Fast Boolean Queries with Minimized Leakage for Encrypted Databases in Cloud Computing

Zhiqiang Wu, Kenli Li, *Senior, IEEE*, Keqin Li, *Fellow, IEEE*, and Jin Wang, *Senior,IEEE*

Abstract—This research revisits the fundamental problem of processing privacy-preserving Boolean queries over outsourced databases on untrusted public clouds. Many current Searchable Encryption (SE) schemes try to seek an appropriate trade-off between security and efficiency, yet most of them suffer from an unacceptable query leakage due to their conjunctive/disjunctive terms that are processed individually. We show, however, this trade-off still can be deeply optimized for more security. We consider a Boolean formula as a set of deterministic finite automatons (DFAs) and propose a novel approach to running an encrypted DFA, which can be effectively and efficiently processed by the cloud. We give three constructions for conjunctive, disjunctive, and Boolean queries, respectively. Their notable advantages are single-round, highly-efficient, adaptively-secure, and leakage-minimized. A lot of experiments are made to evaluate overall efficiency. Testing results show that the schemes achieve enhanced security almost without sacrificing anything of search efficiency.

Index Terms—Cloud Computing, Privacy Preserving, Searchable Encryption.

1 Introduction

Cloud computing enables ubiquitous, convenient, cost-effective, and on-demand network access. Outsourcing data and computing services to clouds becomes popular. However, the key roadblock of cloud computing is data privacy. Clouds are not fully trusted since the servers might be broken by hackers or malicious cloud managers. They can illegally use the private data that is outsourced by data owners or sell users' privacy for money. To preserve data privacy in cloud computing, researchers proposed searchable encryption (SE), which encrypts the private data in such a way that the data can still be queried efficiently. The cloud now can provide search services directly over encrypted data without learning any sensitive information.

Encrypted Boolean computation is a fundamental functionality of database systems. A Boolean query is a series of intersection, union, or negative operations of multi-dimensional condition strings, in which each condition can match zero or more results. Consider the following Boolean query $\varphi = (w_{1,1} \vee w_{1,2} \vee \cdots) \wedge (w_{2,1} \vee w_{2,2} \vee \cdots) \cdots \wedge (w_{u,1} \vee w_{u,2} \vee \cdots)$. A data owner encrypts her documents into a set of encrypted files and outsources the files to the cloud. To quickly match the data, the owner also creates an encrypted index for her documents. The encrypted files and the index constitute an encrypted data table on the public cloud. The owner and data users share a set of secret keys K, which can encrypt the Boolean query φ into $T_K(\varphi)$. The data user sends $T_K(\varphi)$ to the cloud to quickly search the data table to get a set of file identifiers $DB(\varphi)$ that matches φ. We assume the owner and the users are trusted, but the cloud is not fully trusted. The cloud may attempt to obtain information about the content of files and queries from the clients' requests when performing any operations. The cloud needs to return all results accurately while being prohibited from learning any private data.

If the returned file identifiers are in encrypted form when being sent back to the users, we call this scheme response-hiding [28]. We use $DB^*(\varphi)$ to denote a set of encrypted results for φ. If the result identifiers are in plain-text, we call this scheme response-revealing. Let $DB(\varphi)$ denote a set of unencrypted result identifiers. In a response-hiding scheme, if the result sets are not padded with dummy values, generally, we have $|DB^*(\varphi)|=|DB(\varphi)|$.

Ideally, we want the cloud to do anything yet we wish the cloud to learn nothing. It seems like a paradox that can not be well addressed since any operations over encrypted data will induce knowledge that can be learned by the cloud. We call this knowledge leakage. The target of a well-designed SE scheme is to minimize the leakage and retain good search performance.

There are two difficulties when we handle the following encrypted SQL query, "select * from users where name='Tom' or name='Jerry' and gender='Male' ". First, if a sub-linear single-keyword SE scheme is not optimized for the efficiency of Boolean queries, the query will turn out to be a linear search of the data table. Thus, most single-keyword SE schemes don't work well with Boolean queries. Second, if an SE scheme is not optimized for query leakage, its procedure will leak almost half of the records of the data table according to their access pattern.

It is urgent to design an efficient and practical conjunctive/Boolean query SE scheme that provides strong privacy guarantee, since most of the range, substring, wildcard, multi-keyword, and phrase queries can be constructed from Boolean queries. Recent state-of-the-art tree-based SE works

- *Zhiqiang Wu and Jin Wang are with the School of Computer & Communication Engineering, Changsha University of Science & Technology, Hunan, China, 410114 (email: cxiaodiao@hnu.edu.cn).*
- *Kenli Li is with the College of Information Science and Engineering, and National Supercomputing Center in Changsha, Hunan University, Hunan, China, 410082 (email: lkl@hnu.edu.cn).*
- *Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 (email: lik@newpaltz.edu).*

Corresponding author: Jin Wang (e-mail: jinwang@csust.edu.cn).

are capable of processing conjunctive queries with sub-linear search complexity, such as KRB proposed by Kamara and Papamanthou in [9], PBTree in PVLDB'14 [11], IBTree introduced by Li and Liu in ICDE'17 [25], and VBTree introduced by Wu and Li in VLDBJ'19 [43]. However, their query leakage is not well-studied. Consider such a conjunctive query, $a \wedge b \wedge c$. In the above schemes, according to their access pattern, the cloud has extra knowledge DB(a), DB(b), DB(c), DB($a \wedge b$), DB($b \wedge c$), and DB($a \wedge c$). In fact, we want the cloud only to learn DB($a \wedge b \wedge c$). If we convert the schemes into response-hiding ones (remapping file identifiers to other forms at the client-side), they still leak size patterns: $|DB(a)|$, $|DB(b)|$, etc.

Is there a more efficient and secure solution to handling Boolean queries? Our target is to minimize the leakage and improve the overall performance of the index.

1.1 Security Model

We adopt the IND-CKA2 security (i.e., the adaptive IND-CKA) definition proposed by Curtmola et al. in CCS'06 [4], to evaluate the security strength of an SE scheme. The security is parameterized by a leakage function $\mathcal{L}$ that will output all knowledge induced by setup, search, and update operations.

When the encrypted database is initialized, we refer to knowledge of the cloud (the adversary) that comes from the outsourced database as setup leakage (i.e., $\mathcal{L}(EDB)$). If EDB leaks one or more properties of keywords, such as term frequency, distance related information, and order information [15], we call the scheme property-preserving, otherwise it is non-property-preserving. All IND-CKA or IND-CKA2 schemes should be non-property-preserving. When an encrypted query Q is issued, we refer to knowledge induced by queries as query leakage (i.e., $\mathcal{L}(Q)$). The query leakage mainly consists of two parts, search pattern (i.e., the repetition of queries using the same keywords) and access pattern (i.e., the result sets of queries or some information for accessing the encrypted data). If the query leakage contains information correlated to some unqueried keywords (contents that have not been submitted by the users) with non-negligible probability, we say that the leakage is uncontrollable. Otherwise, it is controllable [5]. The leakage $\mathcal{L}$ of all IND-CKA or IND-CKA2 schemes should be controllable.

1.2 Limitation of Prior Art

We classify all single-round searchable encryption solutions into three categories, tree-based ones [9], [11], [25], [43], inverted-index-based ones [26], [28], and other ones [2], [35].

First, in most of current tree-based single-round searchable encryption schemes, when a user submits a Boolean query to the cloud, the cloud learns extra statistical information about the query in addition to the query result, because the cloud can use parts of the trapdoor to query the index with additional results matched. The cloud learns more information than necessary.

Second, the inverted-index-based single-round SE schemes, such as OXT [26] and BIEX [28], enjoy a non-optimized leakage profile. Consider the disjunctive query $a \vee b \vee c$. They leak $|DB(a)|$, $|DB(b)|$, $|DB(c)|$, and etc. In fact, the user wants the cloud only to learn $|DB(a \vee b \vee c)|$. The key problem of this leakage is that it enables the cloud to learn which term is the main factor that causes the final result. OXT and BIEX still suffer from the s-term problem. An s-term of the conjunctive query is the term whose result set is the smallest among all the query terms. In short, the s-term problem denotes that their search efficiency of $a \wedge b$ is not equal to that of $b \wedge a$, and the complexity of $(a \vee b) \wedge (c \vee d)$ is not equal to that of $(c \vee d) \wedge (a \vee b)$.

Third, many searchable encryption schemes, such as the public-key encryption scheme [35], are linear-search solutions.

1.3 Proposed Approach

We introduce the novel forward/intersection/backward token concepts, to avoid a term of a Boolean formula to be handled individually by the cloud because this is the key drawback of most of Boolean SE schemes. A Boolean query is encrypted into these three tokens. We describe the cloud on how to handle the encrypted Boolean query at a high level. The user creates and submits these three types of tokens to the cloud, where the intersection token is dependent on the computational result of the forward token, and the backward token also depends on computational results of the intersection tokens. The computational result of the backward token will yield one or two new forward tokens. Thus, the cloud can recursively apply this rule, until it reaches a final result. This computation looks like accessing encrypted linked lists. We note that in such a process, the query leakage is extremely minimal (level-2-revealing).

We propose an approach to handling and running deterministic finite automatons (DFAs) over encrypted data. We first precompute all DFA transition states that will be used in future, encrypt and put them into each tree node of a tree-based index [43]. When we search on a tree from top to bottom, a Boolean query is considered as a set of encrypted DFAs, where each DFA will be computed in each accessed tree node. The forward/intersection/backward tokens can also help the DFAs to be recursively processed. The notable advantage of the encrypted DFAs is that a DFA-state can be obliviously and efficiently changed from one state to another.

With the above approaches, we present three schemes: VBT-1, VBT-2, and VBT-3 for conjunctive, disjunctive, and Boolean queries, respectively. These three solutions share the same search algorithm. Their difference is that in each tree node, each DFA is constructed particularly. Note that, this DFA is only a logical concept since it is always in encrypted forms in any queries. All DFA-states don't exist alone at all.

1.4 Key Contributions

We summarize our contributions in four aspects.

First, we propose ideal/real encrypted Boolean function (IEBF/REBF) concepts to help one to seek an optimal trade-off between security and efficiency for a single-round Boolean SE scheme. An adaptively-secure level-2-revealing construction is an optimal security-efficiency trade-off that we pursue.

Second, we show how to encrypt a deterministic finite automaton (DFA). We also give a novel approach to processing the DFA over encrypted data. We apply this approach to building sub-linear Boolean SE schemes.

Third, we present three SE schemes that support building an adaptively-secure leakage-minimized sub-linear-search-efficiency encrypted index. As far as we know, VBT-1 is the first single-round sub-linear level-2-revealing conjunctive SE scheme with scalable index size. VBT-2 is the first single-round sub-linear level-2-revealing disjunctive SE scheme. VBT-3 is the first single-round sub-linear level-2-revealing Boolean SE scheme.

Fourth, VBT-3 can hide inner operators of an encrypted Boolean query, such as $\wedge$ and $\vee$.

We note that the basic tree structure (VBTree) used here is not the key contribution of this paper. For ease of illustrating the overall design, we modify the definition of VBTree and detail it in Section 3. As stated in the security analysis, VBTree in [43] and other trees (e.g., PBTree, IBTree) are much weaker than the proposed solutions.

2 Related Work

Searchable symmetric and structured encryption has been studied for a long time [2], [3], [4], [5], [6], [9], [12]. And much progress has been made in current researches, including scalability improvement [18], update privacy [19], [20], [24], [40], [41], expressiveness improvement [11], [25], [26], [28], [29], [43], locality [42], and index rebuilding [38]. Searchable encryption can also be implemented with functional encryption [10], property-preserving encryption [15], secure multi-computation [23], [30], homomorphic encryption [14], and ORAM [17], [22], [45]. These solutions can be applied to encrypted databases [7] [8].

All searchable encryptions can be classified into two categories, non-interactive constructions and interactive ones (e.g., Oblivious RAMs [45]). The second is, in general, much stronger than the first at the cost of high communication overhead or additional computation, since much private-computing work has been done by the client or other parties. Blind Seer in S&P '14 [29] gave an interactive approach to run Boolean queries based on Yao's Garbled Circuits [30], [36]. Since a Garbled Circuit can be used only one time, this makes the scheme highly-interactive.

In SIGMOD '09, Wong et al. proposed a secure distance-computing scheme, called KNN computation [13]. The advantage of secure KNN is that it can partially hide the distances of the points. To some extent, the secure KNN can be considered as an SE scheme whose keywords are a set of converted points. Unfortunately, the secure KNN has a notable drawback that its query leakage is uncontrollable. A secure-KNN computation will leak a set of formulas that are correlated to some unqueried keywords (points). This leakage will enable the adversary to distinguish the simulated view from the real view. More disastrously, in the following years, most of secure-KNN-based SE schemes cannot be proven secure under the IND-CKA (or, IND-CPA) model, such as [31], [32]. Chosen-plaintext attacks against secure KNN were proposed in [33] and [34]. A remedy is to convert the numeric comparison tests to equality tests like the work in ICDE '17 [37].

3 Secure Boolean Computation and Related Definitions

We present the ideal/real encrypted Boolean function concepts, to accurately describe the query-leakage level of Boolean SE schemes. They will help us to seek a practical and acceptable trade-off between security and overall efficiency.

3.1 Ideal Encrypted Boolean Computation

Definition 3.1 (*Ideal Encrypted Boolean Function*). Let $\prod = (Setup, Trapdoor, Search)$ be an $\mathcal{L}$-adaptively-secure single-round Boolean searchable encryption scheme. Let φ be a Boolean query (Q is its encrypted form). Assuming the user submit the query only one times, we observe the search leakage $\mathcal{L}^{search}(\varphi)$. We say $f = Search$ is an ideal encrypted Boolean function (IEBF) if the search leakage profile $\mathcal{L}^{search}(\varphi)$ can be written as $\mathcal{L}'(Q, DB(\varphi))$ or $\mathcal{L}'(Q, DB^*(\varphi)))$, where $\mathcal{L}'$ is a stateless function. If f is an IEBF, then we call $\prod$ level-1-revealing.

The computation can be expressed as $f(Q, EDB) = DB(\varphi)$. Unfortunately, in reality, it still is a difficulty to build an IEBF, since the query is handled by the cloud itself, and then the cloud learns not only the input and output but also intermediate results because the results are generated step by step. Thus, all Boolean computation will induce leakage that can be learned by the cloud. Although we cannot avoid the leakage, this information can be reduced to an acceptable level.

Note that, the function f is computed in untrusted environments. If the algorithms require the cloud to send Q and EDB back to the client or trusted hardware for computation, it will deviate from the intention of designs. The design goal of IEBF is to make maximum use of cloud resources and reveal as little information as possible.

Theorem 3.1 (*IEBF implementation*). If there are a random oracle H, and an RCPA-secure private-key encryption algorithm Enc, then there exists an IEBF f.

Proof: Given a random oracle H, we use the following naive idea to construct an IEBF. We consider all u-dimensional Boolean queries as single-keyword strings. The data owner precomputes all available Boolean conditions Ψ that can be issued in the future, converts them into single-keyword strings, and puts them into the index. The encrypted index can be written as $\mathbf{I} = \{(H(\varphi), Enc(DB(\varphi)) : for\ all\ \varphi \in \Psi\}$. Now, an encrypted database $EDB = (\mathbf{I}, \mathbf{D})$ is created, and an ideal Boolean function f is constructed, such that for any input φ, we have $f(H(\varphi), EDB) = Enc(DB(\varphi))$.

Obviously, the index size of this scheme turns out to be $\Omega(\sum_i 2^{m_i})$, where m_i is the number of distinct keywords in the i-th data file. Even for supporting two-dimensional queries, the index size is not acceptable. We should make this construction practical.

3.2 Real Encrypted Boolean Computation

We present subquery-privacy concept to describe the query leakage. Let φ be an unencrypted Boolean query (Q is its encrypted form). We call subqueries of φ $subq(\varphi)$, which denotes all possible Boolean queries that consist of query terms which come from the original φ (e.g.,

$subq(a \vee b) = \{a, b, a \wedge b, a \vee b\}$). We call subquery privacy of φ $\partial(\varphi)$=$\{|DB(q)| : for\ all\ q\ \in subq(\varphi)\ and\ q \neq \varphi\}$ (e.g., $\partial(a \vee b) = \{|DB(a)|, |DB(b)|, |DB(a \wedge b)|\}$).

Subquery privacy is useful information that can be utilized by attackers [44]. Almost all single-round Boolean SE schemes partially or fully leak this information. So we need to minimize it if we can do well.

Definition 3.2 (*Real Encrypted Boolean Function*). Let $\prod = (Setup, Trapdoor, Search)$ be an $\mathcal{L}$-adaptively-secure single-round Boolean searchable encryption scheme. Let φ be a Boolean query. Assuming the user submit the query only one times, we observe the search leakage $\mathcal{L}^{search}(\varphi)$. We say $f = Search$ is a real encrypted Boolean function (REBF) if the leakage profile $\mathcal{L}^{search}(\varphi)$ don't contain any subquery privacy $\partial(\varphi)$. If f is a REBF, then we call $\prod$ level-2-revealing.

The REBF notation (level-2-revealing) guarantees that except for the final results, no vital privacy is revealed to the cloud when Boolean queries are issued. REBF is a strong security notation. As far as we know, there are no single-round Boolean SE schemes that achieve level-2-revealing. For example, we cannot construct a REBF from the OXT scheme [26], since when $w_1 \wedge w_2 \wedge w_3$ is issued in OXT, the cloud learns $|DB(w_1 \wedge w_2)|$, $|DB(w_1 \wedge w_3)|$, and $|DB(w_1)|$.

A level-2-revealing solution is a wonderful security-efficiency trade-off for a single-round SE scheme since we can implement a REBF efficiently by many approaches. If the leakage $\mathcal{L}^{search}(\varphi)$ of a single-round $\mathcal{L}$-adaptively-secure SE scheme contains more information than that in a level-2-revealing one, we label the scheme as level-2^{+}-revealing.

4 VBT-1: An Adaptively-secure Conjunctive Scheme

Let's study conjunctive queries first. If a REBF supports only conjunctive queries, we call it a real encrypted conjunctive function (RECF). We now propose an adaptively-secure level-2-revealing conjunctive scheme called VBT-1. The construction consists of three polynomial-time algorithms $VBT\text{-}1 = (Setup, Trapdoor, Search)$, where the algorithms are based on the tree data structure [43] for sub-linear search efficiency.

4.1 Overview of the Design

We logically consider a conjunctive/Boolean query as a set of deterministic finite automatons (DFAs), where each DFA can be used to match a unique keyword set. All data files are put into a tree-based index, where each tree node corresponds to the keyword set. If one DFA can efficiently match the keyword set, all the DFAs can be recursively processed among all the tree nodes.

Encrypting and Running a DFA. We precompute all DFAs and put them into each tree node of the tree-based index. We consider simple DFAs first in this section. A simple DFA is an automaton that has only two or several DFA-states x, y. The owner (user) initially stores $c = w \oplus x$ on the cloud. Thus c has been learned by the cloud, and w and x are hidden. If the user sends a mask $x \oplus y$ to the cloud, the DFA-state x can be obliviously changed to y, yet the cloud learns only $w \oplus x \oplus x \oplus y = w \oplus y$. We refer to $w \oplus x$ and $w \oplus y$ as encrypted DFA-states of x and y, respectively. In such a process, the cloud learns nothing about $\{w, x, y\}$. If we recursively apply this approach, the DFA can be obliviously changed from one state to another. To eliminate correlations of all DFAs, each DFA corresponds to a unique path value. With this approach, we can encrypt and run all DFAs with minimized leakage.

4.2 Index Setup

We first should employ a tree-based index. To introduce the tree structure in [43], we review three related concepts.

Full Binary Tree. A full binary tree is a binary tree with $2^L - 1$ tree nodes and 2^{L-1} leaves, where L (root $L = 1$) is the height of the tree.

Path(v). Let v be a non-terminal tree node. $Path(v)$ is a root-to-node-v path string, where each left-side branch is '0' and right-side branch is '1'.

Nodes(i). Let $i \in [0, 2^{L-1} - 1]$, and $leaf_i$ denote the i-th leaf in the tree. $Nodes(i)$ denotes a set of tree nodes along the root-to-$leaf_i$ path.

Definition 4.1 (*VBTree [43]*). A virtual binary tree (VBTree) is an encrypted full binary tree with the following properties:

1) Each tree node contains zero or more different encrypted keywords.
2) The encrypted tree contains only keywords. The tree nodes and branches are not explicitly stored in the tree.
3) To index a keyword w of the i-th file ($i \in [0, 2^{L-1} - 1]$), the keyword is inserted into each tree node of $Nodes(i)$. The inserted items are $\{E_K(Path(v), w)\}_{v \in Nodes(i)}$, where K is a set of secret keys of the data owner, and E is a deterministic encryption algorithm that takes as input a path value $Path(v)$, a keyword w and K, and outputs a set of binary strings.

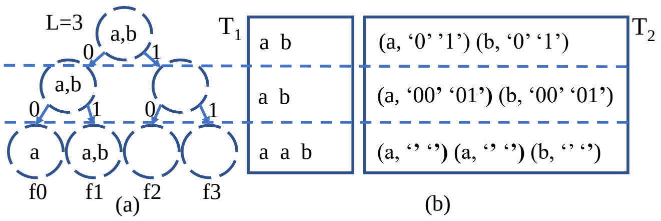

L=3
a,b
0
1
a,b
0
1
0
1
a
a,b
f0
f1
f2
f3
(a)
T1
a b
a b
a a b
(a, '0' '1') (b, '0' '1')
(a, '00' '01') (b, '00' '01')
(a, '' '') (a, '' '') (b, '' '')
T2
(b)

Fig. 1: An example tree

Figure 1(a) shows a logical view of a virtual binary tree with height $L = 3$. $Path(leaf_1)$ denotes string '01'. $Nodes(1)$ is a set of tree nodes of paths {'', '0', '01'}. To locally add a file {'a','b'} at node 'f1', we should insert the keywords into these three tree nodes. Figure 1(b) is a physical view of the tree, where T_1 and T_2 are two hash tables. For easily encrypting the tree, items and branches are stored separately in (T_1, T_2), respectively, where each entry should be identified by the corresponding path string. The entry (a, '00', '01') denotes that the children of the current node are '00' and '01', and so on.

Let $W(v)$ denote a set of keywords in node v. For any nonterminal node v, if v_l is its left child node and v_r is its right child node, we can prove that the keyword set of v are

$W(v) = W(v_l) \cup W(v_r)$. To create a nearly-balanced tree, n files are inserted into the leaves varying from $leaf_0$ to $leaf_{n-1}$.

We define two operations to handle a hash table T for managing the tree: '<<' and '>>'. Let $T << (k, x)$ denote inserting an encrypted key-value pair (k, x) into the hash table T within two steps. Note that, it is **NOT** $T[k] = x$. First, the value k is split into two parts (k_1, k_2) in fixed sizes (k_1 is the high bits, and k_2 is the low bits). Second, let $T[k_1] \leftarrow x \oplus k_2$. Let $T >> (k, x)$ denote retrieving an encrypted key-value pair (k, x) from the hash table T by the key k with the value x returned within two steps. First, the value k is split into two parts (k_1, k_2) in fixed sizes. Second, let $x \leftarrow T[k_1] \oplus k_2$. We use $T << \{(k, x), (k', x'), \cdots\}$ to denote an insertion of a set of key-value pairs.

We now present a concrete instance of the tree. Let F, P, V be three keyed pseudo-random functions, and H_1 and H_2 be two collision-resistant hash functions modeled as random oracles, where $F, P : \{0,1\}^l \times \{0,1\}^* \rightarrow \{0,1\}^l$, $V : \{0,1\}^l \times \{0,1\}^* \rightarrow \{0,1\}^s, H_1 : \{0,1\}^* \rightarrow \{0,1\}^{2l}$, and $H_2 : \{0,1\}^* \rightarrow \{0,1\}^{3l}$. Let the left child of the tree node v be v_l, and the right child of the tree node v be v_r. A virtual binary tree has been put into two hash tables T_1 and T_2. We define the above function $E_K(Path(v), w)$ as the following algorithm. To index a keyword w at a non-terminal tree node v, the data owner uses two operations: $T_1 << (H_1(F_K(w||1) \oplus P_K(Path(v))), F_K(w||2) \oplus P_K(Path(v)))$, and $T_2 << (H_2(F_K(w||3) \oplus P_K(Path(v))), ((F_K(w||4) \oplus P_K(Path(v_l)));$
$(F_K(w||4) \oplus P_K(Path(v_r)))))$. If the tree node is a leaf, we put the corresponding encrypted identifier ID_i into the leaf as a result, i.e., $T_2 << (H_2(F_K(w||3) \oplus P_K(Path(v))), (ID_i; 0))$, where the concatenated zero denotes the leaf reached, and ID_i is the i-th identifier that is encrypted by (Enc, Dec), a CPA-secure private-key scheme.

VBT-1.Setup gives the pseudo-code of building an encrypted index for a set of files based on the tree. Given a set of n files, the algorithm outputs an encrypted index, where $File(i)$ denotes a set of distinct keywords of the i-th file, id_i denotes the plain-text identifier of the i-th file with ID_i the encrypted form.

The index size of VBT-1 is $O(NL) \approx O(N \log n)$, and the construction time is $O(NL) \approx O(N \log n)$. The index size of T_2 is slightly larger than that of T_1. Let W denote a set of keywords that can be queried. We assume $N = \sum_{w \in W} |DB(w)|$. In the best case, if all of the files are the same (except for the file identifiers), the index size is $O(N)$. In the worst case, if all of the keywords in all documents are different, the index size is $O(NL) \approx O(N \log n)$.

VBT-1.Setup()

```
1: initialize $T_1 = \{\}; T_2 = \{\}$
2: for $i$=0 to $n-1$ do
3:     for all $w$ in $File(i)$ do
4:         for all $v$ in $Nodes(i)$ do
5:             $k_1 \leftarrow H_1(F_K(w||1) \oplus P_K(Path(v)))$
6:             $x \leftarrow F_K(w||2) \oplus P_K(Path(v))$
7:             insert $T_1 << (k_1, x)$
8:             $k_2 \leftarrow H_2(F_K(w||3) \oplus P_K(Path(v)))$
9:             if $v$ is a leaf then
10:                $ID_i \leftarrow Enc_K(id_i; V_K(id_i||w))$
11:                insert $T_2 << (k_2, (ID_i; 0))$
12:            else
13:                randomly generate a bit, $b \xleftarrow{\$} \{0,1\}$
14:                if $b$ is 1 then
15:                    $(t_1; t_2) \leftarrow P_K(Path(v_l))||P_K(Path(v_r))$
16:                else
17:                    $(t_1; t_2) \leftarrow P_K(Path(v_r))||P_K(Path(v_l))$
18:                end if
19:                $y \leftarrow (F_K(w||4) \oplus t_1; F_K(w||4) \oplus t_2)$
20:                insert $T_2 << (k_2, y)$
21:            end if
22:        end for
23:    end for
24: end for
25: output $I = (T_1, T_2)$
```

4.3 Trapdoor Computation

To search over the encrypted index, the data user creates a trapdoor and sends it to the cloud. The trapdoor is a set of encrypted conditions constructed from the query. Let p be a string of $L - \lceil \log n \rceil - 1$ zeros to denote the start path of the search process (we assume the user learns the value n, i.e., the number of files outsourced to the cloud). Note that, the search begins from the node of path p instead of the root. Given a u-dimensional conjunctive keyword query $q = w_1 \wedge w_2 \wedge \cdots \wedge w_u$, the trapdoor is $T(q) = (ft, it, bt)$, whose first element is $ft = F_K(w_1||1) \oplus P_K(p)$, whose second element is $it = \{F_K(w_i||2) \oplus F_K(w_{i+1}||1)\}_{i \in [1, u-1]}$, and whose third element is $bt = \{bt_1, bt_2\} = \{F_K(w_u||2) \oplus F_K(w_u||3), F_K(w_u||4) \oplus F_K(w_1||1)\}$.

Forward/intersection/backward tokens. The forward token ft is an encrypted block to get the next token that can be XORed with the intersection token. The intersection token it is a set of tokens that are used to test whether the keywords are in the current tree node or not. The backward token bt is the encrypted blocks for accessing both the left and right child tree nodes of the current tree node. The whole trapdoor consists of these three parts: forward token ft, intersection token it, and backward token bt. Any matched final results depend on these three tokens. Without any part of the trapdoor, the result cannot be correctly computed.

4.4 Searching over VBT-1

There are two kinds of operations on the tree: search processing at a non-terminal tree node, and search processing at a leaf. ***VBT-1.Search*** recursively invokes the sub-algorithms Search and Test to traverse over a tree with the search beginning on the tree node that is designated by the user.

At a non-terminal node v, the search process test whether the first term w_1 is in the tree node by checking $T_1 >> (H_1(ft), x)$. If w_1 is not in this node, x will be nothing, which means the search algorithm returns 'not-found' immediately, otherwise, the value x can be XORed by each individual intersection token. If x is XORed by an individual intersection token, a new forward token $x \oplus it_i$, which is the forward token of w_2, will be generated, where it_i is the i-th intersection token. Repeatedly applying this, the cloud can learn whether the keywords $\{w_2, w_3, \cdots, w_u\}$ are in the current tree node or not. If one of the keywords is not in this tree node, the search returns 'not-found' immediately. If they all exist in this tree node, the cloud takes the lastly returned value of the term w_u, i.e, $T_1 >> (H_1(y), z)$, where y is the forward token of w_u, and z is the lastly returned value of all intersection tokens. The value z will be used to access the hash table T_2 for searching for the left-side or right-side tree node. If z is XORed by the first element of the backward token bt_1, i.e., $z' \leftarrow (z \oplus bt_1)$, the cloud now has the token z' to access the hash table T_2. The search algorithm runs $T_2 >> (H_2(z'), (t_1; t_2))$, and the value $(t_1; t_2)$ will be outputted, where t_1 and t_2 can also be XORed by bt_2. We now have $ft_1 \leftarrow t_1 \oplus bt_2$, and $ft_2 \leftarrow t_2 \oplus bt_2$. In fact, the values $\{ft_1, ft_2\}$ are the next forward tokens of w_1 to access the left and right (or the right and left) subtrees of the current tree node, respectively. The cloud uses tokens (ft_1, it, bt) and (ft_2, it, bt) as new trapdoors to recursively traverse the subtrees.

If the search process reaches a leaf, which can be checked by testing whether the value ft_2 is zero or not, the search algorithm considers $ID_i \leftarrow ft_1$ as an encrypted file identifier and outputs a result. The cloud recursively runs this, until all results are matched.

We note that given a query φ to search over the tree, an encrypted result ID_i at a leaf node is outputted, if and only if the accessed leaf matches the query φ. There are three occasions. 1) If a search completes in a non-terminal node, it cannot get the final result certainly, because all subtree nodes are encrypted by the information that comes from their father nodes. 2) Consider a search reaches a leaf node, assuming the query doesn't match the leaf. So the search will complete in one of the intersection token processing. The search doesn't have the encrypted key to access T_2 to get the final encrypted result ID_i, because ID_i is encrypted by the backward token and the prior information. 3) Consider a search reaches a leaf, and the query also matches the leaf. Certainly, ID_i can be correctly outputted. In this occasion, the accessed path forms a decrypted linked list beginning from the root to the leaf.

```
//User
VBT-1.Trapdoor($\varphi = w_1 \wedge w_2 \cdots \wedge w_u$)
 1: $ft \leftarrow F_K(w_1||1) \oplus P_K(p)$
 2: $it \leftarrow \{F_K(w_i||2) \oplus F_K(w_{i+1}||1)\}_{i\in[1,u-1]}$
 3: $bt \leftarrow \{F_K(w_u||2) \oplus F_K(w_u||3), F_K(w_u||4) \oplus F_K(w_1||1)\}$.
 4: build $T(\varphi) = (ft, it, bt)$ and send it to the cloud
//Cloud
VBT-1.Search($T(\varphi)$; $T_1, T_2$)
 1: parse $T(\varphi)$ as $(ft, it, bt)$.
 2: invoke $(b_i, ft_1, ft_2)$←Test($(ft, it, bt)$; $T_1, T_2$)
 3: if $b_i$=false, then return 'not found'
 4: if the current node is a leaf by checking $ft_2$, then parse $ft_1$ as a file
    identifier $id$ and return one encrypted result.
 5: invoke, Search($(ft_1, it, bt)$; $T_1, T_2$)
 6: invoke, Search($(ft_2, it, bt)$; $T_1, T_2$)
//Cloud
VBT-1.Test($(ft, it, bt)$; $T_1, T_2$)
 1: read $x$: $T_1 >> (H_1(ft), x)$
 2: if $x$ is empty, return $(false, 0, 0)$
 3: for all $it_i \in it$ do
 4:     $y \leftarrow x \oplus it_i$
 5:     read $z$: $T_1 >> (H_1(y), z)$
 6:     if $z$ is empty, return $(false, 0, 0)$
 7:     $x \leftarrow z$
 8: end for
 9: parse $bt$ as $(bt_1, bt_2)$
10: $z' \leftarrow z \oplus bt_1$
11: read $(t_1; t_2)$, i.e., $T_2 >> (H_2(z'), (t_1; t_2))$
12: if $(t_1; t_2)$ is empty, return $(false, 0, 0)$
13: compute $(ft_1, ft_2) \leftarrow (t_1 \oplus bt_2, t_2 \oplus bt_2)$
14: return $(true, ft_1, ft_2)$
```

4.5 Correctness Analysis

Let's consider an example in a tree node. We assume the tree node of path p contains keywords 'a' and 'b'. If the cloud runs a query $a \wedge b$, we now show Test Procedure how to work. The trapdoor of $a \wedge b$ is $\{F_K(a||1) \oplus P_K(p), \{F_K(a||2) \oplus F_K(b||1)\}, \{F_K(b||2) \oplus F_K(b||3), F_K(b||4) \oplus F_K(b||1)\}\}$. The cloud first searches on T_1 by $F_K(a||1) \oplus P_K(p)$, and $x \leftarrow F_K(a||2) \oplus P_K(p)$ will be returned. Then, the cloud computes $x \oplus it[0]$=$F_K(a||2) \oplus P_K(p) \oplus (F_K(b||1) \oplus F_K(a||2))$=$F_K(b||1) \oplus P_K(p)$, which is the forward token of keyword 'b'. The cloud runs $T_1 >> (H_1(F_K(b||1) \oplus P_K(p)), z))$, and z=$F_K(b||2) \oplus P_K(p)$ will be returned. The value z can be XORed by bt_1, i.e., $z' = (z \oplus bt_1) = F_K(b||2) \oplus P_K(p) \oplus (F_K(b||2) \oplus F_K(b||3)) = F_K(b||3) \oplus P_K(p)$. Now, the cloud has the token z' to access the hash table T_2. The cloud searches on T_2, by $T_2 >> (H_2(z'), (t_1; t_2))$, and $(t_1; t_2)$=$(F_K(b||4) \oplus P_K(p_1); F_K(b||4) \oplus P_K(p_2))$ will be returned. The cloud computes $(t_1 \oplus bt_2, t_2 \oplus bt_2)$=$(F_K(b||4) \oplus P_K(p_1) \oplus F_K(b||4) \oplus F_K(a||1), F_K(b||4) \oplus P_K(p_2) \oplus F_K(b||4) \oplus F_K(a||1))$=$(F_K(a||1) \oplus P_K(p_1), F_K(a||1) \oplus P_K(p_2) = (ft_1, ft_2)$. They are two new forward tokens to search on its two subtrees respectively.

The above computation can be viewed as a DFA that has only two transition states, '0' and '1'. If the DFA matches the tree node, the accepted state is '1', otherwise '0'. We will extend this DFA in the next sections.

The forward token has two purposes. One is to test whether a keyword is in a tree node or not, and the other is to generate the next forward token. If an intersection token is XORed by the output of the forward token, a new forward token will be generated. Thus, the cloud learns whether all queried conjunctive terms exist in this tree node or not. To access the subtrees, the cloud requires two new forward tokens, which can be created from the output of the final intersection token and the backward token. The first element of the backward token enables the cloud to access T_2, and the second element is for accessing the subtrees.

4.6 Search Complexity

The search complexity is $O(u \min_{w\in\varphi} |DB(w)| \log n)$, where $\varphi = w_1 \wedge w_2 \wedge \cdots \wedge w_u$ is a conjunctive query. The query time consists of two parts: one is for traversing all target tree nodes, and the other is the query time in each tree node.

To get all the final results, in the worst case, the search process will traverse the tree nodes whose size is the smallest among all the individual queries $\{O(|DB(w_i)| \log n)\}_{i\in[1,u]}$, because if a tree node doesn't match the conjunctive query, the search procedure of the current subtree will return immediately. In reality, since $\min_{w\in\varphi} |DB(w)| \ll n$, the conjunctive query time is sub-linear.

5 VBT-2: An adaptively-secure Disjunctive SE Scheme

In PBTree [11] and IBTree [25], they convert the numeric comparison tests to equality tests and consider a numeric range query as a disjunctive query. We now study the weakness of range queries and show how to build an adaptively-secure level-2-revealing disjunctive SE scheme.

5.1 Statistical Information of Range SE Schemes

A numeric range query can be considered as a disjunctive query using prefix encoding [11]. According to their design, a number is converted to a set of prefix strings, and a numeric range query is converted to a disjunctive query. Their target is to use this range to match the stored encrypted prefix strings. For example, number 6 is considered as {'0011','001*','00**','0***','****'}, where each string denotes a range string that matches number 6. Given a range query, say [0,8], it is converted into {'0***','1000'}, where '0***' means a range $[0,7]$ and '1000' means a hex string of 8. The value 6 will be matched by $[0,8]$ correctly since they share the common string '0***'.

We use $R(e)$ to denote a set of prefix strings that are converted from the integer e, where $e \in [0,B]$, and use $R([a,b])$ to denote a set of prefix strings that are converted from the range $[a,b]$. We can conclude that for any e, e is in $[a,b]$, if and only if $R(e) \bigcap R([a,b]) \neq \varnothing$ [11].

The problem of the above schemes, such as PBTree and IBTree, is that they are level-2^+-revealing. They leak all individual queries (i.e., DB('0***') and DB('1000')), and the cloud learns the main factor that causes the final result (i.e., DB('0***')). The more disastrous thing is that the cloud learns $\frac{|DB(\text{'0***'})|}{|DB(\text{'1000'})|}$, which is exactly 8, only if the distribution of the result set is uniform.

Generally, given a range query $[a,b]$, we define its one-dimensional statistical feature as

$$SF([a,b]) = \left\{ \frac{|DB(w_1)|}{\epsilon}, \cdots, \frac{|DB(w_t)|}{\epsilon} \right\},$$

where ϵ=min $|DB(w_i)|_{i\in[1,t]}$, $\epsilon \neq 0$, and $R([a,b]) = \{w_1, \cdots, w_t\}$. Although the cloud learns nothing about $[a,b]$, it has $SF([a,b])$, which perhaps leads to a severe breakage [16], [21], [39], [44]. So we need to eliminate it if we can do well.

5.2 Overview of VBT-2

Compared with conjunctions, it is harder to protect subquery privacy of disjunctive queries, since the result set of disjunctions, generally, is larger than that of conjunctive queries. If we use the same indexing algorithm as VBT-1 for disjunctive queries, the result cannot be computed with level-2-revealing. The problem is that this will enable the cloud to learn the selectivity of each disjunction. We address this problem by trading storage complexity for more privacy.

We now show how to protect the subquery privacy of the disjunctive or range queries. We should modify the indexing algorithm in VBT-1 and the indexing elements in each tree node because, in VBT-1, the DFA has only one transition state (except the accepted state). In VBT-2, given a tree node with path p, the DFA has two transition states $P_K(p||0)$, which logically denotes '0', and $P_K(p||1)$, which logically denotes '1'. If the cloud can compute DFAs '$0 \vee 0 = 0$', '$0 \vee 1 = 1$', '$1 \vee 0 = 1$', and '$1 \vee 1 = 1$', without learning '1' or '0', the cloud will handle more complex disjunctive queries based on these operations. Our approach is to precompute all related transition states and store them in each tree node.

5.3 Indexing Disjunctive Elements

We first give the keyword indexing algorithm for a tree node. A DFA-state is an encrypted block constructed from a node path and a value of true or false. Given a tree node v with path $p = Path(v)$, the value that denotes false in this node is encoded as $P_K(p||0)$, called state-0, and the value that denotes true in this node is encoded as $P_K(p||1)$, called state-1. Given a keyword w, it has five encrypted states $F_K(w||0)$, $F_K(w||1)$, $F_K(w||2)$, $F_K(w||3)$ and $F_K(w||4)$. A keyword state is an encrypted block constructed from the keyword and its sequence number. If w is in this tree node, we insert four values into the hash table, $T_1 << \{(k_{00}, x_0), (k_{01}, x_1), (k_{10}, x_1), (k_{11}, x_1)\}$, where $k_{ij} \leftarrow H_1(F_K(w||i) \oplus P_K(p||j))$ $(i,j \in \{0,1\})$, and $x_i \leftarrow F_K(w||2) \oplus P_K(p||i)$ $(i \in \{0,1\})$. (k_{10}, x_1) means that state-0 is altered to state-1 when w is in this tree node. (k_{11}, x_1) means that state-1 is still not changed when w is in this tree node. (k_{00}, x_0) and (k_{01}, x_1) mean the operations to initialize the keyword states. Similarly, if w is not in this tree node, we should still insert four values into the hash table, $T_1 << \{(k_{00}, x_0), (k_{01}, x_0), (k_{10}, x_0), (k_{11}, x_1)\}$. (k_{10}, x_0) means that state-0 is not changed when w is not in this tree node. (k_{11}, x_1) means that state-1 is not changed even if w is not in this tree node. (k_{00}, x_0) and (k_{01}, x_0) mean state initializing. Suppose the left child of v is v_l and the right child is v_r, then the state-1s of v_l and v_r are $t_1 \leftarrow P_K(Path(v_l)||1)$ and $t_2 \leftarrow P_K(Path(v_r)||1)$, respectively. For accessing the child tree nodes, we should insert items related to w into the hash table T_2 like the procedure in VBT-1, i.e., $T_2 << (k_2, y)$, where $k_2 \leftarrow H_2(F_K(w||3) \oplus P_K(p||1))$ and $y \leftarrow (F_K(w||4) \oplus t_1; F_K(w||4) \oplus t_2)$ or $y \leftarrow (F_K(w||4) \oplus t_2; F_K(w||4) \oplus t_1)$. In the special case, if v is a leaf, we do the same work like in VBT-1.

Next, given the i-th file, for any tree node $v \in Nodes(i)$, and any keyword $w \in W$, we use the above algorithm to process all keywords. Repeat this procedure until all files have been processed. The pseudo-code is shown in ***VBT-2.Setup***.

Let m denote the dictionary size (i.e., $m = |W|$). Since all tree items are nearly balanced, the index size can be considered as $O(mn)$, which is a trade-off between query privacy and storage overhead.

5.4 Searching over VBT-2

The searching algorithm is the same as that in VBT-1. The user builds a search trapdoor (ft, it, bt) for $w_1 \vee \cdots \vee w_u$ and sends it to the cloud. The procedure will output a set of encrypted file identifiers that match the disjunctive query.

Let 's consider an example. Let p be the start path. Suppose keyword b exists only in the first file and keywords a and c are not exist. If the user queries $a \vee b \vee c$, the trapdoor is (ft, it, bt), where ft=$F_K(a||0) \oplus P_K(p||1)$, it=$\{F_K(a||2) \oplus F_K(b||1), F_K(b||2) \oplus F_K(c||1)\}$, and $bt = \{F_K(c||2) \oplus F_K(c||3), F_K(c||4) \oplus F_K(a||0)\}$. The cloud first computes $T_1 >> (ft, x)$, and $x = F_K(a||2) \oplus P_K(p||0)$ will be outputted. Since a is not in this tree node, x is related to state-0. Next, the cloud computes $x \oplus it[0] = F_K(a||2) \oplus$

$P_K(p||0) \oplus F_K(a||2) \oplus F_K(b||1)$=$F_K(b||1) \oplus P_K(p||0)$, which is the forward token of b. The cloud computes $T_1 >> (F_K(b||1) \oplus P_K(p||0), x')$, and $x' = F_K(b||2) \oplus P_K(p||1)$ will be outputted. x' is related to state-1 and it can still be XORed by $it[1]$. The cloud computes $T_1 >> (F_K(c||1) \oplus P_K(p||0), x'')$, and $x'' = F_K(c||2) \oplus P_K(p||1)$ will be outputted. x'' is related to state-1. With x'' and bt, the cloud searches T_2, and it will get two new forward tokens of keyword a for both subtrees. The cloud can recursively apply this algorithm.

In the above searching process, the DFA-states do not exist alone, and they are always encrypted by mask values, which are keyword states. The keyword states still do not exist alone. Therefore, the cloud learns nothing about which term is the main factor that causes the result from a query. In the above example, the cloud learns nothing about $DB(a) = \varnothing$, $|DB(b)| = 1$ and $DB(c) = \varnothing$, and it learns only $|DB(a \vee b \vee c)| = 1$.

The search complexity of VBT-2 is $O(u|DB(\varphi)| \log n)$.

```
VBT-2.Setup()
initialize $T_1 = \{\}; T_2 = \{\}$
for $i$=0 to $n-1$ do
    for all $w \in W$ do
        for all $v \in Nodes(i)$ do
            $k_{00} \leftarrow H_1(F_K(w||0) \oplus P_K(Path(v)||0))$
            $k_{01} \leftarrow H_1(F_K(w||0) \oplus P_K(Path(v)||1))$
            $k_{10} \leftarrow H_1(F_K(w||1) \oplus P_K(Path(v)||0))$
            $k_{11} \leftarrow H_1(F_K(w||1) \oplus P_K(Path(v)||1))$
            $x_0 \leftarrow F_K(w||2) \oplus P_K(Path(v)||0)$
            $x_1 \leftarrow F_K(w||2) \oplus P_K(Path(v)||1)$
            if $w \in File(i)$ then
                $T_1 << \{(k_{00}, x_0), (k_{01}, x_1), (k_{10}, x_1), (k_{11}, x_1)\}$
            else
                $T_1 << \{(k_{00}, x_0), (k_{01}, x_0), (k_{10}, x_0), (k_{11}, x_1)\}$
            end if
            let $k_2 \leftarrow H_2(F_K(w||3) \oplus P_K(Path(v)||1))$
            if v is a leaf then
                $ID_i \leftarrow Enc_K(id_i; V_K(id_i||w))$
                insert $T_2 << (k_2, (ID_i; 0))$
            else
                randomly generate a bit, b$\xleftarrow{\$}$ $\{0,1\}$
                if b is 1 then
                    $t_1 \leftarrow P_K(Path(v_l)||1)$
                    $t_2 \leftarrow P_K(Path(v_r)||1)$
                else
                    $t_1 \leftarrow P_K(Path(v_r)||1)$
                    $t_2 \leftarrow P_K(Path(v_l)||1)$
                end if
                $y \leftarrow (F_K(w||4) \oplus t_1; F_K(w||4) \oplus t_2)$
                insert $T_2 << (k_2, y)$
            end if
        end for
    end for
end for
output $I = (T_1, T_2)$
```

```
//User
VBT-2.Trapdoor($\varphi = w_1 \vee w_2 \cdots \vee w_u$)
$ft \leftarrow F_K(w_1||0) \oplus P_K(p||1)$
$it \leftarrow \{F_K(w_i||2) \oplus F_K(w_{i+1}||1)\}_{i \in [1,u-1]}$
$bt \leftarrow \{F_K(w_u||2) \oplus F_K(w_u||3), F_K(w_u||4) \oplus F_K(w_1||0)\}$.
build $T(\varphi) = (ft, it, bt)$ and send it to the cloud
//Cloud
VBT-2.Search($T(\varphi)$; $T_1, T_2$)
invoke, VBT-1.Search($T(\varphi)$; $T_1, T_2$)
```

5.5 Level-2-revealing Range queries

We now do the attractive thing: level-2-revealing numeric range queries. Recall that a range query can be considered as a disjunctive query. We first convert all integers that exist in files into prefix keywords and put them into the index. Next, we can use range queries to search over the outsourced database.

Given a range query $[a, b]$, in a level-2^+-revealing disjunctive SE scheme, $SF([a, b])$ contains much information, yet in VBT-2, $SF([a, b])$=$\varnothing$ (if $|R([a, b])| \geq 2$).

6 VBT-3: An adaptively-secure Boolean SE Scheme

6.1 Overview of VBT-3

Given a tree node with path p, the DFA has many transition states with the form of $P_K(p||x)$, which denotes state-x. State-1 means '1', and state-0 means '0', and etc. If the cloud can compute simple Boolean expressions, such as '$(0 \vee 0) \wedge (1 \vee 0) = 0$', '$(0 \vee 1) \wedge (1 \vee 1) = 1$', without leaning '1' or '0', the cloud will obliviously handle more complex Boolean queries based on these operations.

Like in VBT-2, our approach is to precompute all transition states that will be used in the future and store them in each tree node. For simplicity, we assume the queries are in conjunctive normal form (CNF). If a CNF formula consists of only 0, 1, parenthesis, $\wedge$, and $\vee$, we call it a simple CNF.

A simple CNF formula is a deterministic finite automaton (DFA) that consists of a 5-tuple, $(S, \sum, \delta, s_0, s_a)$, where S is a finite set of DFA-states; $\sum$ is a finite set of input symbols called the alphabet; δ is a transition function, δ:$S \times \sum \rightarrow S$; s_0 is an initial state, $s_0 \in S$; and s_a is a set of accepted states, $s_a \subseteq S$. An encrypted Boolean computation is, in fact, an array of encrypted DFA computations. Table 1 shows the transition table of the DFA, where S={0,1,'0∧(0','0∧(1','1∧(0','1∧(1'} , $\sum$={'0','1','∨0','∨1', '∨0)', '∨1)', '∧(0', '∧(1' }, s_0 is 0 or 1, and $s_a = \{0, 1\}$. For example, δ**(**'1∧(0','∨0)'**)**=0, and δ**(**'1∧(0','∨0'**)**='1∧(0'. To reduce the index size, this table shows only the simplified states. The query $x \wedge y$ is converted into $x \wedge (y \vee y)$ to be suitable for this table. For easily expressing keyword states, we write a DFA transition as $d = \delta(a, (b, c))$, where a is a DFA-state, b is a string that is related to a keyword, d is a result state, and c is 1 or 0, which denotes the keyword of the symbol is in the current tree node or not. (b, c) will output a valid symbol.

6.2 Precomputing and storing all DFAs

To run the above DFAs obliviously, for each tree node and each keyword, we precompute $|S||\sum| - 20$=28 key-value pairs (referring to Table 1) and put them into T_1 like the approach mentioned in VBT-2. The sub-procedure of the setup is shown in ***VBT-3.Indexword***, where $F_K(b_w||1)$ denotes a keyword state for keyword w to precompute the DFAs, and b_w is a string consisting of an operator, a bracket or the keyword. The function $Replace$ is a string-replacing algorithm (e.g., $Replace(b,$'#'$, c)$ ='∨1' if b ='∨#' and 'c=1'). The pseudo-code to update T_2 is the same as that in VBT-2. For each keyword w in W, each file i, and each node v in Nodes(i), the setup procedure repeatedly invokes $Indexword(i, w, v)$ to build the index. The index size of VBT-3 is $O(\beta mn)$, where $\beta = |S||\sum|$.

One symbol in $\{0, 1\}$ denotes an operation to initialize a DFA-state. The other symbols are used to run the CNF query. In the simple DFA, the operators such as ∧,∨, and ∧) are all considered as parts of keywords. Thus, a Boolean query is converted into an array of keywords first. This conversion will help us to hide all the Boolean operators. Note that, an invalid state is ignored since a CNF query doesn't have such a state (e.g., '$0 \vee (0'$).

6.3 Level-2-revealing Boolean Queries

Consider a CNF query $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_u$, where each disjunction $\varphi_i = w_{i,1} \vee \cdots \vee w_{i,l(i)}$. The user first serializes φ into an array of keywords $(w_1, \cdots, w_t)$. For example, $(a \vee b) \wedge (c \vee d)$ is serialized into ('a','∨b','∧(c','∨d)'). Meanwhile, remove all operators and generate an array of keywords $(a_1, \cdots, a_t)$ (e.g., ('a','b','c','d')). Let p be the start path. Next, let $ft \leftarrow F_K(w_1||1) \oplus P_K(p||1)$, let $it \leftarrow \{F_K(w_k||2) \oplus F_K(a_{k+1}||1)\}_{k\in[1,t-1]}$, and let $bt \leftarrow \{F_K(w_t||2) \oplus F_K(w_t||3), F_K(w_t||4) \oplus F_K(w_1||1)\}$. Third, build $T(\varphi) = (ft, it, bt)$ and send it to the cloud. Fourth, the cloud invokes VBT-1.Search($T(\varphi); T_1, T_2$). Its search efficiency is $O(u \min_{i\in[1,u]} |DB(\varphi_i)| \log n)$.

We show an example to run a Boolean query $\varphi = (a \vee b) \wedge (c \vee d)$ in a tree node of path p, assuming there logically exists a DFA for this node. Suppose a and c are not in this tree node, and b and d are in this tree node. For simplicity, let $b^* = `\vee b'$, $c^* = `\wedge (c'$, and $d^* = `\vee d)'$. φ is split into four parts (a, b^*, c^*, d^*). The trapdoor is (ft, it, bt), where $ft = F_K(a||1) \oplus P_K(p||1)$, $it = \{F_K(a||2) \oplus F_K(b^*||1), F_K(b||2) \oplus F_K(c^*||1), F_K(c||2) \oplus F_K(d^*||1)\}$, and $bt = \{F_K(d||2) \oplus F_K(d||3), F_K(d||4) \oplus F_K(a||1)\}$. First, the cloud computes $T_1 >> (ft, x)$, and then it has $x = F_K(a||2) \oplus P_K(p||0)$. The DFA-state is initialized with $P_K(p||0)$ (state-0), since a is not in the node. It computes $x \oplus it[0]$=$F_K(b^*||1) \oplus P_K(p||0)$=$ft_b$. Second, it computes $T_1 >> (ft_b, x')$, and x'=$F_K(b||2) \oplus P_K(p||1)$ will be yielded, since b is in this tree node. The DFA-state is obliviously changed from $P_K(p||0)$ to $P_K(p||1)$ (state-1). Third, it computes $x' \oplus it[1] = F_K(b||2) \oplus P_K(p||1) \oplus F_K(b||2) \oplus F_K(c^*||1) = F_K(c^*||1) \oplus P_K(p||1) = ft_c$. Third, it computes $T_1 >> (ft_c, x'')$, and it has $x'' = F_K(c||2) \oplus P_K(p||`1 \wedge (0')$. The DFA-state is changed from $P_K(p||1)$ to $P_K(p||`1 \wedge (0')$ (state-$`1 \wedge (0'$) . It computes $x'' \oplus it[2]$=$F_K(d^*||1) \oplus P_K(p||`1 \wedge (0')$=ft_d. Fourth, it computes $T_1 >> (ft_d, x''')$, and it has x'''=$F_K(d||2) \oplus P_K(p||1)$. The DFA-state is changed from $P_K(p||`1 \wedge (0')$ to $P_K(p||1)$ (state-1). With bt and x''', the cloud can access T_2 and the subtrees like in VBT-1.

A DFA-state can be obliviously changed from one state to another by a symbol. Repeatedly run the transitions, until reaching a final accepted state. The final accepted state is 1 or 0, which denotes the query matches this tree node or not. In the above process, all transition states are hidden from the cloud, and the cloud learns only the encrypted accepted state.

VBT-3.Indexword(i,w,v)

```
 1: if $w \in File(i)$, then $c \leftarrow 1$, otherwise $c \leftarrow 0$
 2: initialize $A = \{`\#', \vee\#', `\vee \#)', `\wedge (\#'\}$.
 3: **for** all $a \in S$ **do**
 4:     **for** all $b \in A$ **do**
 5:         replace # with $c$ in $b$, i.e.,
            $b' \leftarrow Replace(b, `\#', c)$.
 6:         $d \leftarrow \delta(a, b')$; if $d$ is empty, continue;
 7:         replace string # with $w$ in $b$,
            i.e., $b_w \leftarrow Replace(b, \#, w)$
 8:         $k_1 \leftarrow H_1(F_K(b_w||1) \oplus P_K(Path(v)||a))$
 9:         $x \leftarrow F_K(w||2) \oplus P_K(Path(v)||d)$
10:         insert $T_1 << (k_1, x)$
11:     **end for**
12: **end for**
13: Insert items into $T_2$; the same as the lines 16∼31 of VBT-2.Setup
```

TABLE 1: State transition table of simple CNF computations.

S \ $\sum$	0	1	∨0	∨1	∨0)	∨1)	∧(0	∧(1
0	0	1	0	1	-	-	0∧(0	0∧(1
1	0	1	1	1	-	-	1∧(0	1∧(1
0∧(0	-	-	0∧(0	0∧(1	0	0	-	-
0∧(1	-	-	0∧(1	0∧(1	0	0	-	-
1∧(0	-	-	1∧(0	1∧(1	0	1	-	-
1∧(1	-	-	1∧(1	1∧(1	1	1	-	-

The symbol '-' denotes an invalid state.

7 Security Analysis

7.1 Leakage Function

To formally describe the leakage of the schemes, we analyze the search and access patterns. Let **Q** denote a two-dimensional array that stores all historical Boolean queries issued in order of arrival, e.g., **Q**[0][0]='a', **Q**[0][1]='∨b', **Q**[0][2]='∧(c', and **Q**[0][3]='∨d)' for the 0-th Boolean query $(a \vee b) \wedge (c \vee d)$. Given a Boolean query φ, the set of all tree nodes that the search process needs to traverse is denoted by $tn(\varphi)$. The search pattern of a query consists of four parts: $SP_1(\varphi) = \{i : for\ all\ \mathbf{Q}[i][0] = \varphi[0]\}$, $SP_2(\varphi) = \{(i, j, k) : for\ all\ \mathbf{Q}[i][j] = \varphi[k]\ and\ \mathbf{Q}[i][j+1] = \varphi[k+1]\}$, $SP_3(\varphi) = \{i : for\ all\ \mathbf{Q}[i].last = \varphi.last\}$, and $SP_4(\varphi) = \{(i, j, k, v) : for\ all\ \mathbf{Q}[i][j] = \varphi[k]\ and\ v \in tn(\mathbf{Q}[i])\}$, where $\varphi.last$ denotes the last item in this array. In fact, SP_1 is due to the forward token, SP_2 is due to the intersection token, SP_3 is due to the backward token, and SP_4 is due to all the tokens. The access pattern is some information that is used for accessing the index and the encrypted files. It includes two parts. The access pattern for a node v is $AP(\varphi, v)$=$\{(dfa_v, x)$: if φ matches v then $x \leftarrow 1$, else $x \leftarrow 0\}$, where dfa_v denotes the final encrypted accepted DFA-state of node v, and the access pattern for the encrypted files is $DB^*(\varphi)$. Note that $\{AP(\varphi, v)\}_{v\in tn(\varphi)}$ means the query will reveal all DFA encrypted-accepted-states (not accepted-states) of all accessed tree nodes $tn(\varphi)$.

We now have the leakage function of VBT-3: $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}^{Setup}, \mathcal{L}^{Search})$, where
$\mathcal{L}_1 = \mathcal{L}^{Setup}(\mathbf{I}, \mathbf{D}) = (M_1, M_2, L, n, sizes, ids)$, and
$\mathcal{L}_2 = \mathcal{L}^{Search}(T(\varphi)) =$
$(\{SP_1, SP_2, SP_3, SP_4\}, \{AP(\varphi, v)\}_{v\in tn(\varphi)}, DB^*(\varphi))$.

Given an encrypted tree $\mathbf{I} = (T_1, T_2)$ and a set of encrypted files D, $\mathcal{L}_1$ leakage contains (M_1, M_2), i.e., the number of entries in the hash tables (T_1, T_2) respectively, L, i.e., the height of the tree, and n, i.e., the number of files. $sizes$ and ids are size information and identifiers respectively, which come from the set of files encrypted by a CPA-secure scheme.

Given a Boolean query $T(\varphi)$, $\mathcal{L}_2$ leakage function outputs the search and access patten. The length of Boolean queries u is also in this leakage. $\mathcal{L}_2$ leakage is unavoidable in a single-round SE scheme. Note that, SP_2 and SP_4 are very small compared with $SP(\varphi) = \{(i,j,k) : for\ all\ \mathbf{Q}[i][j] = \varphi[k]\}$ in [25] and [43]. $AP(\varphi, v)$ is also very small compared with $\{DB(w)\}_{w\in\varphi}$ in [25] and [43], whose subquery privacy is not well protected. Another notice is that the real path value of v doesn't equal $Path(v)$ since the real paths are disordered when the index is initialized.

7.2 Leakage Analysis

We now show how VBT-3 achieves minimized leakage. There are three occasions in the searching process. The search completes in a non-terminal tree node. The algorithm completes in a leaf, but it doesn't match the leaf. The algorithm completes in a leaf, and it matches the leaf. Only on the last occasion, as mentioned in VBT-1, can the encrypted results be correctly outputted. Since the search path from the root to the leaf forms an encrypted linked list, without any parts of the linked list, the cloud cannot get the final correct results. Thus, the access pattern for the files is only $DB^*(\varphi)$. This is the significant difference with the state of the arts [25], [26], [28], [43].

The access pattern for node v $\{AP(\varphi, v)\}_{v\in tn(\varphi)}$ denotes, in fact, the DFA whether matches v or not. This is just an optimal point of the security-efficiency trade-off we seek. With this leakage, VBT-3 degrades to a level-2-revealing scheme.

We note that VBT-3 achieves only level-2-revealing (cannot achieve level-1-revealing). This is, however, the best one can do in a single-round SE scheme. For simplicity, we write $\mathcal{L}_2^{level2}$ to denote this leakage is level-2-revealing.

7.3 IND-CKA2 Security

Theorem 8.1 (*IND-CKA2 Security*). If F, P and V are pseudo-random functions, and H_1 and H_2 are different random oracles, then VBT-3 is IND-CKA2 ($\mathcal{L}_1, \mathcal{L}_2^{level2}$)-secure against an adaptive adversary.

Proof: We prove the scheme security at a high level. Let's consider such a stateful and efficient simulator $\mathcal{S}$, who can adaptively simulate the adversary's view including historical queries, the encrypted index, and files, by using only the leakage $\mathcal{L}$. We now prove that the adversary $\mathcal{A}$ cannot distinguish the real view from the simulated view with non-negligible probability. The simulation includes two parts $(\mathbf{I}^*, \mathbf{D}^*)$ and Q^*.

$\mathcal{S}$ creates a set of simulated files $\mathbf{D}^*$ and a simulated index $\mathbf{I}^* = (T_1^*, T_2^*)$ by using random values, since $\mathcal{S}$ has the $\mathcal{L}_1$ leakage.

$\mathcal{S}$ adaptively simulates the search trapdoor $Q = \{ft, it, bt\}$. Consider the query at a start node v. First, $\mathcal{S}$ builds a simulated trapdoor $Q^* = (ft^*, it^*, bt^*) = (ft^*, \{it_i^*\}_{i\in[0,u-2]}, \{bt_1^*, bt_2^*\})$ by using random values with the same size as Q since $\mathcal{S}$ has $\mathcal{L}_2$ leakage. Second, $\mathcal{S}$ randomly chooses a key-value pair (ft^{**}, x^{**}) from T_1^*, and programs the random oracle H_1 such that $T_1^* >> (ft^*, x^*)$ (i.e., $H_1(ft^*) = (ft^{**}; x^* \oplus x^{**})$). The following variables are similar to these ones. Third, $\mathcal{S}$ randomly chooses a set of key-value pairs $\{(b_1^{**}, y_1^{**}), (b_2^{**}, y_2^{**}), \cdots, (b_{u-1}^{**}, y_{u-1}^{**})\}$ from T_1^* and programs the random oracle H_1, such that $b_1^* \leftarrow x^* \oplus it^*[0]$, $T_1^* >> (b_1^*, y_1^*)$, $b_2^* \leftarrow y_1^* \oplus it^*[1]$, $T_1^* >> (b_2^*, y_2^*), \cdots, b_{u-1}^* \leftarrow y_{u-2}^* \oplus it^*[u-2]$, and $T_1^* >> (b_{u-1}^*, y_{u-1}^*)$. Fourth, according to the access pattern of $\mathcal{L}_2$, $\mathcal{S}$ knows that Q^* matches v or not. If Q^* doesn't match v, $\mathcal{S}$ ignores this value (don't simulate the query processing in the subtrees), otherwise $\mathcal{S}$ randomly chooses a pair $(c^{**}, (z_1^{**}; z_2^{**}))$ from T_2^* and programs the random oracle H_2 such that $c^* \leftarrow y_{u-1}^* \oplus bt_1^*$, and $T_2^* >> (c^*, (z_1^*; z_2^*))$. $\mathcal{S}$ randomly chooses two pairs $\{(ft_l^{**}, x_l^{**}), (ft_r^{**}, x_r^{**})\}$ from T_1^* for accessing the subtrees. $\mathcal{S}$ programs the random oracle H_1 such that $ft_l^* \leftarrow z_1^* \oplus bt_2^*$, $T_1^* >> (ft_l^*, x_l^*)$, $ft_r^* \leftarrow z_2^* \oplus bt_2^*$, and $T_1^* >> (ft_r^*, x_r^*)$. $\mathcal{S}$ recursively runs the above procedure until all subtrees are simulated. If the search process reaches a leaf, $\mathcal{S}$ uses the access pattern $DB^*(\varphi)$ for simulation. If a part of search tokens have appeared before, according to the search pattern, $\mathcal{S}$ chooses the corresponding pairs that have been used before for simulation. The simulated trapdoor Q^* will yield the same output as the original one Q.

According to the pseudo-random functions and the CPA-secure encryption algorithm, the simulated index and the original index, the simulated files and the original files, and the adaptively simulated search trapdoors and the original search trapdoors cannot be distinguished in polynomial time with non-negligible probability. This implies that except for the leakage $\mathcal{L}$, the adversary $\mathcal{A}$ learns nothing about the encrypted index, the encrypted files, and the trapdoors. Thus, VBT-3 is secure against an adaptive adversary. Furthermore, since $\mathcal{L}_2$ contains nothing of $\partial(\varphi)$, VBT-3 is level-2-revealing.

8 Experimental Evaluations

Level-2-revealing Boolean computation brings some drawbacks, such as index size blowing up especially in a disjunctive scheme. In this section, we perform some experiments to evaluate overall performance.

8.1 Experimental Methodology

DataSets: We choose the Enron email dataset [1] to evaluate VBT-1. This unstructured dataset consists of 517,401 email files in total, with sizes varying from 1 KB to 391 KB. To evaluate VBT-2 and VBT-3, we create an index for two data columns (age,gender) of a relational data table, where age∈[0,100] and gender∈$\{male, female\}$. We randomly generate 100,000 records and insert them into the table. To create an index for a relational data table, we use the prefix encoding approach and convert the table to a set of unstructured data files [43].

Implementation Details: We conduct our experiments on a desktop computer running Windows 10 Enterprise Edition with 64GB memory and an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor. We choose Blake2b as the pseudo-random functions. The schemes are fully implemented in C++. We also write several testing cases to output experimental data.

The height of the tree L is set to 32. H_1 outputs 40 bytes, with H_2 60 bytes. The maximum load factor of the hash tables is set to α=80%. In VBT-1, a keyword of a tree node consumes 100 bytes (40 bytes in T_1 and 60 bytes in T_2) only if this keyword is in this node. Thus, the index size of VBT-1 is five times that of [43]. In VBT-2, a keyword of a tree node occupies 220 bytes (160 bytes in T_1 and 60 bytes in T_2) even if this keyword is not in this tree node. VBT-3 consumes more disk space than other two, yet it is just our trade-off.

For simplicity, we write DB(w)=15 to denote a query with 15 results. We ignore all communication time and identifier-decrypting time. In all experiments, the indexes are loaded into memory first.

8.2 Index Construction Evaluations

Figure 2 and Figure 4 show the index size and time of VBT-1 are practically acceptable. Although the index size of VBT-1 is five times that of [43], it is still scalable even for an unstructured dataset, which means that the size of the keyword space m is dynamically changed with the number of files n growing.

Figure 3 and Figure 5 show that VBT-2 and VBT-3 are more suitable for a fixed-size dictionary (keyword space). Although they consume much disk space or memory, we can use them to protect those frequently-used keywords, since most of these keywords are much weaker than infrequently-used ones [16].

8.3 Query Processing Evaluations

Experimental results in Figures 6 and 7 demonstrate that these three solutions are highly-efficient, and their query processing time in the millisecond scale. The data also shows that the s-term problem that exists in OXT [26] and BIEX [28] has been well addressed. Figures 6 and 7 give the conjunctive query time and disjunctive query time, respectively, with the number of files n growing.

Figure 8 demonstrates that the Boolean query of VBT-3 is highly-efficient and it is scalable in the final result set size, where "age$\in$[20,30] $\wedge$ g.=male" denotes that all users whose age are between 20 and 30, and whose gender are male. This figure implies that VBT-3 supports multi-dimensional range/keyword Boolean queries. It will take 96.3 ms to search over 100,000 records by using a Boolean query consisting of range-query terms and string terms with 5000 results matched.

8.4 Compared with IBTree and VBTree

Experimental results in Figures 9 demonstrate that VBT-1 is more efficient than IBTree [25] and is almost as efficient as VBTree [43]. We conduct these experiments over 1 million files. All the trees can be optimized with traversal width or height by using their proposed approaches in [25] and [43]. Thus, we generate a set of queries that are all in the worst-case distribution (random distribution). IBTree is slower than the other two due to two facts. First, the number of non-contiguous memory accesses (locality [42]) of IBTree is k in a tree node, where k is the number of hash functions in a Bloom filter. Second, the number of pseudo-random computations of IBTree is k in a tree node, whereas the other two have only one or two locality or computations in a tree node.

To support dynamic updates with forward and backward privacy [19] [27], we use the version control repository (VCR) proposed in [43]. All forward tokens are marked with different versions. This will be left to our future work.

9 Conclusions

In this paper, we propose ideal/real encrypted Boolean function concepts to mark all single-round Boolean-query SE schemes with different security levels. We present a novel approach to encrypt and run deterministic finite automatons (DFAs) on untrusted clouds. Based on this fundamental component, we give three SE constructions for conjunctive/disjunctive/Boolean queries, respectively. Their advantage is that they achieve sub-linear search complexity and enhanced security that we call level-2-revealing. The experimental results show that these solutions support efficient Boolean queries and can be used for building an index for private encrypted databases. Our future work includes 1) designing rich database queries based on the encrypted DFA computation; 2) optimizing the index size of VBT-3; 3) developing new level-2-revealing DFA-query schemes.

References

[1] Enron email dataset. http://www.cs.cmu.edu/~enron/, 2015.
[2] E. J. Goh. "Secure Indexes," IACR Cryptology ePrint Archive, 2003.
[3] D. X. Song, D. Wagner, and A. Perrig. "Practical techniques for searches on encrypted data," in Security and Privacy (S&P). IEEE, 2000.
[4] R. Curtmola, J. Garay, S. Kamara, et al., "Searchable symmetric encryption: improved definitions and efficient constructions," in the 13th ACM conference on Computer and Communications Security (CCS), pp. 79-88. ACM, 2006.
[5] M. Chase and S. Kamara, "Structured Encryption and Controlled Disclosure," in the Theory and Application of Cryptology and Information Security (ASIACRYPT), pp. 577-594. Springer, 2010.
[6] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS), pp. 965-976. ACM, October, 2012.
[7] R. A. Popa, C. Redfield, et al., "CryptDB: protecting confidentiality with encrypted query processing," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), pp. 85-100. ACM, October, 2011.
[8] S. Bajaj, and R. Sion, "TrustedDB: a trusted hardware based database with privacy and data confidentiality," in the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD), pp. 205-216. ACM, 2011.
[9] S. Kamara, and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in International Conference on Financial Cryptography and Data Security, pp. 258-274. Springer, Berlin, Heidelberg, April, 2013.
[10] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in International conference on the theory and applications of cryptographic techniques, pp. 506-522. Springer, Berlin, Heidelberg, May, 2004.
[11] R. Li, A. X. Liu, A. L Wang, et al., "Fast range query processing with strong privacy protection for cloud computing," in International Conference on Very Large Data Bases (VLDB), vol. 7, no. 14, pp. 1953-1964, 2014.
[12] B. Bruhadeshwar, A. X. Liu, B. Jayaraman, A. L. Wang, and R. Li, "Privacy preserving string matching for cloud computing," in 2015 IEEE 35th International Conference on Distributed Computing Systems, pp. 609-618. IEEE, 2015.
[13] W. K. Wong, D. W. L. Cheung, B. Kao, and N. Mamoulis. "Secure kNN computation on encrypted databases," in Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 139-152, ACM, 2009.

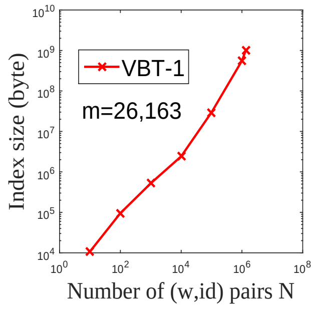


Fig. 2: Index size (VBT-1)

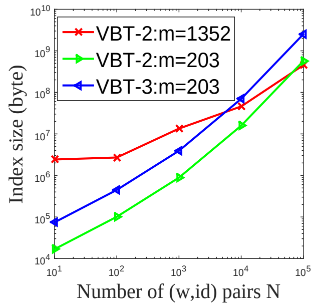


Fig. 3: Index size (VBT-2,3)

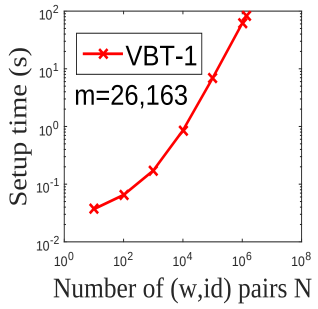


Fig. 4: Index time (VBT-1)

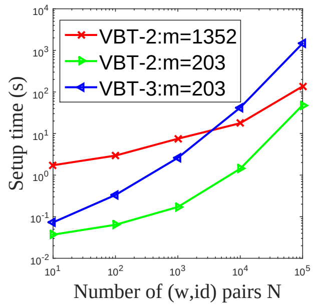


Fig. 5: Index time (VBT-2,3)

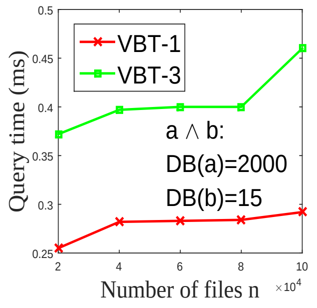


Fig. 6: Conj. queries

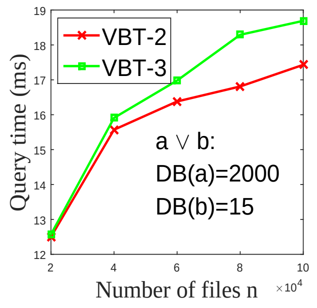


Fig. 7: Disj. queries

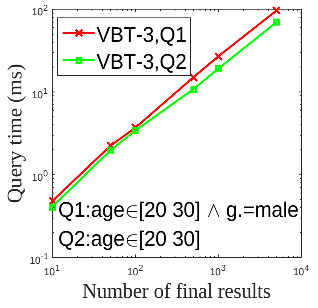


Fig. 8: Boolean queries including ranges

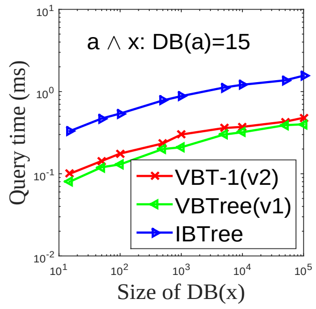


Fig. 9: Compared with IBTree and VBTree

[14] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 24-43, Springer, Berlin, Heidelberg, May, 2010.

[15] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti, "Modular order-preserving encryption, revisited," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 763-777. ACM, 2015.

[16] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 644-655. ACM, 2015.

[17] Z. Chang, D. Xie, and F. Li, "Oblivious ram: a dissection and experimental evaluation," in Proceedings of the VLDB Endowment, vol. 9, no. 12, pp. 1113-1124. 2016.

[18] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: data structures and implementation," In NDSS, Vol. 14, pp. 23-26, 2014.

[19] R. Bost, "$\Sigma o\varphi o\varsigma$: Forward Secure Searchable Encryption," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1143-1154. ACM, 2016.

[20] M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in 2014 IEEE Symposium on Security and Privacy (S&P), pp. 639-654. IEEE, 2014.

[21] G. Kellaris, G. Kollios, K. Nissim, and A. O'neill, "Generic attacks on secure outsourced databases," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1329-1340. ACM, 2016.

[22] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in Annual International Cryptology Conference (Crypto), pp. 563-592. Springer, Berlin, Heidelberg, 2016.

[23] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in Proceedings of the 15th ACM conference on Computer and communications security (CCS), pp. 257-266. ACM, October, 2008.

[24] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in NDSS, vol. 71, pp. 72-75, February, 2014.

[25] R. Li, and A. X. Liu, "Adaptively secure conjunctive query processing over encrypted data for cloud computing," in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 697-708. IEEE, April, 2017.

[26] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. C. Ro?u, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in Annual Cryptology Conference (Crypto), pp. 353-373. Springer, Berlin, Heidelberg, August, 2013.

[27] Y. Zhang, J. Katz, and C. Papamanthou. "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in 25th USENIX Security Symposium (USENIX), pp. 707-720, 2016.

[28] S. Kamara, and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 94-124. Springer, Cham. April, 2017.

[29] V. Pappas, F. Krell, B. Vo, et al., "Blind seer: A scalable private dbms," in 2014 IEEE Symposium on Security and Privacy (S&P), pp. 359-374. IEEE, May, 2014.

[30] A. C. Yao, "Protocols for secure computations," In FOCS, vol. 82, pp. 160-164, November, 1982.

[31] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," IEEE transactions on parallel and distributed systems (TPDS), vol. 27, no. 2, pp. 340-352. IEEE, 2016.

[32] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," IEEE Transactions on parallel and distributed systems (TPDS), vol. 25, no. 1, pp. 222-233. IEEE, 2014.

[33] B. Yao, F. Li, and X. Xiao, "Secure nearest neighbor revisited," in 2013 IEEE 29th international conference on data engineering (ICDE), pp. 733-744. IEEE, April, 2013.

[34] C. Gu, and J. Gu, "Known-plaintext attack on secure kNN computation on encrypted databases," Security and Communication Networks, vol. 7, no. 12, pp. 2432-2441. 2014.

[35] D. Boneh, and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in Theory of Cryptography Conference (Crypto), pp. 535-554. Springer, Berlin, Heidelberg, February, 2007.

[36] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in Proceedings of the 2012 ACM conference on Computer and communications security (CCS), pp. 784-796. ACM, October, 2012.

[37] X. Lei, A. X. Liu, and R. Li, "Secure knn queries over encrypted data: Dimensionality is not always a curse," in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 231-234. IEEE, April, 2017.

[38] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in Annual International Cryptology Conference (Crypto), pp. 339-370. Springer, Cham, August, 2018.

[39] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in Proceedings of the 22nd ACM SIGSAC conference on computer and communications security (CCS), pp. 668-679. ACM, October, 2015.

[40] K. S. Kim, M. Kim, D. Lee, et al., "Forward secure dynamic searchable symmetric encryption with efficient updates," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1449-1463. ACM, October, 2017.

[41] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized I/O efficiency," Transactions on Dependable and Secure Computing (TDSC). IEEE, 2018.

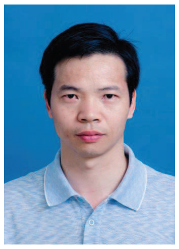
[42] D. Cash, and S. Tessaro, "The locality of searchable symmetric encryption," In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 351-368. Springer, Berlin, Heidelberg, May, 2014.

[43] Z. Wu, and K. Li, "VBTree: forward secure conjunctive queries over encrypted data for cloud computing," The VLDB Journal—The International Journal on Very Large Data Bases, vol. 28, no. 1, pp. 25-46, 2019.

[44] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, " The tao of inference in privacy-protected databases," in Proceedings of the VLDB Endowment, vol. 11, no. 11, pp. 1715-1728, 2018.

[45] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS), pp. 299-310. ACM, November, 2013.

Zhiqiang Wu received the B.S. degree in computer application technology from the Central South University, China, in 2003, and received the Ph.D. degree in computer science from the Hunan University, China, in 2019. He currently works with the Changsha University of Science and Technology. His research interests include network security, data encryption, embedded systems, software architecture, high-performance computing, and big data computing. He has authored several papers in international journals, such as the VLDB Journal.



Kenli Li received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a Visiting Scholar with the University of Illinois at Urbana-Champaign, from 2004 to 2005. He is currently a Full Professor of computer science and technology with Hunan University and the Deputy Director of the National Supercomputing Center, Changsha. He has authored over 150 papers in international conferences and journals, such as the IEEE-TC, the IEEE-TPDS, and the IEEE-TSP. His major research includes parallel computing, cloud computing, and big data computing. He is an Outstanding Member of CCF. He is currently serving on the editorial boards of the IEEE TRANSACTIONS ON COMPUTERS and the International Journal of Pattern Recognition and Artificial Intelligence.

Keqin Li Dr. is a SUNY Distinguished Professor of computer science with the State University of New York. He is also a Distinguished Professor at Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyberphysical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has published over 630 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He currently serves or has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow.

Jin Wang received the B.S. and M.S. degrees from the Nanjing University of Posts and Telecommunications, China, in 2002 and 2005, respectively, and the Ph.D. degree from Kyung Hee University, South Korea, in 2010. He is currently a Professor with the Changsha University of Science and Technology. He has published more than 300 international journal and conference papers. His research interests mainly include wireless sensor networks, network performance analysis, and optimization. He is a Senior Member of the IEEE and a member of ACM.