Reducing Paging and Exit Overheads in Intel SGX for Oblivious Conjunctive Keyword Search

Qin Jiang, Saiyu Qi, Xu Yang, Yong Qi, Jianfeng Wang, Youshui Lu, Bochao An, Ee-Chien Chang

Abstract—Paging and exit overheads have been proven to be the performance bottlenecks when adopting Searchable Symmetric Encryption (SSE) with trusted hardware such as Intel SGX for keyword search. This problem becomes more serious when incorporating ORAM and SGX to design oblivious SSE schemes such as POSUP [1] and Oblidb [2] which can defend against inference attacks. The main reason comes from high round communication complexity of ORAM and constrained trusted memory created by SGX. To overcome this performance bottleneck, we propose a set of novel SSE constructions with realistic security/performance trade-offs. Our core idea is to encode the keyword-identifier pairs into a bloom filter to reduce the number of ORAM operations during the search procedure. Specifically, Construction 1 loads the bloom filter into the enclave sequentially, which outperforms about $1.7\times$ when the dataset is large compared with the performance of the baseline that directly combines ORAM and SGX. To further improve the performance of Construction 1, Construction 2 classifies keywords into groups and stores these groups in different bloom filters. By additionally leaking the keywords in search token belonging to which groups, Construction 2 outperforms Construction 1 by $16.5 \sim 36.8\times$ and provides an improvement of at least one order over state-of-the-art oblivious protocols.

Index Terms—Conjunctive keyword search, Bloom filter, ORAM, Intel SGX

1 Introduction

With the proliferation of cloud computing, users are increasingly interested in outsourcing personal and commercial data on the cloud. However, the cloud is untrusted, and storing the encrypted dataset on the cloud is a way to protect the privacy of the dataset. The challenge lies in processing the user's encrypted queries directly on the encrypted dataset in an efficient performance and secure way. Consider a client uploads an encrypted email dataset to the cloud. When the client sent an encrypted query of the from " retrieve all emails with the keywords 'name=Bob' and 'time =2022' and 'topic= research' ", the cloud should be able to perform this conjunctive query without leaking any sensitive information such as the addresses and contents of the emails [3]. To overcome this challenge, searchable encryption (SE) schemes [4]–[6] have been proposed to support privacy-preserving keyword search.

Unfortunately, most SSE schemes suffer access pattern leakages which can be abused by the attacker to infer plaintexts about the dataset or the queries [7], [8]. To mitigate such leakages, a general approach is to rely on oblivious RAM (ORAM). ORAM allows a client to read and write data from the server without revealing access patterns by shuffling and re-encrypting data in the dataset for every read/write operation. However, the cost of ORAM remains prohibitively expensive as every ORAM operation incurs a poly-logarithmic round communication overhead. Existing works [1], [2], [9]–[11] propose to use trusted execution environments (TEEs) such as Intel SGX [12] to reduce the communication overhead of the ORAM to improve search efficiency. These schemes regard the enclave created by SGX as an ORAM client and regard the untrusted memory of the cloud as an ORAM server. By relying on cloud-hosted close-to-data trusted hardware, the communication cost between the enclave and the untrusted memory in one server instead of the network can be reduced substantially. However, the polylogarithmic round complexity of ORAM operation (read/write) is not reduced, which can increase the number of enclave exits. Enclave exits contain page swaps, and each page swap is about 40K cycles. This is one of the reasons why ORAM operation conducted on the SGX-enabled server can incur significant performance degradation [11], [13]. In addition, the enclave storage is limited (128MB in our experiment). When the data structure within the enclave is larger than 96 MB, it can cause paging overhead which is leading to about $134\times$ slower than insecure runs for a 4GB data structure. To conduct an oblivious conjunctive keyword query [14] on an SGX-enabled server, the ORAM client firstly issues a read ORAM operation to the untrusted memory for each keyword in the query. These read ORAM operations get lists of the document identifiers for each keyword. Then the procedure within the enclave can get the intersection of these lists for the conjunctive query. This state of affairs raises the following question:

Is it possible to reduce the paging and exit overhead in SGX for conjunctive keyword search while preserving a practical level

Q. Jiang is with the School of Computer Science and Technology, Xi'an Jiaotong University, 710049, China and Engineering Research Center of Digital Forensics, Ministry of Education, Nanjing University of Information Science and Technology, 210044, China (e-mail: qjiang16@stu.xjtu.edu.cn).
S. Qi, X. Yang, Y. Qi and B. An are with the School of Computer Science and Technology, Xi'an Jiaotong University, 710049, China (e-mail: saiyu-qi@mail.xjtu.edu.cn; yangxu@stu.xjtu.edu.cn; qiy@xjtu.edu.cn; adolfhitler@stu.xjtu.edu.cn).
Y. Lu is with the School of Electrical Engineering, Xi'an Jiaotong University, 710049, China (e-mail: lucienlu@xjtu.edu.cn).
J. Wang is with School of Cyber Engineering, Xidian University, China (e-mail: jfwang@xidian.edu.cn).
E. C. Chang is with the School of Computing, National University of Singapore, Singapore (e-mail: changec@comp.nus.edu.sg).
This work was supported in part by the National Natural Science Foundation of China under Grant 62172328 and 62072357, in part by the National Key R&D program of China No.2021YFB2700900 (Corresponding author: Yong Qi).

TABLE 1: Comparison with related works

Scheme	Singly-oblivious	Doubly-oblivious	Query type	Query efficiency	Paging reduce
POSUP [1]	×	✓	*Sing.*	$\mathcal{O}(\log^2(m))$	×
Oblidb [2]	✓	×	*Sing. & Ran.*	$\mathcal{O}(\log^2(m))$	×
Oblix [9]	✓	✓	*Sing.*	$\mathcal{O}(\log^2(n))$	×
ShieldStore [11]	×	×	*Sing.*	$\mathcal{O}(1)$	✓
Construction 1	✓	×	*Conj.*	$\mathcal{O}(\log(m))$	✓
Construction 2	✓	×	*Conj.*	$\mathcal{O}(\log(m))$	✓

(†In this table, Singly-oblivious denotes that access pattern of untrusted memory is oblivious. Doubly-oblivious denotes that access patterns of enclave and untrusted memory are oblivious. *Query type* includes range query (Ran.), single keyword query (Sing.) and conjunctive query (Conj.). n denotes the number of documents in the dataset. m denotes the number of indexed values, $n \gg m$. The label ✓shows that the protocol can achieve the corresponding attribute in the column. Otherwise, it indicates ×.)

of security?

Our protocols in this work answer this question in confirmation. Specifically, our Construction 2 can significantly improve the efficiency of oblivious conjunctive keyword queries with acceptable leakage.

Overview of BXT. We first give the main idea of BXT [15] to clearly explain the core technical idea behind our protocols. Similar to general SSE schemes, BXT builds a data structure named $Tset$. Specifically, $Tset$ stores an encrypted inverted index that maps each keyword w to the identifiers of documents containing w. For simplicity, we name the set of identifiers as *ID* list. To reduce the communication overhead of conjunctive queries, an additional data structure named $Xset$ is built to store a set of elements named $Xtag$ computed from document identifier(id)-keyword pairs. For example, for each $w \in W$, $xtag = f(xtrap, id)$, where $id \in DB(w)$, id is the identifier of the document, $DB(w)$ is the *ID* list containing w, $xtrap$ is the search token of w and f is an unpredictable function. For a conjunctive search query $Q = \{w_1, w_2, ..., w_q\}$, the client sends search token $Q' = \{xtrap_1, xtrap_2, ..., xtrap_q\}$ to the cloud. BXT first gets $DB(w_1)$ from $Tset$ using $xtrap_1$ and then filters the identifiers containing other keywords in Q using $Xset$ and $DB(w_1)$. Specifically, for each id in $DB(w_1)$ and a keyword w_i, $(2 \leq i \leq q)$, BXT calculates $Xtag$ and checks whether $Xtag$ is in $Xset$ or not. For example, for query Q', the cloud firstly gets $DB(w_1)$ from $Tset$ and $DB(w_1) = \{id_3, id_5\}$. Then, the cloud checks whether id_3 is belonging to the result or not by the following steps: first, the cloud computes $xtag_i = f(xtrap_i, id_3)$, $i \in \{2, ..., q\}$. Second, the cloud checks whether $xtag_i$ is belonging to $Xset$. If for all $i \in \{2, ..., q\}$, $xtag_i \in Xset$, then id_3 is belonging to the result. Otherwise, id_3 is not belonging to the result. In addition, we can use a similar procedure to check whether id_5 is belonging to the result or not.

Inspired by BXT, we can load the data structure $Xset$ into the enclave to reduce the number of ORAM operations for keywords except for the first keyword in the search token. While it might appear that it is simple to reduce the number of ORAM operations, there exists a challenge that requires careful consideration.

Challenge. The challenge lies in the limited storage capability of enclave created by SGX. As the entire keywords in the large dataset are encoded into $Xset$, the size of $Xset$ is significantly large even though we use the bloom filter to store it. When we conduct a conjunctive keyword query using such a bloom filter, the frequency of context switches between the enclave and the untrusted memory has a significant impact on the response time of the query.

Contributions. To address the above challenge, we first provide two straw-man solutions for oblivious conjunctive keyword search to confirm the performance bottlenecks within SGX using a set of experiments. We observe that the paging overhead is the main reason for performance degradation when the size of $Xset$ exceeds the storage constraint of SGX. Based on these insights, we design two constructions for oblivious conjunctive keyword search. Specifically, Construction 1 firstly stores $Xset$ in a bloom filter and then divides the bloom filter into a number of splices and organizes these splices as a new data structure named hierarchical bloom filter HBF. When operating a conjunctive query, the procedure reads HBF into the enclave sequentially, which can reduce the paging overhead caused by the large size of the bloom filter. Our experiment shows that the search performance of Construction 1 can achieve about $1.7\times$ higher than the straw-man solution. To further improve the efficiency, our observation is that each keyword may have correlations with other keywords on the search token. This is meaning that keywords in one query are in one group. Based on this observation, we design Construction 2. In particular, we first classify the keywords contained in the dataset into several groups. Then for each group, we connect each keyword and its corresponding document identifier as a pair and store this pair in a bloom filter. We store these bloom filters into HBF. When receiving a search token, a problem for the cloud server is finding the groups containing keywords in the search token. To overcome this problem, we employ a data structure named bloom tree proposed in [16] into the enclave. Our experiments show that Construction 2 can improve search efficiency significantly. We give the comparison with related works in Table 1. In summary, we make the following contributions:

- We design two straw-man solutions to analyze the reasons for the performance degradation for oblivious conjunctive keyword search using SGX.
- We propose two constructions reducing the paging and exit overhead in SGX with efficient performance for oblivious conjunctive keyword search. Moreover,

we give possible directions to improve the security of our constructions.
- We implement the protocols and evaluate the search performance of our constructions. The experiment shows that Construction 2 outperforms Construction 1 by $16.5 \sim 36.8\times$ and outperforms previous works by more than one order of magnitude.

2 Preliminaries

2.1 Intel SGX

Intel SGX [12], [17] is Intel's instruction extension which aims to create an isolated and trusted area named enclave in the memory. Enclave can guarantee the security of sensitive computation and protect the integrity and confidentiality of the data. In this work, we use the remote attestation feature of SGX to create a secure channel between the client and the enclave of the cloud. Due to the protection mechanism of SGX, privileged softwares such as OS kernel, hypervisor and firmware cannot access the enclave memory directly. SGX allocates only 128MB of trusted space and 96MB of space available to users. Works in [11], [13], [18] show that the limited size of enclave can lead to significant performance degradation of the application when loading the entire data structure into the enclave. Meanwhile, the other research line [19]–[22] shows that adversaries can infer data within enclave by side-channel attacks such as changing the page table or observing data communication granular to a cache line. Access pattern leakages generated from access operations on untrusted storage can be used to infer valid information such as the search keywords or the plaintext of the encrypted dataset.

2.2 Path ORAM

Path ORAM is a tree-based ORAM [23], [24], a cryptographic protocol that consists of two components a client-side and a server-side. This protocol re-encrypts each data on the client side and confuses the storage location of the data on the server side for every read/write access. This property of ORAM can be used to implement oblivious data structures. In our scheme, we use the ORAM interface as a black box and the security of the ORAM is critical for our scheme. We denote ORAM's formal security definition as follows:

ORAM Security. Let $\vec{x} := ((op_1, a_1, data_1), ..., (op_M, a_M, data_M))$ denotes a data request sequence of length M, where op_i denotes a read or write operation, a_i denotes the address of the block being read or written and $data_i$ denotes the data being written.

Let$A(\vec{x})$ denotes the (possibly randomized) sequence of accesses to the untrusted storage given the sequence of data requests $\vec{x}$. An ORAM construction is said to be secure if:

For any two data request sequences $\vec{x}$ and $\vec{y}$ which have the same length, their access patterns $A(\vec{x})$ and $A(\vec{y})$ are computationally indistinguishable by anyone but the client ORAM controller.

2.3 Bloom filter and bloom tree

Bloom filter [25] is a m bit array which is used to answer a membership query [26], [27] such that whether a given item x is belonging to a set X.

For a set of n items, $X = \{x_0, x_1, ..., x_{n-1}\}$, the bloom filter B use a hash family $H = \{h_1, h_2, ..., h_k\}$ which contains k independent, uniformly random hash functions to encode the items contained in X into B. More specifically, $B[i]$ denotes the ith bit in the bloom filter B, where $0 \leq i \leq m-1$. To record an item x_r, each hash function maps x_r to B by setting B[h_j(x_r)]=1 for $1 \leq j \leq k$. To answer a membership query for x_r, if $B[h_j(x_r)] = 1$, for all $1 \leq j \leq k$, the bloom filter gives a positive response.

Although it is easy to check whether an item belongs to a bloom filter or not, it may return a false positive response which means that $x \notin X$, but the filter erroneously gives a positive answer. The probability of the false positive, denoted as fp, can be computed as

$$fp = (1 - (1 - \frac{1}{m})^{n\times k}))^k \approx (1 - e^{-\frac{n\times k}{m}})^k \quad (1)$$

we can minimize the false positive probability by setting the number of hash functions

$$k = \frac{m}{n} \times ln2 \quad (2)$$

Bloom tree proposed by Yoon et al. [16] is a compact and randomized data structure used for multiple-set membership testing. Multiple-set membership testing [26] is a process for checking an item in which set in a deterministic time. Bloom tree [16], [28] uses the memory space efficiently and requires fewer memory accesses, which has a significant advantage for the size-limited on-chip memory. For instance, given a bloom tree which is a d-arry complete search tree, the nodes of the bloom tree are bloom filters. When building the bloom tree, we encode the keys of items into the bloom filters from the root to the leaf, which denotes the value by way of using the set of hash functions. As in [16], each set of hash functions can determine the edge from the current node to its child node when checking the value of an item. Given a queried keyword, the bloom filter can return its corresponding group identifier when the hash functions from the root to leaf satisfy $H_j^i(y) := 1$ where $j \in [1, d]$ and i is the level of the accessed node in bloom tree and d denotes the number of distinct hash function sets of the accessed node.

3 System overview

3.1 System model

The high-level design of our solution is shown in Fig. 1. Our system consists of two parts: the client and the SGX-enabled server. There are two phases in the high-level design: setup phase and query phase. The details of these phases are shown as follows:

Setup phase. Initially, a secure connection needs to be established between the user and the enclave of the cloud using the remote attestation of SGX. Then, the data user possesses a dataset which is a collection of n documents. Index structures such as inverted index, bloom tree, *HBF* are generated from these documents. Then the data user stores the inverted index in ORAM data structure and encrypts the documents using the symmetric encryption scheme. Finally, the data user uploads these index structures and encrypted documents to the SGX-enabled cloud.

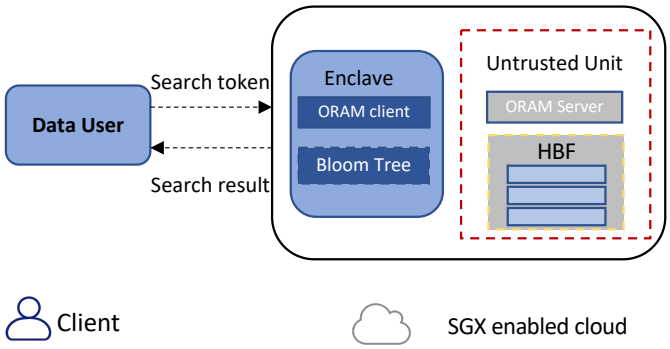


Fig. 1: The diagram of high-level design

Query phase. When the client issues a search token to the cloud by the secure connection, the search procedure does the following steps: first, when the enclave receives the search token, the procedure within the enclave decrypts the search token and parses the keywords. Second, the procedure uses the read operation of ORAM to read the encrypted ID list of the least frequency keyword from $Tset$. The keyword with the least frequency of occurrence can be known by the general statistics or at the cost of small additional storage. Third, the procedure finds out the search result by filtering the ID list within the enclave and uses *HBF* and bloom tree in our constructions respectively.

3.2 Threat model

We assume that our system provides SGX and an attacker is with full control of the cloud. More specifically, we assume that the adversary can observe memory addresses on the memory bus and data on the untrusted memory. In our scheme, we focus on reducing the access pattern leakages of data stored on untrusted memory. The security strength is the same as Oblidb [2]. While the access pattern to data and code within enclave by side-channels such as page faults and branching history can be abused by the adversary, there are many works [1], [9] to mend these attacks. Furthermore, we assume the security of the enclave and a secure channel created by the remote attestation of SGX which can be used to communicate messages securely between the client and the cloud.

4 Straw-man Constructions

In this section, we begin with two straw-man constructions for oblivious conjunctive keyword search, from which we derive the reasons for performance bottlenecks of oblivious conjunctive keyword search on SGX-enabled cloud.

4.1 Straw-man Construction 1

A naive approach is to get the ID lists of the keywords using ORAM operations from the encrypted inverted index and interact these ID lists within the enclave to get the search results. More specifically, we first construct an inverted index from the datasets and then use path ORAM to store the inverted index. We put the path ORAM server on the untrusted server and the ORAM client within the enclave. In addition, we construct a hash table within the enclave to map the keywords to the identifiers of the blocks on the ORAM server side. When receiving the search request, the codes within the enclave decrypt the search token and then parse the plaintext to get the keywords. The codes invoke the read operation of path ORAM for each keyword to get the corresponding ID list in the encrypted inverted index. When finished searching all keywords, we intersect these ID lists to get the results. In addition, we call this straw-man solution **NoBF**.

Performance analysis. The query cost of NoBF is linear to the number of keywords contained in the search token. For each keyword, we need an expensive ORAM read operation, which causes frequent communications between the enclave and the untrusted memory. As the number of keywords contained in the search token increases and the size of datasets enlarges, the query cost becomes significantly expensive.

4.2 Straw-man Construction 2

NoBF uses expensive path ORAM operations, the roundtrip communication between the enclave and the untrusted part will delay the query response time. Inspired by *BXT* proposed by Cash et al. [15], we add data structure Xset and load Xset to the enclave to improve the query performance. Similar to *BXT*, we built *Xset* by adding *xtag*. More specifically, for each keyword, we define *xtag* as a string, which contacts the keyword with ID. As the space of enclave is limited, we utilize bloom filter to store *Xset*. Specifically, we use a family of hash functions $H = \{h_1, h_2, ..., h_m\}$, such that each function maps *xtag* to the bloom filter. The index data structures needed to build is shown in Algorithm 1. When receiving the search token, the steps of the search procedure on the server are described as follows:

- Decrypt the search token within the enclave and parse the keywords as $\{w_1, w_2, ..., w_q\}$.
- Harness read operation of ORAM to read the ID list of the lest frequent keyword w_1 from the encrypted inverted index.
- Decrypt the ID list within the enclave and test whether id in the ID list belongs to the search result or not. More specifically, for each id in the ID list, we test whether $xtag_i$ computed by combining id with w_i is belonging to bloom filter for $i \in \{2, ..., q\}$. For all $i \in \{2, ..., q\}$, $xtag_i$ belongs to the bloom filter, then id belongs to the result. Otherwise, id does not belong to the search result. Fig. 2 illustrates the idea of this step.

Algorithm 1: Build

Input: Dataset D
Output: BF
construct inverted index IV using D;
store IV in ORAM as E_IV;
store the keyword-identifier pairs computed from IV in BF;

We call this straw-man construction as **Baseline**. The pseudocode of the Baseline is shown in Algorithm 2. To

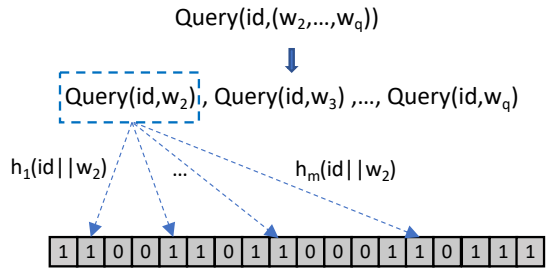


Fig. 2: Test whether identifier id belongs to the result

Algorithm 2: Baseline

Input: search token τ, Inverted index E_IV, BF_id
Output: Search result C_IDList

```
C_IDList=SearchTrusted(τ, E_IV, BF_id);
return C_IDList;
Function SearchTrusted (τ, E_IV, BF_id):
    BF ← Load(BF_id);
    Tplain=Dec(SK,τ);
    parse Tplain as w_1, w_2, ..., w_q;
    IDList_W_1=ORAM(w_1,E_IV);
    posList=∅;
    for id_i in IDList_W_1 do
        flag ← 1;
        for w_i in {w_2, ..., w_q} do
            xtag ← id_i||w_i;
            for h_i in H do
                pos = h_i(xtag);
                if BF[pos] ≠ 1 then
                    flag ← 0;
                    break;
        if flag==1 then
            posList.add(id_i)
    return posList;
```

evaluate the performance impact of the database's size increase, we evaluate the delays of conjunctive queries by implementing these straw-man constructions. To better compare the performance of these straw-mans, we use the size of bloom filter in Baseline to present the size of the database and set the false positive of bloom filter as 0.001. For simplicity, we refer to the setup details of the system in Section 7.1. In addition, we evaluate query delays only containing the search time on the bloom filter within the enclave, which we refer to as **Baseline_Batch**.

Performance analysis. When the number of keywords in the search token is 14, Fig. 3 presents the performance of the straw-man constructions with increasing key-value pairs. As shown in the left sub-graph of Fig. 3, when the size of bloom filter is smaller than 67 MB, the performance of **NoBF** and **Baseline** are close. However, when the bloom filter's size is larger than 100MB, the performance of **Baseline** seriously decreases. As shown in the right sub-graph of Fig. 3, the time of query response without containing the time to load the bloom filter into the enclave is fast. Therefore, the

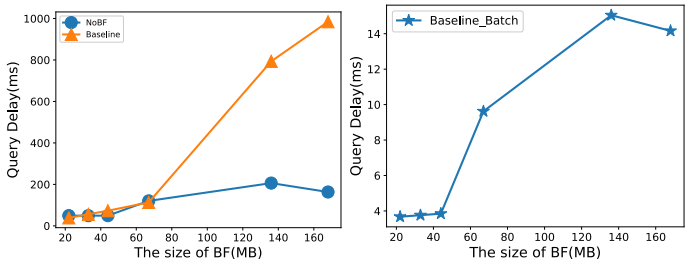


Fig. 3: The performance comparison of straw-man constructions

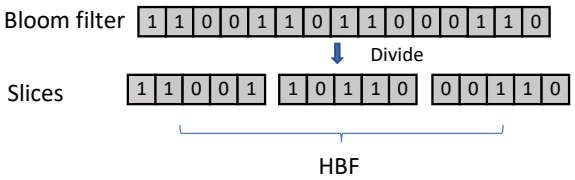


Fig. 4: A simple example of HBF in Construction 1

main cost of the search procedure in Baseline is to load the bloom filter into the enclave.

5 Constructions

We describe the basic **Cryptographic tools** firstly. Our construction makes use of a symmetric-key encryption scheme(**Gen**, **Enc**, **Dec**). For instance, we use a Rijndael AES-GCM encryption scheme provided by SGX SDK. More specially, we use sgx_rijndael128gcm_encrypt and sgx_rijndael128gcm_decrypt for **Enc** and **Dec** respectively.

5.1 Construction 1

Intuition. Although the sealed property of SGX can make the large size of a bloom filter proceed, the frequency of context switch and communication between the enclave and untrusted part of the system can delay the response time of search query. Construction 1 divides the bloom filter into multiple bloom filters at the client side and then stores these bloom filters in a table named HBF and uploads HBF to the untrusted cloud. An illustrative example of HBF is shown in Fig.4. Formally, we give the pseudo-code of Build in Algorithm 3:

Algorithm 3: Build

Input: Dataset D
Output: HBF

```
construct inverted index IV using D;
store IV in ORAM as E_IV;
store the keyword-identifier pairs computed from IV
  in BF;
divide BF into {bf}_m;
store {bf}_m in HBF;
```

The search algorithm executed at the server side is shown in Algorithm 4, which takes the search token t,

encrypted inverted index E_IV and HBF as input and outputs the id list which satisfies the search token. See details as follows:

Algorithm 4: Search

Input: search token τ, Inverted index E_IV, HBF
Output: Search result C_IDList

1 C_IDList=SearchTrusted1(τ, E_IV, HBF);
2 return C_IDList;
3 **Function `SearchTrusted1`** (τ, E_IV, *HBF*)**:**
4 Tplain=Dec(SK,τ);
5 parse Tplain as $w_1, w_2, ..., w_q$;
6 $IDList\_W_1$=ORAM(w_1,E_IV);
7 $posList$=$\emptyset$;
8 **for** id_i *in* $IDList\_W_1$ **do**
9 $posList_i$=$\emptyset$;
10 **for** w_i *in* $\{w_2, ..., w_q\}$ **do**
11 $xtag \leftarrow id||w_i$;
12 **for** h_i *in* H **do**
13 pos=h_i(xtag);
14 $posList_i$.add(pos);
15 $posList$.add($posList_i$);
16 **for** *line in HBF* **do**
17 $BF \leftarrow Load(line, HBF)$;
18 **for** $posList_i$ *in* $posList$ **do**
19 **if** $pos \in posList_i$ & $BF[pos] \neq 1$ **then**
20 $posList$.remove($posList_i$);
21 $IDList\_W_1$.remove(id_i);
22 return $IDList\_W_1$;

Specifically, Algorithm 4 uses the path ORAM to read the id list named $IDList\_W_1$ which contains the identifiers of documents containing w_1 which is the least frequency among the search keywords. To test whether the identifiers within $IDList\_W_1$ are contained in the search result or not, there are two steps to test whether the *xtag* is belonging to the bloom filter or not. First, we calculate the positions of xtag encoded in bloom filter using hash functions (line 8-15). Second, we read the bloom filters sequentially from HBF on the untrusted part of the system to the enclave by using function $Load$. Then we test whether the elements stored at the corresponding positions of the bloom filter are the ones. If there is a zero, then we have to decide whether the remove the array $posList_i$ from $posList$ and remove id_i in $IDList\_W_1$ from (line 16-22).

5.2 Construction 2

Intuition. Construction 1 needs to load the whole bloom filters in HBF into the enclave, which leads performance bottleneck caused by the context switches between the untrusted memory and the enclave. The reason is that we need to calculate the positions of HBF corresponding to multiple keywords parsed from the search token, and the hash functions distribute the keywords evenly in HBF. This motivates us to find out a solution that we only need to read few bloom filters in HBF instead of all bloom filters into the enclave. An observation is that a conjunctive query usually contains multiple keywords and the universe keywords in the dataset can be divided into several classes. The main idea of Construction 2 is that the users classify the universe of keywords into multiple groups and then store each group in a bloom filter which is a component of HBF. Finally, the client uploads HBF to the server side. When receiving the search token from the client, the server first finds the group identifiers IDs of the keywords contained in the search token and then loads the corresponding groups to the enclave and then uses these bloom filters to filter out the search result. As there are a lot of approaches to classify the universes of keywords in machine learning, we do not consider the problem of how to classify the keywords. However, how to find the bloom filters in HBF corresponding to the keywords in the search token is an important problem to solve. This problem can be reduced to hardness of finding the group identifiers of the keywords.

Find the group identifier of the keyword in search token. To solve the above problem, we make use of the bloom tree's properties proposed in [16]. As the nodes in bloom tree are bloom filters, it is suitable for limited storage of the enclave. We chose the number of hash functions at each level of bloom tree according to the equation proposed in [16] that is

$$k_i = \begin{cases} log_2(d) & 0 \leqslant i \leqslant l-1 \\ log_2\left(\frac{l \times (d-1))}{u_c \times d}\right) & i = l \end{cases} \quad (3)$$

, where k_i denotes the number of hash functions for a bloom filter node at level i, l is the height of the bloom tree, u_c is the upper bound for the probability of a classification failure and d is the branching factors.

We describe Construction 2 as follows:

Algorithm 5: Build

Input: Dataset D
Output: HBF

1 construct inverted index IV using D;
2 store IV in ORAM as E_IV;
3 HBF $\leftarrow \emptyset$;
4 divide the universe keywords set W to m groups $\{gr\}_m$;
5 construct BF_tree using $\{gr\}_m$;
6 $i = 0$;
7 **for** gr *in* $\{gr\}_m$ **do**
8 $BF_i \leftarrow \emptyset$;
9 **for** kw *in* gr **do**
10 $IDList \leftarrow IV(kw)$;
11 **for** id *in* $IDList$ **do**
12 $xtag \leftarrow kw||id$;
13 insert $xtag$ in BF_i;
14 HBF.add(BF_i);
15 $i++$;

Algorithm 5 firstly constructs the inverted data structure (Line 1) and then stores the inverted data structure in ORAM (Line 2). To get the groups of the universe keywords W, Algorithm 5 constructs a bloom filter tree BF_tree (Line 5). An example of constructing bloom tree is shown in Fig. 6. Then, Algorithm 5 encodes the keyword kw in each group

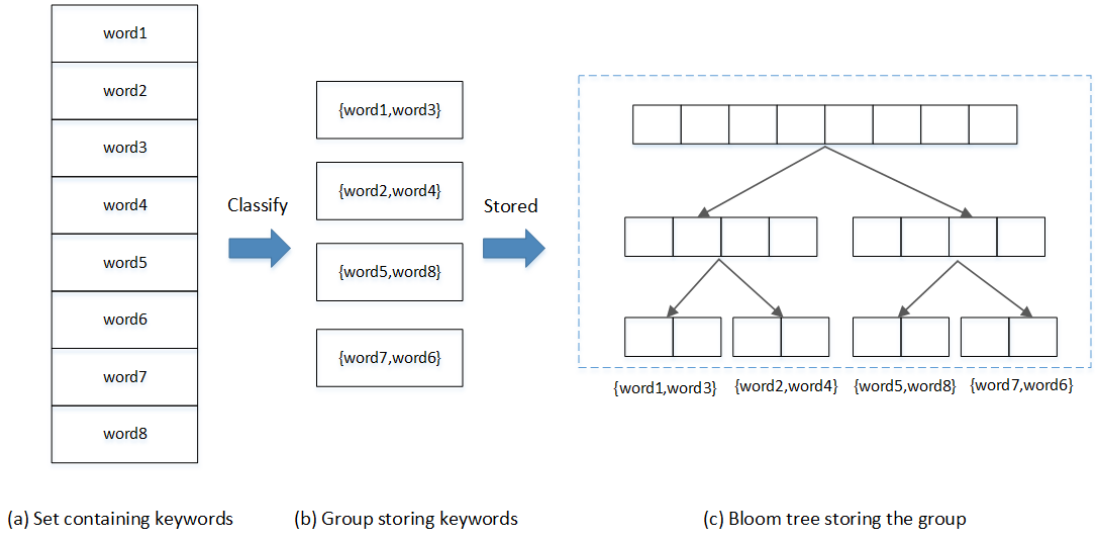


Fig. 5: An example of constructing bloom tree

gr and its corresponding ID list into a bloom filter (Lines 6-13). Finally, store these bloom filters into HBF(Line 14).

Algorithm 6 firstly loads BF_tree into the enclave and then performs the search operations within the enclave. In Algorithm 6, the code executed in the enclave decrypts the search token to get the search keywords first (Line 1) and then uses the read operation of ORAM to get the ID list of the first keyword obliviously(Line 8). Then we search the bloom tree using **Search_BF_tree** in [16] to get the ID list of the groups containing the keywords ID_group and the set W_search which contains the keyword lists corresponding to the group in ID_group (Line 9-10). When obtaining the ID list of the documents satisfying the least frequent keyword such as w_1 and the ID list of the group, we use function **ReadObj_Bf** to get the corresponding bloom filter bf in HBF (Line 15) and corresponding keyword list(Line 16) and then search the ID list containing all the keywords (Line 17-26).

Security discussion. The leakage in Construction 2 compared with Construction 1 is the number of groups containing the keywords in one search token. If the search keywords in one search token are in the same group, there is no leakage in Construction 2. If the search keywords in one search token contain server groups, the number of groups is leaked and which group is leaked. We think this leakage is acceptable. Considering the keywords containing characteristics of several features to search for a patient in the hospital datasets, it is acceptable to know that the characteristics are distributed in different disease datasets.

5.3 Discussion

We leak the identifier of the group contained in the keywords in one search token in Construction 2. In order to increase the security of Construction 2, we propose Construction 3 to protect the group information of the keywords in search token. Similar to Construction 1, a naive approach to protect the group information is to load all the group bloom filters into the enclave sequentially. The communication overhead is extremely expensive which can delay the response time of the queries. Inspired by [29], an additive homomorphic encryption scheme executing at the server side can be used to save the network bandwidth between the enclave and the untrusted memory. Using the homomorphic encryption scheme, Construction 3 can increase security and decrease the bandwidth between the enclave and the untrusted memory.

Similar to Construction 2, Construction 3 firstly loads the bloom tree into the enclave and then performs the search operations within the enclave. The difference from Construction 2 is the way to get the bloom filter from HBF stored on the untrusted memory. More specifically, when getting the identifier ids of the group bloom filters, the procedure executed within the enclave uses vectors E(1) or E(0) which can hide 1 or 0 by encrypting 1 or 0 to retrieve the bloom filters obliviously. We show an example of getting bloom filter obliviously in Fig.6, which gets the bloom filter from the untrusted memory obliviously.

6 Security Analysis

Similar to the security theorem statement of ObliDB [2], we give an informal theorem statement in which the simulator algorithm only sees the leakage functions that we intend to leak to demonstrate that our constructions is oblivious. Let D be a dataset, and Q be a query, and $P = Cons(D, Q)$ be Construction 1 or Construction 2 and $TRACE$ be the trace of memory access for every operation in the process of the corresponding construction. We define the leakage functions of the constructions denoted by P including the build leakage L_1 and the search leakage L_2 for a conjunctive query Q. For Construction 1, L_1 contains the number of documents, the size of each bloom filter in HBF, the size of the inverted index and the size of each record in the inverted index and L_2 contains the number of search results denoted by R and $TRACE$. For Construction 2, L_1 contains the number of documents, the size of each bloom filter in HBF, the number of nodes of bloom tree and the size of each node of bloom tree, the size of the inverted

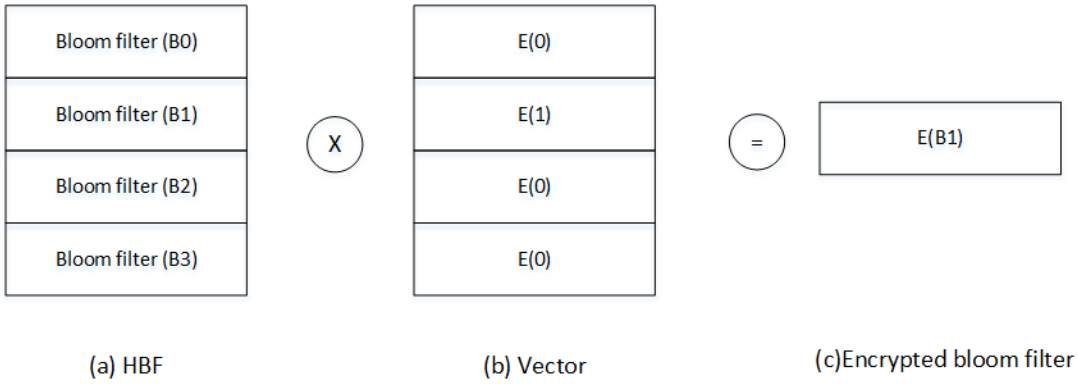


Fig. 6: An example for getting bloom filter obliviously

Algorithm 6: Search

Input: search tokenτ, bloom tree BF_tree, HBF
Output: Search result C_IDList

```
1  Load BF_tree into the enclave;
2  C_IDList=SearchTrusted2(τ, BF_tree, HBF);
3  return C_IDList;
4  Function SearchTrusted2 (τ, BF_tree, HBF):
5      C_IDList← ∅;
6      Tplain=Dec(SK,τ);
7      parse Tplain as {w_1, w_2, ..., w_q} ;
8      IDList_W_1=ORAM(w_1);
9      W_search ←⊥
10     ID_group=Search_BF_tree({w_1, w_2, ..., w_q},
         BF_tree, &W_search);
11     Size=IDList_W_1.size();
12     flag[Size] ← true;
13     i ← 0;
14     for id in ID_group do
15         bf ← ReadObj_Bf(id,HBF);
16         w_list ← W_search[i++];
17         for id' in IDList_W_1 do
18             index← the index of flag corresponding
                 id';
19             if flag[index] then
20                 for w_i in w_list do
21                     xtag ← id'||w_i;
22                     for h_i in H do
23                         pos=h_i(Xtag);
24                         if bf[pos] = 0 then
25                             flag[index]=false;
26                             break;
27     for id in IDList_W_1 do
28         index← the index of flag corresponding id;
29         if flag[index] then
30             C_IDList.add(id);
31     return C_IDList
```

index and the size of each record in the inverted index, and L_2 contains the number of search results and $TRACE$.

Theorem 1. For all D, P, Q, and a security parameter λ, there exists a probabilistic polynomial-time simulator SIM such that all efficient adversaries A,

$$\left|Pr\left[A(SIM(\widetilde{TRACE}(|D|, P, Q))) = 1\right]\right.$$
$$-Pr\left[A(TRACE(|D|, P, Q)) = 1\right] \leq negl(\lambda)$$

Proof 1. At a high level, the proof is to describe a probabilistic polynomial time (PPT) simulator SIM for which a PPT adversary A can distinguish between $TRACE(D, Q)$ and memory traces created by SIM with negligible probability using leakage functions L_1 and L_2 for each construction. The main difference between Construction 1 and Construction 2 to build SIM is to simulate the traces $\widetilde{TRACE}$ which is indistinguishable from $TRACE$. For each query q, SIM chooses a random string as the search token whose length is the same in P by L_1. Using L_1 and P, SIM initializes n arrays, each item in the block is a random string whose length is equal to the length of the item in HBF in P. Using $TRACE(D, Q)$, SIM selects the corresponding blocks and selects random positions from these blocks and replaces the items with the identifiers in R. Now SIM simulates secure hardware: SIM loads the bloom filters in HBF to the enclave via $TRACE(D, Q)$ in L_2. When receiving the search tokens, SIM simulates the search result $\tilde{R}$ using R contained in L_2. If the identifiers are contained in R, SIM randomly selects the positions of bloom filters whose items are 1s as the results of the hash functions. For the identifier not contained in R, SIM randomly selects at least one position of bloom filters whose item is 0 as a part of the result of the hash functions. Due to the security of the pseudo-random function, the security of the symmetric encryption scheme, the security of SGX and the obfuscation of Path ORAM, A can not distinguish the simulated traces $\widetilde{TRACE}$ with $TRACE$ of P.

7 Implementation and Evaluation

7.1 Experimental Setup

Setup. We implemented our constructions in 3100 LOCs of C++ and C. The client and server were executed on a

Dell workstation equipped with an Intel Core E3-1225 v5 CPU@ 3.30GHZ with Ubuntu 16.04 Server, 8GB memory and 4 threads for each CPU. Note that, we run the SGX driver, SDK, and platform software version 1.8.

Parameters. We configure the following Parameters in our experiments. (1) Number of key-identifier pairs. (2) Number of keywords in the search token. (3) Size of bloom filter in HBF. (4) Size of each block of path ORAM. (5) False positive rates of bloom filter in HBF and bloom tree.

Datasets. The basic input of our construction is the inverted index. Whether it is a real dataset or a synthetic dataset, it needs to be processed in the form of an inverted index. After our evaluation, we observe that there is no difference between our index structure construction and search procedure on the synthetic dataset and the real datasets. We use a synthetic dataset and a real dataset Enron email [30] to evaluate our experiments. Specifically, we implement the synthetic datasets by generating key-identifier pairs with a uniform distribution.

7.2 Preprocessing evaluation

We use four synthetic datasets as shown in Table 1. The configurations of parameters and the setup costs of the data structures used in our schemes are shown in Table 1, Table 2 and Table 3.

TABLE 2: Configurations and setup costs of path ORAM

Dataset	*Block*(10^3*B*)	$N_{\text{key}-\text{vale}}$	Setup(s)
Tiny	55	15061133	187.452
Small	55	40091315	244.740
Medium	55	54132772	302.644
Large	55	60131027	427.029

TABLE 3: Configurations and setup costs of HBF

Dataset	*HBF(MB)*	*fp*	Setup(s)
Tiny	53	0.0001	9.189
Small	80	0.0001	13.798
Medium	102	0.0001	22.926
Large	135	0.0001	31.263

TABLE 4: Configurations and setup costs of Bloom Tree

Dataset	*Bloom Tree(KB)*	*fp*	BF_{group}*(B)*	Setup(ms)
Tiny	45	0.000001	240000	12.207
Small	70	0.000001	240000	20.399
Medium	87	0.000001	240000	27.631
Large	116	0.000001	240000	33.281

We give the setup costs for index data structures, including the data structures of path ORAM, HBF and bloom tree. We observe that the overhead for constructing the path ORAM is more expensive than constructing HBF and bloom tree and the overhead for constructing bloom tree is cheap.

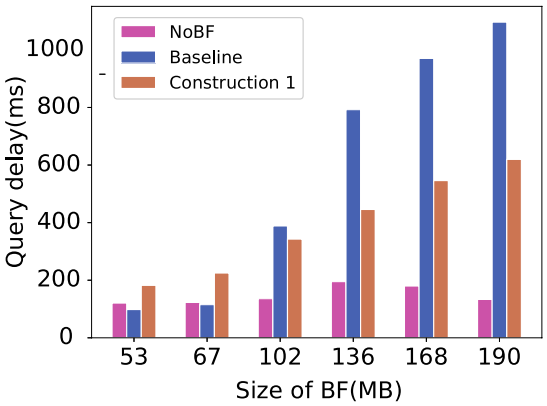


Fig. 7: The evaluation of Construction 1

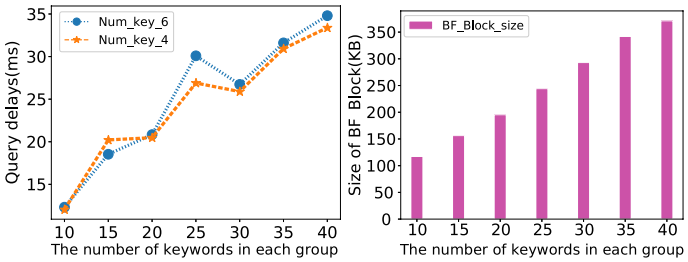
7.3 Evaluation and analysis.

The performance of Construction 1. To compare the performance of Construction 1 with the straw-man constructions, we evaluate the query delays on the server side with respect to different sizes of bloom filters which can store all of the key-value pairs, as shown in Fig. 7. In this experiment, there are 14 keywords contained in one search token, and the false positive rate of bloom filter in Baseline and Construction 1 is 0.0001. The size of bloom filter in HBF of Construction 1 is 14000 bytes. The number of key-value pairs in the dataset ranges from 2.4*10^7 to 8.5*10^7. In Fig. 7, we use the bloom filter size in Baseline to denote the size of the dataset.

Performance analysis. As shown in Fig. 7, we observe that when the size of the bloom filter in Baseline is smaller than 100MB, the query delays of Construction 1 are slightly smaller than Baseline, while when the size of the bloom filter in Baseline is bigger than 100MB, Construction 1 can reduce the query delay 0.5x but higher than NoBF. It is reasonable because when the size of bloom filter is bigger than 100MB, query delays of Baseline contain the cost of communication and paging cost. Compared with the Baseline, the query delays of Construction 1 contain the cost of the context switch but no the paging cost. When the size of the bloom filter is smaller than 100MB, the frequency of the context switch of Construction 1 is higher than Baseline.

The performance of Construction 2. In these experiments, we set the false positive of the bloom filter in HBF to 0.0001 and the false positive of the bloom tree to 0.000001. We repeated fifty times to report the average results. For simplicity, we do not encrypt the search queries, which has little impact on the query delays for each construction. We evaluate the query delays on the server side with respect to different sizes of the group bloom filter and the different number of groups contained in one search token.

1) Varying the group size. We evaluate the performance of Construction 2 which varies the number of keywords in each group with 6.1*10^7 key-value pairs. In this experiment, we consider the situation where the keywords in one search token belong to the same group. We evaluate the impact of bloom filter size in HBF on query delays. Fig. 8 (a) illustrates the query delays with a varied size of bloom filter in HBF shown in Fig. 8(b). With the increase in group sizes, the query delays increase as the communication overheads



(a) (b)

Fig. 8: Query delay with different size of groups

between the enclave and the untrusted memory increase. In Fig. 8(a), we also measure the query delays for the different number of keywords in one search token. We observe that the gap of query delays for different sizes of search tokens is similar. The reason is that the communication overheads of the two search tokens are similar for the same group and the membership testing is efficient on the bloom filter within the enclave.

2) Varying the number of groups containing the search token. We evaluate the impact of groups needed to be loaded into the enclave in Fig. 9. In Fig. 9(a), We use 6.1*10^7 key-value pairs and fix the number of keywords N_{key} in the search token. As the number of groups loaded into the enclave increase, the query delays increase. There is no significant difference among the different numbers of ID list of W_1 that is N_{W1_ID}. Because the query delay is mainly caused by the cost of a context switch between the enclave and untrusted memory. In Fig. 9(b), we select 4.6*10^7 key-value pairs and fix the number of identifiers contained in the result that is N_{result}. We evaluate the query delays with the varied numbers of groups and the varied number of keywords contained in the search token. We observe that the number of keywords in the search token did not significantly affect query delays. With the varied number of keywords in the search token, the query delays were very similar when the number of groups containing the keywords in the query is the same. Fig. 9(c) illustrates the query delays with the varied number of groups on 4.6*10^7 key-value pairs. We observe that the query delays are similar with different sizes of results and increase when the number of groups needed to be loaded into enclave increases. From Fig. 9, we could infer that reducing the query delays needs optimizing the number of groups containing keywords in search token, which can be solved by classification problem.

3) Comparison to Construction 1. To present the performance gap between Construction 1 and Construction 2, we first search the keywords contained in the same group. In this experiment, there are 14 keywords contained in the search token and the false positive of bloom filters in HBF is 0.0001. The number of key-value pairs ranges from 2.4*10^7 to 6.1*10^7.

We evaluate the query delays with various sizes of bloom filters which can contain all the key-value pairs, as shown in Fig. 10. We can observe that Construction 2 can reduce the query delays approximately by $16.5 \sim 36.8\times$. Since the client divides the keywords into several groups, the keywords contained in the search token influence the number of bloom filters in HBF needed to load into the enclave. To show the best performance of Construction 2, we search the keywords contained in one group bloom filter. As we know, the worst case is that the keywords in a conjunctive search are stored in all of the bloom filters in HBF.

4) Comparison to ObliDB. In this experiment, we give the comparison of Construction 2 and ObliDB proposed in [2] and the results are shown in Fig. 11. More specifically, we set the false positive of the bloom filter in HBF to 0.0001 and the false positive of the bloom tree to 0.000001.

(a) Varying the size of HBF. We compare Construction 2 with ObliDB [2] on simulated data in Fig. 11(a). For the X-axis coordinates of Fig. 11(a), we translate the number of key-value pairs to the size of the bloom filter calculated by equation (2). We evaluate the query delay for a search token containing 14 keywords belonging to the same group in Construction 2 and the number of identifiers in the result is 10. Construction 2 outperforms ObliDB's point index selection by $98.2 \sim 134.1\times$. The reason why Construction 2 performs significantly faster than ObliDB is that Construction 2 only needs one memory access using ORAM operation and reads the group bloom filter only once.

(b) Varying the number of keywords in the search token. We compare Construction 2 with ObliDB on 191123 key-value pairs by varying the number of keywords in the search token and all the keywords are contained in one group. Each group in the bloom tree contains 20 keywords. From Fig. 11(b), we observe that Construction 2 outperforms ObliDB by up to $5.3 \sim 41.1\times$.

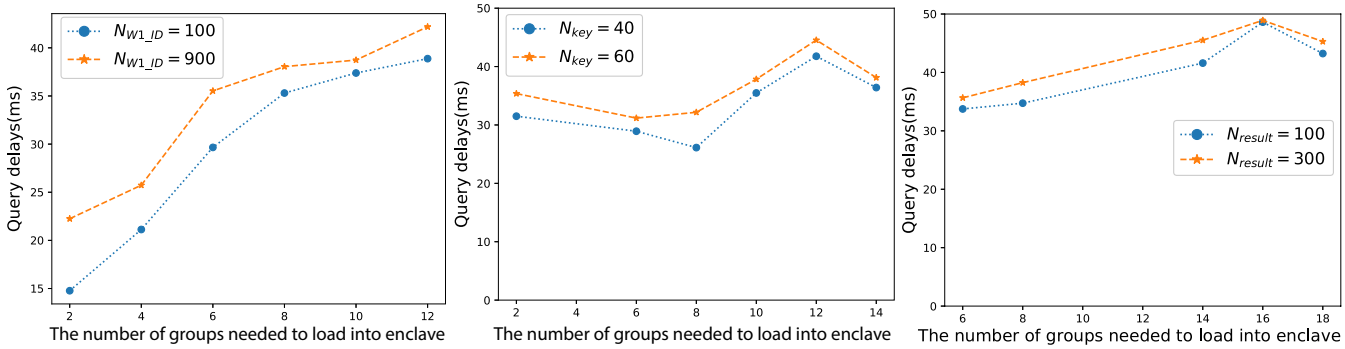
(c) Varying the number of groups loaded into enclave. We compare Construction 2 with ObliDB on 770446 key-value pairs by varying the number of groups needed to load in the enclave using 20 keywords in the search token. The number of the result is 100 and each group in the bloom tree contains 20 keywords. From Fig. 11(c), we observe that Construction 2 outperforms ObliDB by up to $20.1 \sim 42.5\times$.

5) Comparison to POSUP. In this experiment, we give the comparison of Construction 2 and POSUP proposed in [1] and the results are shown in Fig. 12. More specifically, we evaluate the latency of POSUP on the Enron email dataset [30].

(a) Varying the number of emails in the dataset. We compare Construction 2 with POSUP [1] on simulated data in Fig. 12(a). We evaluate the query delay for a search token containing tree keywords belonging to the random groups in Construction 2 and the number of identifiers in the result is variable from 236 to 885. Construction 2 outperforms POSUP's query efficiency by $11.8 \sim 32.4\times$. The reason why Construction 2 performs significantly faster than POSUP is that Construction 2 can do conjunctive queries while POSUP does the single keyword query.

(b) Varying the number of groups loaded into enclave. We compare Construction 2 with POSUP [1] on 350000 emails by varying the number of groups needed to load into the enclave from 2 to 9 using 10 keywords in the search token. Each group in the bloom tree contains 20 keywords in the initial phase. From Fig. 12(b), we observe that Construction 2 outperforms POSUP by up to $15.8 \sim 25.9\times$.

(c) Varying the number of keywords in the search token. We compare Construction 2 with POSUP on 350000



(a) N_{key}=20 (b) N_{result}=200 (c) N_{key}=60

Fig. 9: Query delay for varying the number of groups containing the search token

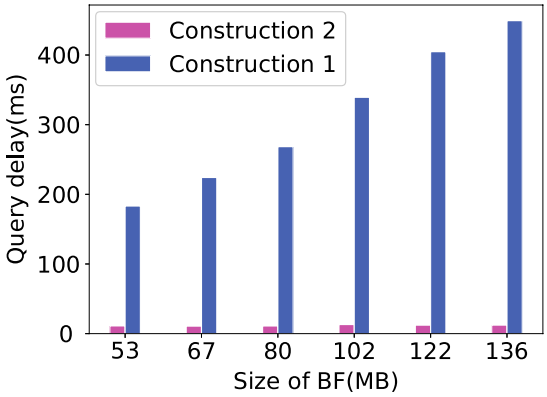


Fig. 10: Comparison of Construction 2 and Construction 1

emails by varying the number of keywords in the search token from 2 to 18 and all the keywords are contained in four random groups. Each group in the bloom tree contains 20 keywords in the initial phase. From Fig. 12(c), we observe that Construction 2 outperforms POSUP by up to $9.8 \sim 35.1\times$.

6) Comparison to Oblix. In this experiment, we give the comparison of Construction 2 and POSUP [1] and Oblix [9] and the results are shown in Fig. 13. We vary the size of key-value pairs contained in the real dataset Enron email [30] from 2^{16} to 2^{23}. For the X-axis coordinates of Fig. 13, we evaluate the query delay for a search token containing 2 keywords belonging to the random groups in Construction 2 and the number of identifiers in the result is variable from 22 to 87. We observe that Oblix outperforms Construction 2's search efficiency by $6 \sim 19.5\times$. The reason why Oblix performs faster than Construction 2 is that Oblix only focuses on the single keywords and one result and does not get the whole results in this experiment. While Construction 2 only needs one memory access using ORAM operation and reads the group bloom filter only once for the whole results.

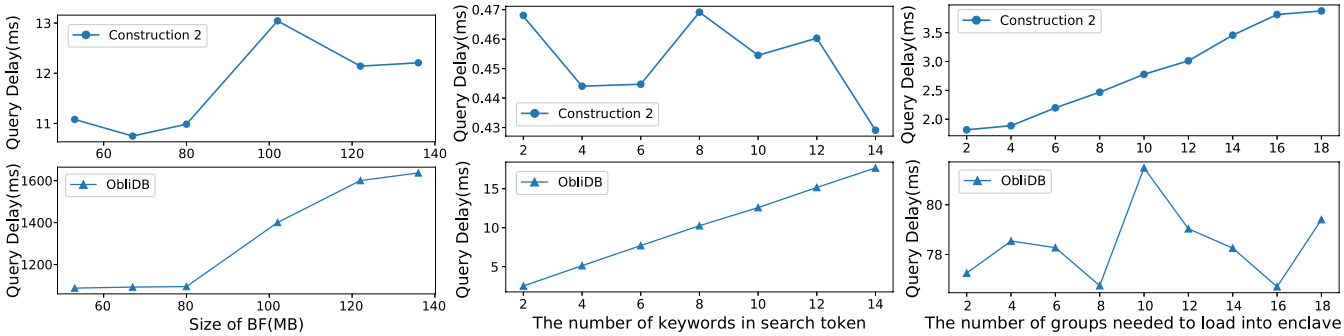
The performance of Construction 3. We give the performance evaluation of Construction 3 compared with Construction 2 and ObliDB in Fig. 14. We vary the number of keywords in the search token. The number of the result is 20. More specifically, in Fig. 14(a), the number of key-value pairs is 104507, and the number of keywords in the group is 10. In Fig. 14(b), the number of key-value pairs is 104507 and the number of keywords in the group is 20. In Fig. 14(c), the number of key-value pairs is 255941 and the number of keywords in the group is 10.

Performance analysis. In Fig. 14, we can observe that the overhead of Construction 3 is extremely expensive. The reason is that the homomorphic encryption scheme is public key encryption scheme which is expensive for efficiency and in order to promise the correctness of additively homomorphic encryption, only a limited length of bloom filters can be computed correctly, which increases the frequency of context switches between the enclave and the untrusted unit. Although using a homomorphic encryption scheme can reduce the bandwidth between the enclave and the untrusted memory, it is very expensive for query response time. If we can use some symmetric homomorphic encryption scheme, we may significantly improve the query efficiency on the server. The promising method is using an additively symmetric homomorphic encryption scheme (ASHE) [31]. Because ASHE is up to three orders of magnitude more efficient than Paillier. However, ASHE can not be applied in Construction 3 directly to improve the search performance. As the goal of this work is to reduce the paging and exits of trusted areas in the untrusted server, how to employ ASHE in Construction 3 to improve the search performance is our future work.

8 Related Work

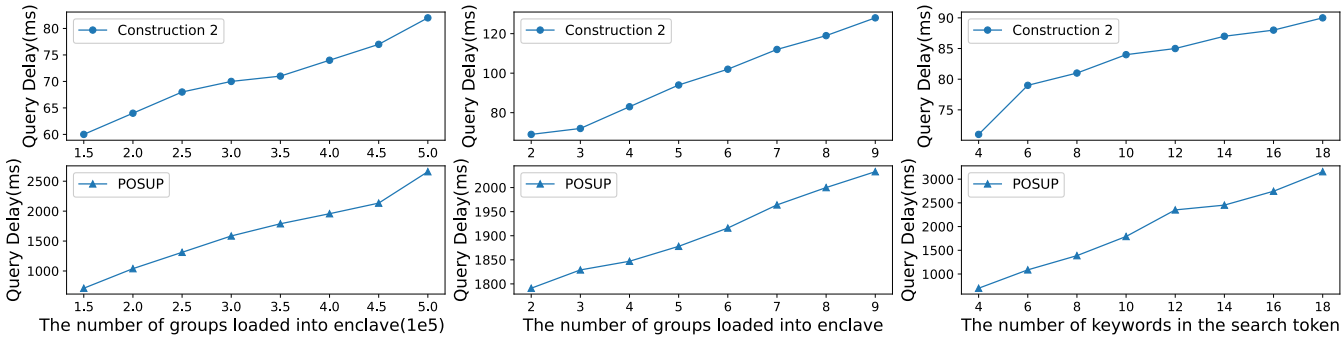
Oblivious search schemes on SGX. Several SSE schemes [1], [2], [9], [17], [19], [32] have been proposed to improve the efficiency of private search on the encrypted database based on Intel SGX. These works focus on secure challenges arising from side-channel attacks from SGX with access pattern obliviousness. Oblix [9] works on doubly-oblivious path ORAM to hide the access pattern. ObliDB [2] combines path ORAM with SGX enclave to construct an oblivious search scheme, but the untrusted server needs to scan the datasets before the search operation which leads to search performance degradation.

Alleviate the limitation of SGX. To mitigate the memory limitation of the enclave created by SGX, several systems [10], [11], [33], [34] have been proposed. Eleos [34] reduces the overheads of paging in SGX by designing a secure user-managed virtual memory abstraction that provides the same



(a) N_{result}=10 (b) N_{result}=10 (c) N_{result}=100

Fig. 11: Comparison of Construction 2 and ObliDB



(a) $N_{keyword}$=3 (b) N_{file}=350000 (c) N_{file}=350000

Fig. 12: Comparison of Construction 2 and POSUP

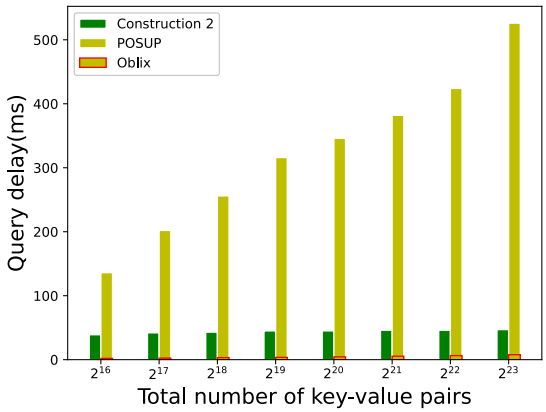


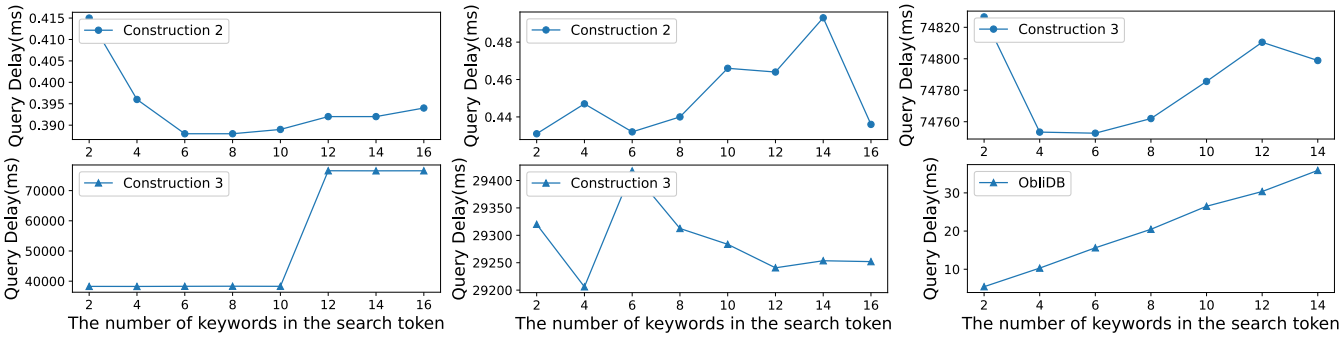
Fig. 13: Comparison of construction2 and Oblix

security and functionality as EPC and the paging mechanism to the enclave. ShieldStore [11] mitigates the performance degradation for accessing enclave memory pages by putting the main secret keys and integrity meta-data in the enclave instead of storing the whole data structures such as a hash table in the enclave. Vault [33] enables a large EPC region by designing integrity tree structures that have lower bandwidth and capacity overheads. HotCalls [13] firstly investigates the sources of performance degradation and then leverages the insights to design a fast interface for the enclave. All of these works do not consider the performance degradation for oblivious conjunctive keyword search. HardIDX [10] leverages the secure enclave to design a secure index and consider the situation where the size of the dataset is larger than the size of EPC but does not consider the access pattern leakage of the private search on the encrypted datasets. Reocs is the first system as we know to provide the oblivious conjunctive search with high search efficiency.

Secure Database and private boolean search. EnclaveDB [35] provides a database engine for guaranteeing the confidentiality and freshness of data and queries. ServerDB [36] focuses on multi-dimensional range queries on the outsourced database. Dynamic SSE schemes [3], [37], [38] focus on the privacy of queries on the untrusted cloud. Unlike these systems, private boolean search [15], [39], [40] on the encrypted database has been proposed to improve the efficiency of boolean search. IEX [39] focuses on the private disjunctive search but does not consider the access pattern leakage. Blind seer [40] can search arbitrarily boolean search in a sub-linear search efficiency but has significant communication overheads caused by the garbled circuit. BXT [15] and OXT [15] can perform private conjunctive search efficiently but not consider the access pattern leakage.

9 Conclusion

This work first addresses the problem of performance degradation for oblivious conjunctive keyword search due to the limitation of secure memory created by SGX. We first



(a) Size$_{BF}$=4612B (b) Size$_{BF}$=7378B (c) Size$_{BF}$=7627B

Fig. 14: The evaluation of Construction 3

identify the bottlenecks of oblivious conjunctive search by combing SGX technology using straw-man solutions. The insights from the straw-man solutions allow us to design efficient protocols. This work uses a bloom filter within the enclave to filter the identifiers list of the keywords, which severely reduces the frequency of exiting the enclave and paging overhead. In addition, this work first divides the keywords into classes and uses the bloom filter tree to find the objective bloom filter in HBF efficiently. Our implementation shows that our schemes are suitable for searching a large number of keywords in one search token and the state of the arts such as oblidb [2] and Oblix [9] which focuses on the number of search results is orthogonal to our focus and can be readily used to complement our design.

REFERENCES

[1] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *Proc. Priv. Enhancing Technol*, pp. 172–191, 2019.

[2] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *Proc. VLDB Endow.*, vol. 13, no. 2, pp. 169–183, 2019.

[3] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *Proc. NDSS*, 2021.

[4] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. CCS*, 2006, pp. 79–88.

[5] Y. Lu, "Privacy-preserving logarithmic-time search on encrypted data in cloud," in *Proc. NDSS*, 2012.

[6] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proc. CCS*, 2017, pp. 1243–1255.

[7] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade, "Practical passive leakage-abuse attacks against symmetric searchable encryption," in *Proc. ICETE*, 2017, pp. 200–211.

[8] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. NDSS*, 2012.

[9] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *Proc. IEEE S&P*, 2018, pp. 279–296.

[10] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "Hardidx: Practical and secure index with SGX," in *Proc. IFIP*, 2017, pp. 386–408.

[11] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with SGX," in *Proc. EuroSys*, 2019, pp. 14:1–14:15.

[12] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[13] O. Weisse, V. Bertacco, and T. M. Austin, "Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," in *Proc. ISCA*, 2017, pp. 81–93.

[14] Q. Jiang, Y. Qi, S. Qi, W. Zhao, and Y. Lu, "Pbsx: A practical private boolean search using intel SGX," *Inf. Sci.*, vol. 521, pp. 174–194, 2020.

[15] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. CRYPTO*, 2013, pp. 353–373.

[16] M. Yoon, J. Son, and S. Shin, "Bloom tree: A search tree based on bloom filters for multiple-set membership testing," in *Proc. INFOCOM*, 2014, pp. 1429–1437.

[17] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proc. HASP*, 2013, p. 11.

[18] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: reducing paging overheads in SGX with efficient integrity verification structures," in *Proc. ASPLOS*, 2018, pp. 665–678.

[19] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel SGX," in *Proc. NDSS*, 2018.

[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proc. USENIX WOOT*, 2017.

[21] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proc. EUROSEC*, 2017, pp. 2:1–2:6.

[22] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE S&P*, 2015, pp. 640–656.

[23] Z. Liu, Y. Huang, J. Li, X. Cheng, and C. Shen, "Divoram: Towards a practical oblivious RAM with variable block size," *Inf. Sci.*, vol. 447, pp. 1–11, 2018.

[24] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proc. CCS*, 2013, pp. 299–310.

[25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[26] F. Hao, M. S. Kodialam, T. V. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial bloom filters," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 295–304, 2012.

[27] A. Z. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[28] M. Athanassoulis and A. Ailamaki, "Bf-tree: Approximate tree indexing," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1881–1892, 2014.

[29] Z. Liu, Y. Huang, J. Li, X. Cheng, and C. Shen, "Divoram: Towards a practical oblivious RAM with variable block size," *Inf. Sci.*, vol. 447, pp. 1–11, 2018.

[30] Enron email dataset. [Online]. Available: https://www.cs.cmu.edu/~/enron

[31] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed," in *Proc. USENIX OSDI*, 2016, pp. 587–602.

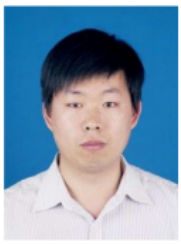
[32] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro:a commodity obfuscation engine on intel SGX," in *Proc. NDSS*, 2019.
[33] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," *ACM SIGPLAN Notices*, vol. 53, pp. 665–678, 03 2018.
[34] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proc. EuroSys*, 2017, pp. 238–253.
[35] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using SGX," in *Proc. IEEE S&P*, 2018, pp. 264–278.
[36] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "Servedb: Secure, verifiable, and efficient range queries on outsourced database," in *Proc. IEEE ICDE*, 2019, pp. 626–637.
[37] C. Zuo, S. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private DSSE for range queries," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 1, pp. 328–338, 2022.
[38] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. ESORICS*, 2019, pp. 283–303.
[39] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Proc. EUROCRYPT*, 2017, pp. 94–124.
[40] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. M. Bellovin, "Blind seer: A scalable private DBMS," in *Proc. IEEE S&P*, 2014, pp. 359–374.

Qin Jiang received her M.S. degree in computer science and technology from ZhengZhou University, ZhengZhou, China, in 2016. She received the Ph.D. degree from Xi'an Jiaotong University, Xi'an, China, in 2022. Currently, She works at Nanjing University of Information Science and Technology. She was a Ph.D. intern student in National University of Singapore. Her research interests include searchable encryption, trusted hardware and blockchain.

Saiyu Qi received the B.S. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2008, and Ph.D. degree in computer science and engineering from Hong Kong University of Science and Technology, Hong Kong, in 2014. He is currently an Associate Processor with the School of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include cloud security, distributed system, trusted hardware and pervasive computing.

Xu Yang received the B.S. degree in information security from University of Electronic Science and Technology of China, Chengdu, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include blockchain technology, searchable encryption.

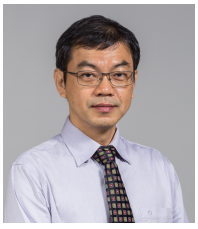
Yong Qi received the Ph.D. degree from Xi'an Jiaotong University, Xi'an, China. He is currently a full professor with from Xi'an Jiaotong University, Xi'an, China. His research interests include operating system, distrusted systems, and cloud computing.



Jianfeng Wang received his M.S. degree in Mathematics from Xidian University, China. He got his Ph.D degree in Cryptography from Xidian University in 2016. Currently, he works at Xidian University. He visited Swinburne University of Technology, Australia, from December 2017 to December 2018. His research interests include applied cryptography, cloud security and searchable encryption.

Youshui Lu received the B.S. degree from The Australian National University, Australia, in 2013, and the M.S. degree from University of Sydney, Australia, in 2015. He works at Xi'an Jiaotong University, China. His research interests include blockchain technology, distributed systems, trusted urban computing and computational social system.

Bochao An received the B.S. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2021. He is currently pursuing the M.S. degree with the School of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include blockchain technology, distributed systems, applied cryptography.



Ee-Chien Chang received the B.Sc and M.S. degree from the National University of Singapore(NUS), and the Ph.D. degree from New York University, New York, in 1998. He is currently an associate Professor with the Department of Computer science, school of computing, NUS. His research interests include blockchain technology, applied cryptography. He has published papers in major conferences/journals such as ACM CCS, USNIX SECURITY, ACM SIGMOD, ESORICS, IEEE TIFS, etc.