the rank is 1, 2, 3, 4, $R$ would be 1, 1, 2, 2, 3, 3, 4, 4. This problem can be solved similarly as how we did in variation 1. At the level $h+1$, the server calls $PEGTTGen(\delta_{c_0.ou}.\lambda$,... $\delta_{c_h.ou}.\lambda$,$\delta_{c_0.ou}[0]$, $\delta_{c_0.ou}[1]$,... $\delta_{c_h.ou}[0]$,$\delta_{c_h.ou}[1]$, $R)$.

3. **One dimensional query:** A client has two values and a server has a set of values. The client wants to know how many values are between his two values in the server's set. More formally, the client has values $x \in \{0,1\}^b$ and $y \in \{0,1\}^b$, and the server has a order set $S = \{s_1, \ldots, s_n\}$ where each $s_i \in \{0,1\}^b$. Assume $s_{i-1} < x \leq s_i$ and $s_{j-1} < y \leq s_j$, the output is $|i-j|$. For this problem, the server first generate two set of encodings to represent $i$ and $j$. Let $R1$ and $R2$ denote two rank encoding sets for $S$. The server runs the protocol which is deal with the ranking problem two times. The only difference is that, instead of using $R$, the server encodes $R1$ and $R2$ at the level $h+1$. Then, the server should generate a minus circuit according to $R1$ and $R2$'s encoding representations. During the evaluation time, the client executes the rank search query two times and then pass the result encodings to the minus circuit to get the result.

4. **Two dimensional range query:** The server has a set $S \in \{\mathbb{Z} * \mathbb{Z}\}^n$ , and the client has two points $P(a,b)$ and $Q(c,d)$ in $\mathbb{Z} * \mathbb{Z}$. The client wants to know how many points in the rectangle range between $P$ and $Q$. We initially focus on how to get the number of points between (0, 0) and one input point $q(a,b)$ first, because if we can do this it can be easily changed to support queries between two points. As an example of this see Figure 9. Assume the client has two points $(a,b)$ and $(c,d)$. The client can respectively get the number of points between $(0,0)$ and $(a,b)$, $(0,0)$ and $(c,b)$, $(0,0)$ and $(a,d)$, $(0,0)$ and $(c,d)$ first. We use the notation $A$, $B$, $C$, and $D$ to denote these numbers of points. Then, the result will be $D$ - $C$ - $B$ + $A$.

Now we give a high-level protocol for two dimensional range query:

(1) The server divides the two dimensional coordinate into $\sqrt{n} * \sqrt{n}$ parts (boxes), where each chunk has $\sqrt{n}$ points. We define "box" as a rectangle part in the coordinate and "chunk" as a bunch of boxes have the same x-coordinate or y-coordinate(see Figure 10). We use notations $C_x$ and $C_y$ to denote sets of encodings for chunk in x-coordinate and y-coordinate. The server assigns each

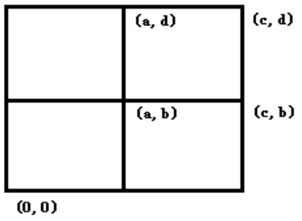

Figure 9: Two dimensional range query

box $b$ an unique number and generates an encoding $b.id$ for $b$. For each point $p \in S$, the server generates encodings $p.POS$ to represent its position. We use the notation $p.x$ and $p.y$ to denote x-coordinate and y-coordinate's value respectively.

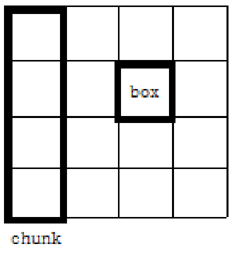

Figure 10: Box and chunk

(2)The server generates two binary search trees $T_x$ and $T_y$ based on the $C_x$ and $C_y$ by using our protocol.

(3)We use $b.num$ to denote the number of points less than or equal to the box $b$. The server generates a two dimensional permute encryption table $T_s$. For point $p$ in box $b$, the server encrypts $b.num$ by using $C_x^{p.x}$ and $C_y^{p.y}$.

(4)If $p$ in box $b$, we use notation $p.box$ to denote $b.id$. For all points $p$ in chunks $C_x$ and $C_y$, the server creates sets $C_x.T$ and $C_y.T$ where each element has tuple $p.box, p.POS$

(5)The server send $T_x$, $T_y$, $T_s$, $C_x.T$ and $C_y.T$ to the client.

(6)Assume the client inputs $q(a, b)$. He does $OT$ with the server to get $q.box$ and encodings of $a$ and $b$.

(7)The client respectively searches $T_x$ and $T_y$ and then gets two encodings $K_x$ and $K_y$

(8)The client searches $T_s$ by using $K_x$ and $K_y$. At the end of this step, the client gets the value $V_s$ which represents the number of points smaller than the box of $p$. Also, the client gets $2\sqrt{n}$ encoding tuples $C_x.T[p.chunk]$ and $C_y.T[p.chunk]$.

(9)For x coordinate, the client passes encodings of its input and $C_x.T[p.chunk].point.POS$ to $\sqrt{n}$ comparison circuits.

(10)For y coordinate, the client does the same operation. In addition, the client passes $q.box$ with $C_y.T[p.chunk].point.box$ to a not-equal circuit in order to ignore the overlap between the two chunks.

In this protocol, the work of the client will be reduced to $O(\sqrt{n}(b + \log n))$. If we do the first three steps over $C_x$ and $C_y$ recursively, the running time will be $O(\log n \log\log n)$, and the space cost will be $O(n \log n \log\log n)$.

5. **Server with small changes:** The current solutions only work for static datasets, because the server is not involved at the query phase. It is possible to extend the protocols to support datasets with a small number of changes. Let's reconsider the "existence problem" before. Assume the server already generated the data $D$ and sent it to the client. Then, the server makes small changes to the dataset, such as removing the data set $R$ or inserting the data set $I$. The server generates only scrambled circuits for the new data instead of generating a new binary search tree for the whole dataset. At the query phase, the client evaluates the original dataset by using initial protocol and then does linear computation for the new scrambled circuits. If the query is in the set of $R$ the client removes it from the result. If the query in the set of $I$ then the client adds it to the result. Assume there are $|R| + |I| \leq \epsilon$ changes from the server, the computational complexity of new circuits at the query phase will be linear of $\epsilon$. Hence, the computational complexity is $O(b \log n + \epsilon b)$ which is $O(b \log n)$ if the $\epsilon$ is small.

## 6 Security Analysis

In this section, we will prove that our protocol is secure in the honest-but-curious adversary model. First, we give the definition of some terminology that we use in this section. We define a function $f$ is negligible, if for all polynomial algorithm $p$ there exists a large number $N$ and for all $n > N$, it holds $f(n) < 1/p(n)$. Let $X = \{X_i\}_{i\in N}$ and $Y = \{Y_i\}_{i\in N}$ be distribution ensembles. We say $X$ and $Y$ are computationally indistinguishable, denoted $X \overset{c}{\equiv} Y$, if for every probabilistic polynomial time distinguisher $D$ and a significantly large $n \in N$, $|Pr[D(X_n) = 1]$ - $Pr[D(Y_n) = 1]|$ is negligible in $|n|$. By following the standard definition, we use view$_i^{\Pi}(x; y)$ to denote the view of the party $i \in \{client, server\}$ when they execute the protocol $\Pi$ on the inputs of $x$ and $y$. (In the rest of this paper, we will use the short notation $c$ and $s$ to denote $client$ and $server$.) The notation view$_i^{\Pi}(x; y)$ ($i \in \{c, s\}$) is equal to $\{x, r_i, m_1^i, \ldots, m_t^i\}$, where $r_i$ is $i$'s random tape and $m_j^i$ is the $jth$ message received by the party $i$. We define a protocol $\Pi$ is secure between two parties if there exists a polytime simulators $Sim_i$ ($i \in \{c, s\}$), such that $\{Sim_i(x, f_i(x, y))\} \overset{c}{\equiv} \{$view$_i^{\Pi}(x; y)\}$, where $x$ and $y$ are those two parties' respective inputs.

### 6.1 Our protocol

Our protocol is composed of the subprotocols: oblivious transfer, scramble circuit evaluation, and PEGTT. Let $\hat{\Pi}$ denotes the protocol where oblivious transfer, scramble circuit evaluation, and PEGTT have been replaced with calls to a trusted third party (TTP). For example, when oblivious transfer happens, instead of engaging the OT protocol, the client and the server both send their inputs to the TTP who sends the outputs back to the chooser. We will show that there exists a simulator with our protocol's inputs is computationally indistinguishable with the view of $\hat{\Pi}$ for the client and the server. Then, we can prove our protocol is secure based on the composition theorem which states that if a protocol is secure when the individual protocols use a TTP then the protocol that results from replacing the TTP versions with secure versions is also secure[14].

The server's inputs of our protocol are a set $S = \{s_i \in \{0,1\}^b : i \in [1,n]\}$ and the output is $\perp$. The client's input is $x$ and the output is a boolean value $b$, such as whether $x \in S$. Now, our goal is to build a simulator $Sim_s$ where $\{Sim_s(S, \perp)\} \overset{c}{\equiv} \{\text{view}_s^{\hat{\Pi}}(x; S)\}$ and a simulator $Sim_c$ where $\{Sim_c(x, b)\} \overset{c}{\equiv} \{\text{view}_c^{\hat{\Pi}}(x; S)\}$.

```
1. Initialize encoding array $EN[h][\ell]$
2. For $j = 1$ to $h$ do the following steps:
    (a) For $i = 1$ to $\ell$, chooses $EN[j][i] \leftarrow \{0,1\}^\kappa$
3. Return $EN$.
```

Figure 11: $Sim_c(x, b)$

For the server side, the creation is trivial, because the server cannot get any information in our protocol. For the client side, assume the size of input bit is $\ell$. The client gets the client encodings $C$ ($|C|$ = $\ell$) and server encodings $S_1$ ($|S_1|$ = $\ell$) at the first level of the circuits. Then, the client gets a new output encodings by evaluating the circuit. For all level $i < h$, the client can get server's input encodings from PEGTT and output encodings from circuit. At the leaf level, the client can only get output encodings from the circuit. Therefore, $\text{view}_c^{\hat{\Pi}}(x; S)$ is an two dimensional array of encoding, which length is $\ell * 2h$. We create a $Sim_c$ that is given in Figure 11 (We assume that the size of server's inputs is known by the client). Clearly, all encodings in $\{Sim_c(x, b)\}$ and $\{\text{view}_c^{\hat{\Pi}}(x; S)\}$ are uniformly distribution and their size is equal. We can create such $Sim_i$ ($i \in \{c, s\}$), thus if all subprotocols in $\hat{\Pi}$ are secure, then so is our protocol.

## 6.2 Chained PEGTT

In this subsection, we give a formal proof of security of Chained PEGTT. The generator's inputs are a sequence of encoding with permuted numbers and messages, which are $\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}$ and the output is $\perp$. The evaluator's inputs include all input encodings with the permutation bit and the k value. Let $n$ denote the level of the tree. The evaluator's inputs are $\{v_i \oplus \lambda_i, e_{v_i}^i, k^{i-1} : i \in [1,n]\}$. The evaluator's output at level $i$ includes the output message $\hat{m}_i$ and a k value $k^i$, which is the same as the input k value at the next level. In order to prove the security, our goal is to create a simulator in the generator's side which can hold $Sim_{gen}(\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}, \perp) \overset{c}{\equiv} \text{view}_{gen}^{ChainedPEGTT}(\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}; \{v_i \oplus \lambda_i, e_{v_i}^i, k^{i-1} : i \in [1,n]\})$. Also, in the evaluator's side, we should create a simulator that $Sim_{eval}(\{v_i \oplus \lambda_i, e_{v_i}^i, k^{i-1}, \hat{m}_i : i \in [1,n]\}) \overset{c}{\equiv} \text{view}_{eval}^{ChainedPEGTT}(\{(\lambda_i, e_0^i, e_1^i) : i \in [1,n]\}, \{\{M_i^j : i \in \{0,1\}^j\} : j \in [1,n]\}; \{v_i \oplus \lambda_i, e_{v_i}^i, k^{i-1} : i \in [1,n]\})$.

```
1. For $j = 1$ to $n$, the simulator do the following steps:
    (a) Generate a number $i' \in \{0,1\}^j$ where $i'^{(i)} = v_i \oplus \lambda_i$ and $r_{i'} \leftarrow \{0,1\}^\kappa$.
        $C_{i'}^j = (r_{i'}, F_{k_{i-1}}(r_{i'}) \oplus F_{e_i}(r_{i'}) \oplus \hat{m}_i)$ where $F$ is a pseudorandom function mapping $\{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^{m+\kappa}$.
    (b) For all $i = \{0,1\}^j$ and $i \neq i'$, the simulator creates $r_i \leftarrow \{0,1\}^\kappa$ and and a random number $r \leftarrow \{0,1\}^{m+\kappa}$. Then, it computes $C_i^j = (r_i, r)$.
    (c) Create table $T_j = \{C_\ell^j : \ell \in \{0,1\}^j\}$.
2. Return the tables $T_1 \ldots, T_n$.
```

Figure 12: $Sim_{eval}(\{v_i \oplus \lambda_i, e_{v_i}^i, k^{i-1}, \hat{m}_i : i \in [1,n]\})$

For the generator's side, the creation is trivial. That's because the generator does not get information from the Chained PEGTT protocol. For the evaluator's side, we create a simulator $Sim_{eval}$ that outputs the lookup tables(see Figure 12). Although the view of evaluator includes the encodings, it is not important because the simulator's inputs includes all those information. Now, if we prove the tables that are created by $Sim_{eval}$ are computationally indistinguishable with the tables in real protocol's view, the protocol is secure.

Let $n$ denote the number of tables in the view of real protocol and $P_0$ denote the real protocol. In $P_0$, the evaluator knows $n$ of the $k$-values and encodings. For the further definition, let $P_i(i \in [0, M])$ denote the protocol where $i$ of the PRFs (for unknown encodings) have been replaced with a random function (each unique encoding uses a different random function). Let $M$ denote the number of encodings/keys that evaluator doesn't know in $P_0$. By using this definition, $P_M$ is a protocol that only uses random functions for unknown encodings.

We claim that $\text{view}_{eval}^{P_i} \overset{c}{\equiv} \text{view}_{eval}^{P_{i+1}}$. The only difference between these two protocol, is that a single PRF has been replaced with a random function. If these two distributions were distinguishable, then it would be possible to distinguish a PRF from a random function. That is if we are given oracle access to either a PRF or a random function, then we can generate either $\text{view}_{eval}^{P_i}$ or $\text{view}_{eval}^{P_{i+1}}$. Hence, if we could distinguish these two views, then we could distinguish the oracle as a random function of a PRF.

We also claim that $\text{view}_{eval}^{P_M} \overset{c}{\equiv} Sim_{eval}$. At places where the evaluator does not know the encodings, the PRFs in $P_M$ have been replaced with a random function. The tuples in the table corresponding to these entries in $P_M$ take the form $(r, f(r) \oplus X)$ where $r \leftarrow \{0,1\}^\kappa$ is a randomly chosen value, $f$ is a random function, and $X$ is some other value (it could be based on a a PRF or another random function). The probability that we have a protocol to distinguish $\text{view}_{eval}^{P_M}$ and $Sim_{eval}$ is $Pr[Dist] = Pr[Dist|Dup] \cdot Pr[Dup] + Pr[Dist|\overline{Dup}] \cdot Pr[\overline{Dup}] \leq Pr[Dup] + Pr[Dist|\overline{Dup}]$ where $Dup$ means duplication of $r$-value (the first term in the tuples) happens. Let $r_i$ and $r_j$ denote $r$-values where $i \neq j$, we can get $Pr[Dup] = \bigcup\limits_{i \neq j} Pr[r_i = r_j] \leq \sum\limits_{i \neq j} Pr[r_i = r_j]$. Since $Pr[r_i = r_j] = 1/2^\kappa$, $Pr[Dup] = n^2/2^\kappa$ where $n$ is in the order of entries. Our assumption is $n$ is a polynomial order in the security parameter $\kappa$. Clearly, $Pr[Dup]$ is negligible. According to the definition of the random function, if $r$-values in all of these tuples are unique, then $f(r)$ is a random value. It is known in the folklore that a random value XORed with anything is a random value. Hence if the $r$ values are all unique, $\text{view}_{eval}^{P_M}$ and $Sim_{eval}$ are identically distributed, which refers to $Pr[Dist|\overline{Dup}] = 1/2$. Therefore, $Pr[Dist] \leq Pr[Dup] + Pr[Dist|\overline{Dup}] = 1/2 + negl(n)$.

Combining these two claims and the fact that $M$ is polynomial in the size of the inputs, with a standard hybrid argument implies that $\text{view}_{eval}^{P_M} \overset{c}{\equiv} Sim_{eval}$.

## 7 Experiments

In this section, we present two experiments. One is the performance comparison between the naive scheme (see section 5) and our protocol in the same input bit size. The other is to show the performances of our protocol in different input bit size. 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. 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 $q \in S$.

For the first experiment, 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 each input number is 16.

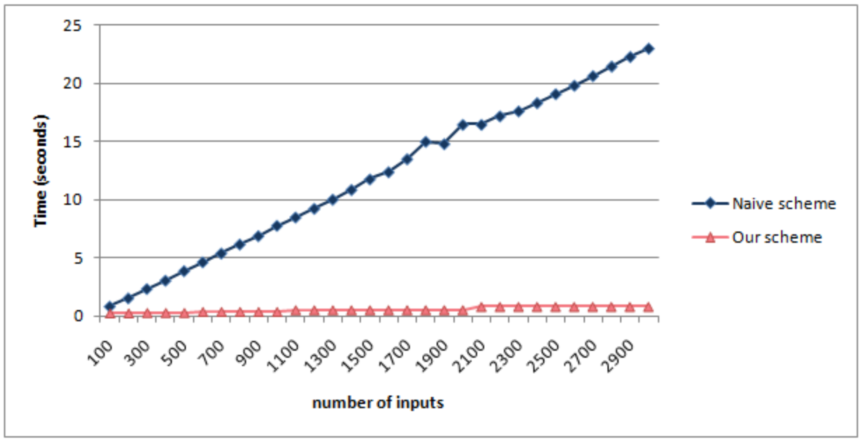

Figure 13: Precomputation time(Naive vs Ours)

**Precomputation time (cf. Figure** 13). 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.

**Communication size (cf. Figure** 14). Both schemes require linear communication size. The

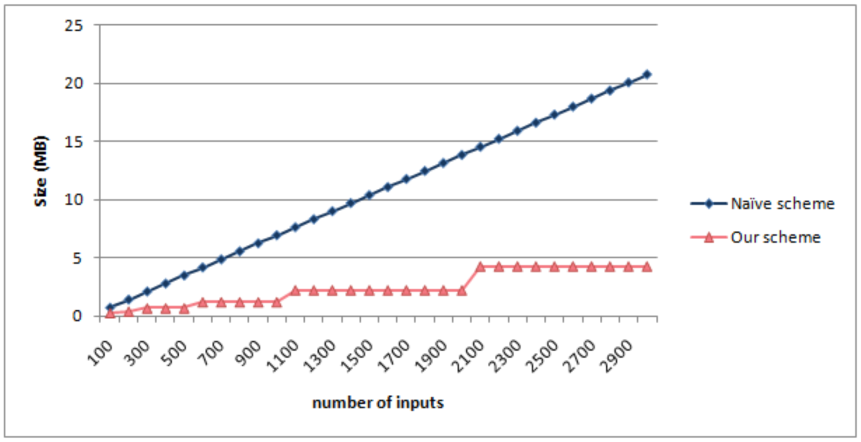

Figure 14: Communication size(Naive vs Ours)

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.

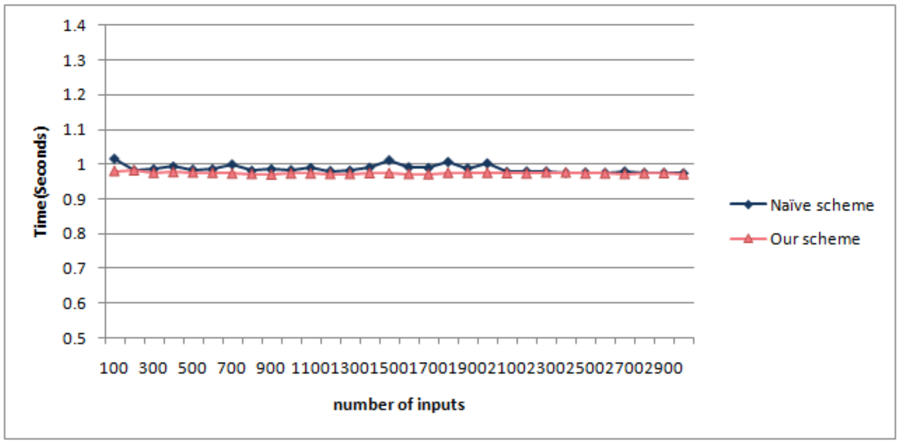

Figure 15: OT time(Naive vs Ours)

**OT time (cf. Figure** 15**).** There is no big difference between our scheme and naive scheme in OT time, because the client does same OT in both scheme for its input.

**Evaluation time (cf. Figure** 16**).** Our scheme significantly improves the performance in evaluation time. In the second experiment, the evaluation time for our solution is still under 0.004 seconds even server's input size increases to 500000. We expect the naive solution will spend 500 seconds if the number of inputs is 500000.

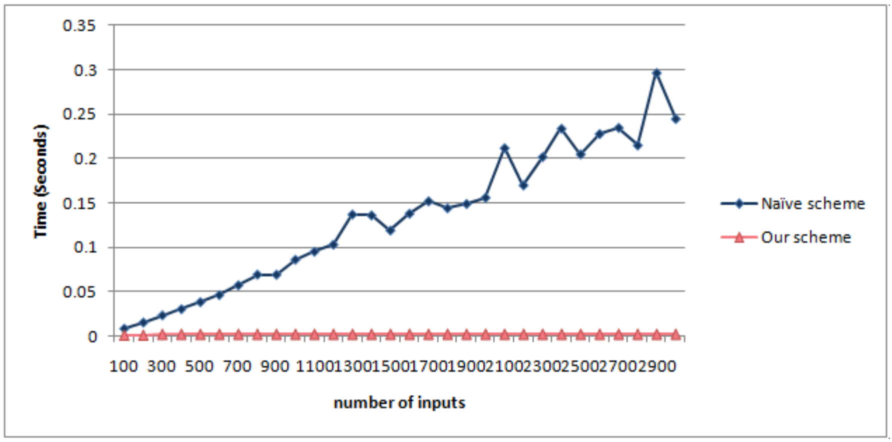

Figure 16: Evaluation time(Naive vs Ours)

In the second experiment, we varied server's input size form 5000 to 515000 in step of 10000. For each input size we run each experiment 20 times and report the mean performance. Since the 16 bits experiment consumes approximately double RAM space than 8 bits, it stopped when the number of input increased to 245000. The goal of this experiment is to show our scheme's performance in different input bit size(8 bits and 16 bits).

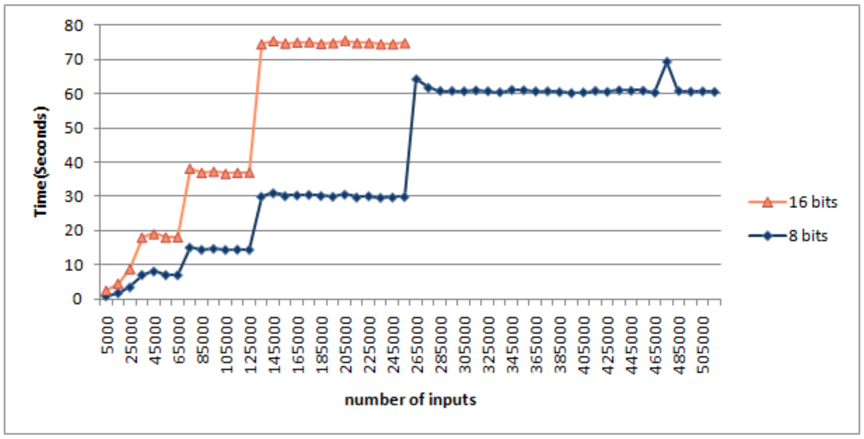

Figure 17: Precomputation time(8bits vs 16bits)

**Precomputation time (cf. Figure** 17). The performance shows that the 16-bit's precomputational time is little over bigger than the double of 8-bit's. That is because 16-bit not only double the gates but also some "extra" gates needed if the circuits became larger.

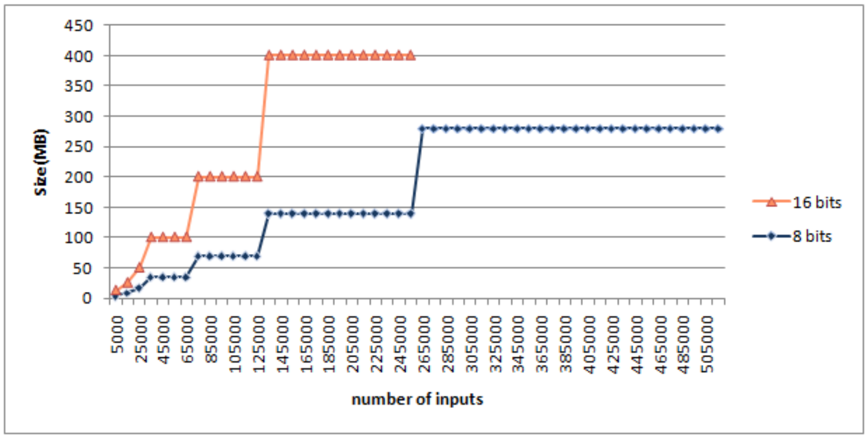

Figure 18: Communication size(8bits vs 16bits)

**Communication size (cf. Figure** 18). The same reason as in the precomputation time, 16-bit's communication size is little bigger than the double size of 8-bit's.

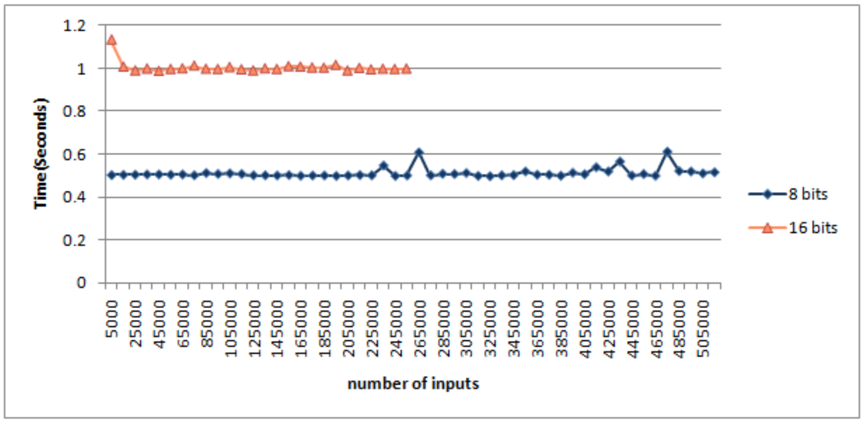

Figure 19: OT time(8bits vs 16bits)

**OT time (cf. Figure** 19). 16-bit's OT time is two times of the 8-bit's, because the work is doubled.

**Evaluation time (cf. Figure** 20). Our scheme performs well in large scale of the input number. The data has several jumps in the figure 20, that is because errors influence a lot if the time scale is only in milliseconds. Even this, our experiments still clearly shows the different performance between 8-bit and 16-bit.

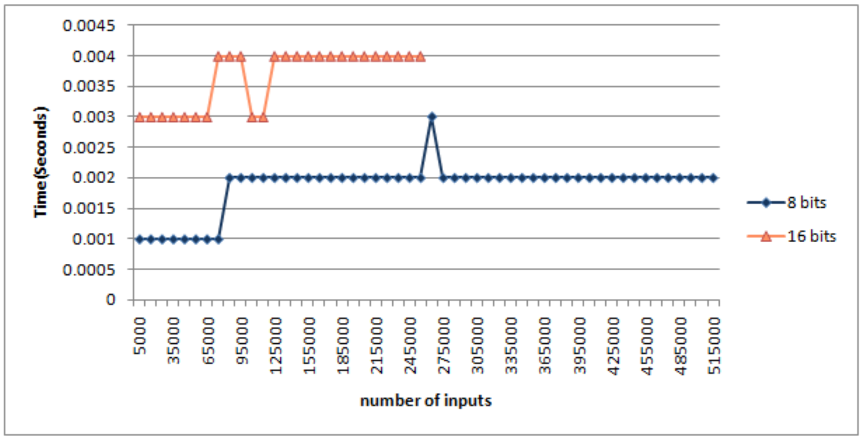

Figure 20: Evaluation time(8bits vs 16bits)

## 8 Summary/Future work

In this thesis, we present a private database search protocol in precomputational model which is allow the data owner can do precomputation at the head of the query time. Our goal is to minimize the query time to sublinear and let the precomputation time is still in linear. We also present several protocols to deal with various problems based on this model. Several interesting avenues for future work include:

1) Currently, the protocol only works for the honest but curious model. Extending the protocol to the malicious adversary model would be very interesting.

2) A limitation of the current approach is the precomputation process must be done for each query. It would be interesting if the precomputation information can be used in multiple queries.

## References

[1] A. C. Yao, How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162-167, 1986.

[2] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *Proc. 13th USENIX Security Symposium*, pages 287-302, 2004.

[3] Y. Lindell, B. Pinkas, A proof of security of Yaos protocol for two-party computation. In *Journal of Cryptology 22*, pages 161-188, 2009.

[4] W. Gasarch. A survey on private information retrieval. *Bulletin of the European Association for The European Association for Theoretical Computer Science,* 82, 2004

[5] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45:965-981, 1998. Earlier version in FOCS 95.

[6] R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2007.

[7] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*,May 2004.

[8] J. Garay, B. Schoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, PKC, volume 4450 of *Lecture Notes in Computer Science*, pages 330-342. Springer, 2007.

[9] H. Y. Lin, W. G. Tzeng. An efficient solution to the millionaires problem based on homomorphic encryption. *LLecture notes in Computer Science* 3531, Springer-Verlag, pages 456-466, 2005.

[10] M. O. Rabin. How to Exchange Secrets with Oblivious Transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[11] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *LProceedings of Crypto 89*, pages 547-557, 1990.

[12] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of SODA 01*, 2001.

[13] J. Katz and Y. Lindell. Introduction to modern cryptography. *Boca Raton: Chapman & Hall/CRC*, 2008.

[14] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. In *J. of cryptology*, Vol.13 No.1 pages 143-202, 2000.

[15] E. Kushilevitz and R. Ostrovsky, Replication Is Not Needed: Single Database, Computationally-Priuate Information Retried. In *38th FOCS*, pages 364-373, 1997.