2015 IEEE 29th International Parallel and Distributed Processing Symposium

Fast Proof Generation for Verifying Cloud Search

Jingyu Zhou*, Jiannong Cao†, Bin Yao*, and Minyi Guo*
*Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China
†Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong

Abstract—As cloud computing has become prominent, the need for searching cloud data has grown increasingly urgent. However, cloud search may be incorrect due to errors of cloud providers and attacks from other malicious tenants. Previous work on verifiable computing returns results with probabilistically checkable proofs, which targets at different applications other than search and requires a large computation overhead.

We propose a hybrid approach for generating proofs of cloud search results. Specifically, we model search indices as sets and search operations as set intersections, and build proofs based on RSA accumulators and aggregated membership and nonmembership witnesses. Because generating witnesses for large sets is computationally expensive, we employ interval-based witnesses for fast proof generation. To reduce proof size, our hybrid method uses Bloom filters when set difference is large. Evaluation on real datasets shows that our hybrid approach generates proofs in an average of 0.197s, up to 83.2% faster than previous work with a smaller proof size. Experiments also show our approach allows incremental updates with constant cost.

Keywords—Verifiable computing, RSA accumulator, Bloom filter.

I. Introduction

The paradigm shift of cloud computing has changed the ways that people use IT systems. Increasingly, individuals and organizations have stored their data in the cloud for higher availability, larger storage, and lower cost. It is important for cloud services to provide search for users' data stored in the cloud. However, cloud search may return incorrect results due to software or hardware errors [1], misbehaving cloud operators [2], or attacks from other tenants. The cloud may also be economically motivated to only compute partial results to save its computation resource and network bandwidth.

With the increasing usage of multiple devices, e.g., smart phones and pads, users' data may only be accessible from cloud storage when switching devices. Additionally, because of limited storage space on mobile devices, users often delete local files after outsourcing data to save local storage space. As a result, the device may keep no knowledge of the outsourced data when a user asks the cloud to perform searches. This is a challenging problem because the cloud must prove search results are produced from valid input data. Additionally, the cloud must quickly generate proofs of correct results for online search. Because the cloud may be malicious to return false results, the client should be able to efficiently verify search results are correctly computed. Finally, the client may dynamically update his data and the cloud needs to support incremental changes.

This paper focuses on verifying cloud search and we design a verifiable index for fast proof generation. As shown in Figure 1, a data owner outsources his data as well as verifiable index files. After the cloud acknowledges the receipt of files, the owner may delete all local data stored in a space-constrained mobile device. The cloud is responsible for storing data and performing keyword-based search for the owner. The owner can quickly verify the correctness of search results with proofs returned by the cloud. If there are any problems during the search, both the cloud and the client may prove them to a third party.

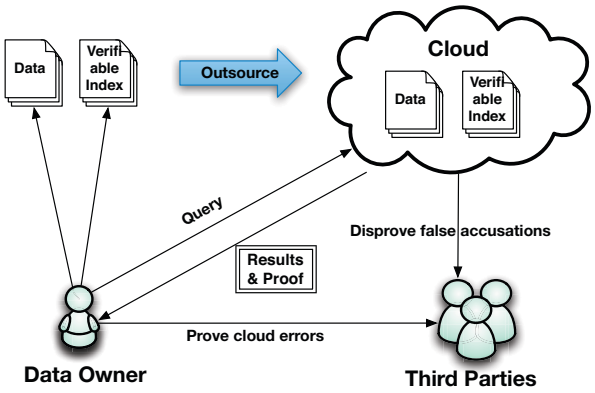

Fig. 1: The two-party search model. A data owner first outsources his data together with a verifiable index to a cloud. Then the owner may ask the cloud to search his data. The cloud returns results with proofs. All communication messages between the owner and the cloud are signed, so that both of them can convince third parties of the other party's errors.

To verify the outsourced computation, previous work has proposed a number of approaches, starting from the theoretic work on interactive proofs [3] and probabilistically checkable proofs (PCP) [4]. Recent work on verifiable computing [5]–[9] has improved these protocols considerably, but generating proofs is at least 10^5 times slower than the outsourced computation. Trusted computing [10]–[14] verifies the integrity of executables before their execution. However, attackers may overwrite the executables in memory after they are checked, but before execution. Consequently, the results could still be incorrect. Finally, a number of previous studies have proposed verifying query results from outsourced databases or data structures [15]–[20]. Except [16], these approaches do not support multi-keyword queries. [16] provided probabilistic assurance for an executed *batch* of queries, not proof for each individual query. In addition, the client needs to possess

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.11

segments of the queried data, which is often not the case for the outsourced cloud search.

Different from generic PCP-based approaches for addressing these challenges, we model search indices as sets and search operations as set intersections, which can be verified more cheaply with collision-resistant RSA accumulators, aggregated (non)membership witnesses, and Bloom filters. Based on the strong RSA assumption [21], a set of values can be condensed into one short accumulator and each value has a short witness. As a result, we can use the signed witness values for verifying the cloud possesses the valid input data. To verify the correct set intersection is performed, we extend the previous (non)membership witness for a single value to multiple values, and the security of this extension can be proven under the strong RSA assumption.

Computing aggregated witnesses as proofs is computationally expensive and results in long delays for returning proofs to clients. To address this problem, we have performed a number of optimizations. First, we use an interval-based witness to reduce the time for computing witnesses of large sets and unknown search keywords. Second, to further reduce the proof size, we introduce a hybrid scheme that combines aggregated witnesses together with Bloom filters. Finally, we use an offline pre-computing strategy and parallel execution to further reduce online proof generation time.

We have implemented a prototype system and evaluated its performance on two real world datasets. Our evaluation demonstrates that our hybrid scheme is up to 83.2% faster than previous work based on aggregated membership witness and Bloom filters [22], and the proof size of our scheme is smaller. Experimental results also show that our approach support incremental updates of users' data with a constant cost. Our contributions include the hybrid design of set intersection proofs and practical optimizations for reducing proof generation time.

The rest of the paper is organized as follows. Section II describes how to verify set intersection results. Section III presents the design of verified cloud search. Section IV discusses the implementation of our prototype system. Section V evaluates the performance of our prototype. Section VI summarizes related work. Finally, Section VII concludes with future work.

II. RSA Accumulator and Set Intersection

In this section, we first describe RSA accumulators and associated set membership and nonmembership witnesses, as well as aggregated witnesses for multiple values. Based on the aggregated witnesses, we construct the proof of set intersections. Finally, dynamic changes of set members are discussed.

A. RSA Accumulator and Membership Witness

Benaloh and de Mare [23] first introduced the idea of *one-way accumulators* that condense a set of values into one short accumulator. Our construction of the verifiable index is based on a particular type of one-way accumulators, i.e., RSA accumulator [18]. The RSA accumulator supports both set membership and non-membership witnesses [18]–[20]. Under the strong RSA assumption [21], it is computationally infeasible to forge these witnesses. Let k be a security parameter. Let $l = \lfloor k/2 \rfloor - 2$. Let $\mathcal{X}_k$ be the set of all primes in $\mathbb{Z}_{2^l}$. Let $n = pq$ be a random modulus of length k, where p and q are two safe primes of equal length.

Suppose the input set $X = \{x_1, x_2, \ldots, x_{|X|}\}$ is a subset of $\mathcal{X}_k$ and g is a random value in QR_n, where QR_n denotes the set of quadratic residues modulus n. Let $u = \Pi_{i=1}^{|X|} x_i$. Then the *set accumulator* is:

$$c = g^u \bmod n. \tag{1}$$

This accumulator has the property that any computationally bounded adversary $\mathcal{A}$ that does not know $\phi(n)$ (the Euler totient function), cannot use polynomial time algorithms to find another set of elements $Y \neq X$ such that the accumulator value for Y is also c. Otherwise, the adversary $\mathcal{A}$ breaks the strong RSA assumption [21].

The *membership witness* for a value x in the set X is:

$$c_x = g^{u/x} \bmod n. \tag{2}$$

The witness can be verified by checking $(c_x)^x = c \bmod n$.

The *nonmembership witness* for a value $x \in \mathcal{X}_k \setminus X$ is a value pair (a, d) [20], subject to:

$$c^a = d^x g (\mathrm{mod} n). \tag{3}$$

B. Aggregated Witness for Set Members

The above (non)membership witnesses can be extended for multiple (non)members. As a result, a single witness can be used to validate the (non)membership for multiple values.

1) Aggregated Membership Witness: For a subset $X' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_{|X'|}}\}$ of X and $v = \Pi_{k=1}^{|X'|} x_{i_k}$, the witness $c_{X'}$ of $X' \subseteq X$ is [22]:

$$c_{X'} = g^{u/v} \bmod n. \tag{4}$$

The witness can be verified by checking $(c_{X'})^v = c \bmod n$.

2) Aggregated Nonmembership Witness: We extend the non-membership witness to a set $Y = \{y_1, y_2, \ldots, y_{|Y|}\}$, such that Y is a subset of $\mathcal{X}_k$ and $Y \cap X = \varnothing$. Let $v = \Pi_{i=1}^{|Y|} y_i$. Because all elements in Y and X are prime numbers, $gcd(u, v) = 1$. We can find a value pair (a, d) such that $au + bv = 1$ and $d = g^{-b} \bmod n$. The nonmembership witness for Y is (a, d). To validate the witness, one can verify $c^a = d^v g \ (\mathrm{mod} n)$.

The above equation holds because $c^a = g^{ua} = g^{1-bv} = g^{-bv} g = d^v g \ (\mathrm{mod} n)$.

Theorem 1: Under the strong RSA assumption, the above construction of aggregated (non)membership witness is a secure universal accumulator.

The security proof this theorem is similar to [20], [21] and is omitted due to space constraint. Here "universal" means that each possible value in the input domain has either a membership or a nonmembership witness.

3) Usage: To use the above accumulators and aggregated witnesses, we need a general solution to map arbitrary elements to prime numbers. Fortunately, the notion of *prime representatives* [18], [24] gives such a solution, which can map general k-bit elements to $3k$-bit prime numbers (i.e., prime representatives). In our search application, the data owner makes the following information public for the cloud or a third-party user:

- A random modulus n and a random number $g \in QR_n$.
- The generator for prime representatives.

The data owner keeps p and q for efficient computation of modulus exponentiation. This is because the owner knows p and q, which are relatively prime with g. Due to Euler's Theorem, which states that $g^{\phi(n)} \bmod n = 1$, we have $g^x \bmod n = g^{x \bmod \phi(n)} \bmod n$, where $\phi(n) = (p-1)(q-1)$.

The data owner also possesses the accumulated value c for verification. If a third-party user needs to verify witnesses generated by the cloud, the user has to contact the owner to obtain the accumulator c.

The above scheme assumes the owner stores accumulators. If the owner is space constrained, he could adopt a different scheme, where the owner digitally signs c and lets cloud servers keep the signed c. Then the owner may remove his local copy of the data. The cloud will return signed c along with the results so that the owner or the third-party user may first verify c and then validate the correctness of the results with c.

C. Proof for Set Intersection Result

Based on the aggregated witnesses, we now discuss the proof of set intersection results. When a client requests the cloud to perform a set intersection operation of two sets, X_1 and X_2. The cloud returns the result $X = X_1 \cap X_2$, together with a *correctness proof*, which is a membership witness that X is a subset of both X_1 and X_2. The witness cannot be forged and the client uses Equation 4 to verify that X is the correct answer.

Using the correctness proof alone is not enough. This is because the server can intentionally hide some information and only return partial results. Thus, we need another *integrity proof* to specify that the returned results X contains all elements in the intersection of X_1 and X_2. In other words, we need attestations of $X_1 \setminus X$ does not contain any elements of X_2. Such an integrity proof contains:

- the set of $X_1 \setminus X$,
- an aggregated membership witness of $X_1 \setminus X$ belonging to X_1,
- and an aggregated witness of $X_1 \setminus X$'s nonmembership for X_2.

Both the aggregated membership witness and nonmembership witness consume constant space. The set $X_1 \setminus X$ contains *check elements* for the intersection results.

1) Handling Multiple Sets: For intersections of multiple sets $X = X_1 \cap X_2 \cap \ldots \cap X_s$, the accumulator-based integrity proof can be extended to contain accumulated membership witness of $X_1 \setminus X$ belonging to X_1, and multiple accumulated nonmembership witnesses that elements in $X_1 \setminus X$ do not occur in one of $X_2, X_3, \ldots, X_s$.

D. Updating Accumulator for Dynamic Set

Because one can dynamically add or delete elements from a set, we need efficient algorithms for updating accumulators.

Let a set of values $\hat{X} = \{\hat{x_1}, \hat{x_2}, \ldots, \hat{x_{s'}}\}$ to be added to the original set X with accumulator c. The new accumulator $\hat{c}$ can be computed as:

$$\hat{c} = c^{\Pi_{i=1}^{s'} \hat{x_i}} \bmod n. \tag{5}$$

On the other hand, if a subset $X' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_{s'}}\}$ of X is deleted, the new accumulator $\hat{c}$ is:

$$\hat{c} = c^{(\Pi_{j=1}^{s'} x_{i_j}^{-1}) \bmod \phi(n)} \bmod n. \tag{6}$$

III. Design

In this section, we first present an overview of our approach. Then we discuss the basic version of our verifiable index and cloud search. Because the basic version has high time and space costs, we perform several optimizations to reduce these costs. Finally, we discuss result verification.

A. Overview

We consider a *two-party model* as shown in Figure 1, consisting of a data owner and a cloud instance. The data owner first builds inverted index for his data and computes accumulators for the index, and then outsources all his data to the cloud. After the cloud acknowledges the receipt of files, the owner may delete his local copy of data, index, and accumulators effectively keeping *no knowledge* about the outsourced data. The cloud convinces the owner that it knows the inverted index via signed index data and accumulator values.

Our verifiable index allows the data owner to convince third parties of the errors of the cloud. Conversely, the cloud may disprove false accusations made by the data owner. To achieve these security properties, we assume both the data owner and the cloud have a secure public and private key pair and there is a secure way of verifying public keys. Both the owner and the cloud must sign their messages so that the other party may present the signed messages to convince a third party.

B. Basic Verifiable Index

Before outsourcing data to the cloud, the data owner first builds a *verifiable index* for his data. We model the verifiable index as a map from a term t_i to a set and two RSA accumulators:

- **Set (*i.e.*, inverted index).** The set in a verifiable index corresponds to the traditional inverted index, where each element in the set represents feature values of the term in a document. Specifically, each set contains a number of $(docID, w)$ tuples, where $docID$ denotes a document ID and w denotes the weight of t_i in the document. The simplest weight can be a single TF or TF-IDF [25] value. For better search results, w can also be a vector of feature values [26]. In our settings, the data owner may freely choose features to build into w.

- **RSA accumulators.** For each term, the owner needs to build two accumulators, one for all tuples in the index and the other for all document IDs in tuples. The reason for the latter accumulator is that integrity proofs do not need to include information on term weights, and constructing proofs on document IDs can be cheaper. The owner can compute the accumulator more efficiently than the cloud (Section II-B3).

The data owner signs each component of the verifiable index and uploads it to the cloud. After that, the owner may discard local copy of the verifiable index. If needed, the owner requests the cloud to return parts of the verifiable index (e.g., the inverted index and accumulators for a keyword), and verifies the data with signatures. Because a set can be easily updated with new elements and RSA accumulator supports dynamic changes (Section II-D), the data owner can incrementally update cloud data.

C. Cloud Search

After an owner issues a signed search query, the cloud server performs keyword search operations in the following steps:

- Look up verifiable index and find each keyword's index data;
- Perform intersection of index data;
- Compute proof on the intersection results;
- Finally sign and return results and proofs to the owner.

The intersection of index data finds documents that contain all query keywords. For instance, given two query keywords q_i and q_j, the cloud first retrieves their inverted indices, I_i and I_j. Let S_i and S_j denote the set of documents appear in I_i and I_j, respectively. The cloud computes the set of documents $S = S_i \cap S_j$ that contain both terms.

Let R_i (or R_j) denote the subset of tuples in I_i (or I_j) containing documents of S. R_i and R_j together represent the indexing results. The proof of indexing results includes a correctness proof and an integrity proof. The correctness proof of indexing results consists of membership witnesses of tuples of all query keywords. For the above two keywords example, the correctness proof includes: 1) R_i is a subset of I_i; and 2) R_j is a subset of I_j. Both are computed with the subset witness given in Equation 4. In general, if the number of search keywords is Q, then the correctness proof takes $O(Q)$ space, i.e., one for each keyword.

The integrity proof proves that no other tuples should be included in the results. Note the integrity proof does not care about weight w in tuples of inverted indices. Assuming $|I_i| \leq |I_j|$ in the above example, the accumulator-based integrity proof contains attestations that for all tuples in $I_i \setminus R_i$, the corresponding document does not appear in I_j. Such a proof contains three parts:

- the set $S_i \setminus S$;
- the witness that $S_i \setminus S$ is a subset of S_i;
- the witness $S_i \setminus S$ does not contain elements in S_j.

In general, if the number of search keywords is Q, the third part contains up to $Q-1$ nonmembership witnesses. I.e., we need to prove each element in $S_i \setminus S$ does not appear in at least one of $Q-1$ other sets. Thus for multiple query keywords, the size of integrity proof takes $O(Q) + O(|S_i|)$ space. To reduce the size of integrity proof, we always choose the smallest inverted index to be S_i.

Given a set of inverted indices (one for each query keyword) and search results, we first sort inverted indices in the increasing number of elements and choose the smallest one to build the integrity proof, because the proof size for any other index is larger. When building nonmembership witnesses, we only need to prove that a value (i.e., a document ID in this case) does not appear in one of the other $Q-1$ sets. Again, we choose the smallest set that satisfies such a condition, because computing nonmembership witnesses for a larger set is computationally more expensive.

D. Optimizations

The basic version of verifiable index has a high time cost. Figure 2 illustrates the results of a micro benchmark of witness generation times, where the average of 200 different values is reported. We can observe that the time for generating witnesses is linear to the size of sets. For 20,000-element sets, both types of witnesses require more than one second. Because a search operation may involve multiple such large sets, the time for generating proofs can easily become several seconds, which is undesirable for an online service.

Another problem with the basic version is that the size of integrity proof is unbounded and can consume a large amount of space due to many check elements in $S_i \setminus S$.

In the following, we discuss several optimizations for reducing these time and space costs.

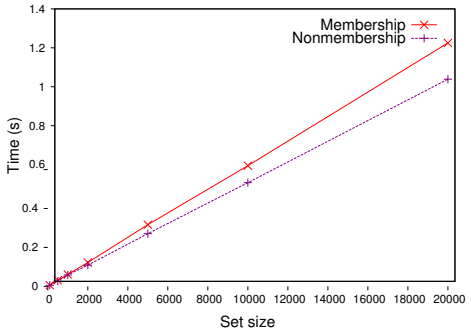

Fig. 2: Comparison of the witness generation times on a 2.9 GHz Intel Core i7 CPU.

1) Interval-based Witnesses for Large Sets: The idea of interval-based witnesses is to split a large set into a number of smaller sets and compute (non)membership witnesses for these smaller ones. As illustrated in Figure 3, we first sort elements in a large set X and divides the set into smaller sets (also called *value intervals*) X_i of fixed size. Then we compute the accumulator value b_i for each smaller set X_i. c is the accumulator value for the set $B = \{b_1, b_2, \ldots\}$ of middle layer values. For each b_i, we compute its membership witness c_{b_i} of $b_i \in B$. Note c can be considered as the accumulator for the whole set X.

To compute a membership witness for a value v, we simply find the interval k containing v and calculate the membership

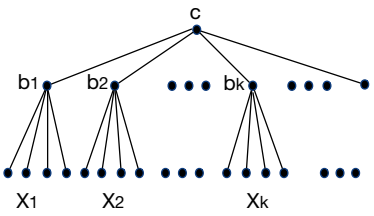

Fig. 3: Interval-based witnesses are based on a tree structure. Elements in a large set X are divided into a number of smaller sets (or value intervals), e.g., X_k. b_k is the accumulator of X_k. c is the accumulator of the middle layer values.

witness $\hat{c}$ within the small set X_k. Finally, tuple $(\hat{c}, b_k, c_{b_k}, c)$ represents the proof of v in the X, and anyone can use $\hat{c}$ and b_k to verify v is in X_k and use c and c_{b_k} to verify b_k is part of c. Because b_k, c_{b_k}, and c are computed in advance, we only need to calculate $\hat{c}$ during the search, which can be quickly obtained with a small size for X_k. Nonmembership proof for a value can similarly computed and verified, where $\hat{c}$ represents a value pair.

Finally, we can easily extend the above single value witness to multiple values. Given a set of values, we first find the intervals containing these values. Then for each interval X_k, we compute the aggregated witness $\hat{c}$ for values in this interval. The proof for all values is a collection of tuples $(\hat{c}, b_k, c_{b_k}, c)$ from each interval k.

2) Bloom Filters for Proof Size Reduction: To address the problem of large integrity proof size, we employ counting Bloom filters [27]. Given two sets X_1 and X_2. The idea is to construct two Bloom filters $B(X_1)$ and $B(X_2)$, and the integrity proof contains these two Bloom filters together with significantly fewer check elements.

Assuming $X = X_1 \cap X_2$ and hash functions can evenly distribute elements in a range of $\{1, \ldots, m\}$, elements in $X_1 \setminus X$ will be hashed to different values from those in $X_2 \setminus X$ with high probability. As a result, the element-by-element minimum of $B(X_1)$ and $B(X_2)$, denoted as $\hat{B}$, is a close approximation of the Bloom filter $B(X)$.

For each element $j \in \{1, \ldots, m\}$, if we denote $B(X)_j$ as the jth element of the Bloom filter for set X, then the following condition holds:

$$B(X)_j \leq \hat{B}_j, \tag{7}$$

because each element in X appears in both X_1 and X_2. $\hat{B}_j$ denotes the jth element of the Bloom filter $\hat{B}$.

In the above formula, if $B(X)_j = \hat{B}_j$, then there must be $B(X)_j$ elements in X that hashed to j. If $B(X)_j < \hat{B}_j$, then there must be a pair of sets (C_1, C_2) that are subsets of $X_1 \setminus X$ and $X_2 \setminus X$, respectively, satisfying the following conditions:

$$B(X)_j + B(C_1)_j = B(X_1)_j, \tag{8}$$

$$B(X)_j + B(C_2)_j = B(X_2)_j. \tag{9}$$

That is, C_1, C_2 contain the check elements that hash to j, too. Due to hash functions, the number of check elements can be significantly smaller than $X_1 \setminus X$ or $X_2 \setminus X$.

In summary, the Bloom filter based integrity proof contains:

- the Bloom filters $B(X_1)$ and $B(X_2)$ and their signatures,
- two check element sets C_1, C_2 and their accumulated membership witnesses w_1, w_2.

In the above integrity proof, signatures and witnesses consume constant space. The size of Bloom filters can be reduced with a compression algorithm [28]. Let the counting Bloom filter have k hash functions with m counters. Let $l_1 = \frac{k|X_1|}{m}, l_2 = \frac{k|X_2|}{m}$ be the loads of $B(X_1)$ and $B(X_2)$ and let l be the maximum load allowed. Both the size z of the compressed counting Bloom filters and the expected size of C_1, C_2 are bounded [22]:

$$z = mH(l), \tag{10}$$

$$E[|C_1|] \leq m(1 - e^{-l_2})l_1 \leq ml_1l_2, \tag{11}$$

$$E[|C_2|] \leq m(1 - e^{-l_1})l_2 \leq ml_1l_2, \tag{12}$$

where $H(l)$ is the entropy of a Poisson distribution with mean l. The expected number of check elements, $|C_1| = |C_2| = ml_1l_2 = k^2|X_1||X_2|/m$, is minimized when k is one.

The client verifies the integrity proof by checking:

- the correctness of Bloom filters with their signatures;
- set X, C_1, and C_2 are disjoint;
- w_1 and w_2 proves $C_1 \subset X_1$ and $C_2 \subset X_2$, respectively;
- for each $j \in \{1, \ldots, m\}$ such that $B(X)_j < \hat{B}_j$, there are enough check elements satisfying Equation 8 and Equation 9.

The idea of using Bloom filters for verifying set intersections was first proposed in [22]. We note that this approach can generate larger proof than the accumulator-based one if the set $X_1 \setminus X$ is small. This paper leverages both approaches for different cases and achieves smaller proof than each of these two methods (Section V-B).

3) Pre-computing Prime Representatives: As discussed in Section II-B3, we need to compute prime representatives [18], [24] for inverted index data. This computation can be performed offline and we employ a *pre-computing* strategy that computes and stores prime representatives for all inverted indices. Our experiments in Section V-E show that this strategy alone can save more than 92.6% of time for generating proofs.

4) Unknown Search Keywords: In the above construction, we assume the inverted index contains all search keywords. If a search keyword does not appear in the inverted index, an empty result is returned together with a nonmembership proof, specifying the word is not in the dictionary of all known keywords. A possible solution is to use a Bloom filter of all known keywords as the proof. However, we find that even the size of compressed Bloom filter [28] is in the order of hundreds of kilobytes for a modest dictionary containing 50,000 words, a high overhead for search results. As we discussed previously, the simple accumulator-based nonmembership witness requires time in the order of seconds for a large dictionary.

To quickly generate small proofs for unknown search keywords, our solution is based on the membership witness (Equation 2), where the witness proof is of constant size and is computed offline. Let $w_1, w_2, \ldots, w_{|W|}$ be the sorted sequence of words in the dictionary. Instead of computing

prime representatives of w_i, we compute prime representatives of the $|W|+1$ intervals (w_i, w_{i+1}) for $i = 0, \ldots, |W|$, where w_0 and $w_{|W|+1}$ denote $-\infty$ and $+\infty$, respectively. The proof of nonmembership for a search keyword $w \in (w_i, w_{i+1})$ can equivalently be represented as the witness of membership for interval (w_i, w_{i+1}). Because prime representatives and witnesses for all intervals can be computed beforehand, the online search only needs to find the interval and to retrieve the corresponding constant-size witness, consuming $O(\log |W|)$ time. Our evaluation in Section V-C shows that such a proof can be found within one millisecond.

5) Single-keyword Query: If a user query only contains a single keyword, the query result contains the whole inverted index for the keyword, and the proof for the result can be the owner's signature of the index data. Because signatures are faster to be generated and verified than accumulators, the signatures of index data are used as fallback proofs for single-keyword queries. Returning all inverted indices for multiple-keyword queries is undesirable, because many indices are much larger than the search results.

E. Verification of Search Results

The data owner verifies the returned results with proofs. Here we focus on the verification of multi-keyword search without unknown keywords. The verification must pass all of the following checks:

- Results and proofs are signed by the cloud;
- Tuples for each keyword must contain the same set of document IDs (results);
- Tuples for each keyword represent a subset of the keyword's inverted index (correctness proof);
- The integrity proof is valid (results and integrity proof):
  - For accumulator-based integrity proof, the set of document IDs in integrity proof is the complement of indexing results, and nonmembership witnesses of integrity proof are correct and cover all document IDs in the integrity proof;
  - For Bloom filter based integrity proof, the set of document IDs in integrity proof is disjoint with those of indexing results, and there are enough check elements in the integrity proof.

After verifying the search results, the owner can perform document ranking based on feature values in tuples of the results. Verifying server-side document ranking is an interesting topic for future work.

F. Proving to a Third Party

So far, we have discussed the owner's verification of cloud indexing results. Now we describe how our scheme allows both the data owner and the cloud to prove to a third party.

To prove a misbehaving cloud, the owner sends a search result and its proof to a third party, who can verify the signature made by the cloud and perform the same checks as the owner as described in Section III-E. The difference is that the third party's verification of correctness and integrity proof is computationally more expensive, because the third party does not know $\phi(n)$ and has to perform large modular exponentiation. If the cloud intentionally uses the wrong inverted indices for the query, the owner can convince the third party by asking the cloud to present the inverted indices to the third party.

On the other hand, the owner cannot frame an honest cloud. This is because the cloud signs its output and the owner cannot change the content of search results and proofs. Also, if the owner falsely claims that the cloud returns results for irrelevant query keywords, the cloud can present the owner's signed query requests and signed inverted indices. The third party can easily verify the correctness of results by performing intersection on the inverted indices.

IV. Implementation

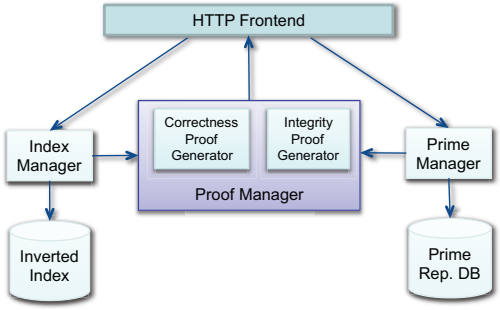

Fig. 4: The architecture of verifiable search service.

We have implemented our approach on both Linux and Mac OS X platforms. The prototype consists of approximately 11,000 lines of C++ code, including 4,000 lines of headers. We use the Lemur toolkit [29] for an initial parsing of documents, which removes stop words and performs stemming. We then generate inverted index files based on Lemur index, and compute the values of RSA accumulators and prime representatives for all index terms. We use the NTL library [30] for modular exponentiation, inverse computation and multiplication of large integers. For compressed Bloom filters, we use a publicly available arithmetic coding compressor [31].

Figure 4 illustrates the architecture of our verifiable online search service. A client interacts with the service through an HTTP frontend, which forwards the request to an `index manager` and a `prime manager`. The `index manager` retrieves data from the inverted index and constructs the query result. The `prime manager` retrieves prime representatives for the corresponding inverted index. Both the query result and the prime representatives are fed into a `proof manager` for generating correctness and integrity proofs. In the implementation, these managers run on different CPU cores for maximum concurrency.

The `prime manager` is introduced to allow quick computation of proofs for indexing results, which accesses precomputed prime representatives. For large datasets, prime representatives and accumulators need extensive computation. We have implemented a parallel program to perform precomputing using MPI. For this task, the simple strategy of balancing the number of index terms on each MPI process is

not efficient, because different terms can have a large variance in the size of index data. As a result, we choose to balance the number of index records for each MPI process, which results in higher speedup.

V. Evaluation

In this section, we evaluate our verifiable index by comparing the performance of the following schemes:

- **Accumulator.** This is the baseline scheme that builds correctness proofs with aggregated membership witnesses and constructs integrity proofs using the aggregated nonmembership witnesses.
- **Bloom.** This scheme [22] uses Bloom filters for constructing integrity proofs and employs aggregated membership witnesses for correctness proofs and check elements.
- **Interval Accumulator.** This scheme improves the `Accumulator` scheme with interval-based aggregated (non)membership witnesses as proofs for large sets.
- **Hybrid.** Our proposed method, based on the `Interval Accumulator` scheme, builds integrity proofs as a combination of aggregated nonmembership witnesses and Bloom filters.

All these schemes use the same optimization for unknown keyword and single keyword queries as discussed in Section III-D, and they all build correctness and integrity proofs in parallel. Additionally, all schemes use `prime manager` with pre-computed prime representatives.

A. Settings and Datasets

Experiments were performed on a Linux cluster and a Mac OS X 10.9.2 laptop. The Linux cluster consists of 15 machines connected with Gigabit Ethernet, where each node has two six-core Intel Xeon E5645 2.4 GHz CPU, 64 GB memory. The laptop machine has an Intel Core i7 2.9 GHz CPU and 16 GB memory. Our RSA accumulator is based on 1024-bit RSA modulo. Each document ID is represented by a 32-bit integer and the inverted index contains TF weights of terms, which are also 32-bit integers. For Bloom filters, the optimal parameter of one hash function is chosen. We choose 100 as the fixed set size for interval-based witnesses, as our micro benchmark shows that (non)membership witnesses can be computed within a few milliseconds. For all experiments, the average number of 10 runs is reported.

We use two datasets in our evaluation. For both datasets, we use the stop words list from Mallet [32], which consists of 823 words.

- **Enron email.** This dataset [1] consists of 517,424 email messages with a total size of more than 2.5 GB. There are 1,672,898 unique terms with an average document frequency of 144.1.
- **20-newsgroup.** This dataset[2] consists of 19,997 newsgroup documents with a total size of 90.5 MB. There are 185,910 unique terms and the average document frequency is 140.6.

To search the Enron dataset, we use a public set of queries [3]. In the experiments, we use a set of 24 queries to evaluate the performance for generating and validating proofs. The number of query keywords ranges from one to three: two single-keyword queries, six three-keyword queries, and 16 two-keyword queries. Among these queries, two of them (one two-keyword and one three-keyword queries) contain unknown search keywords.

B. Overall Results

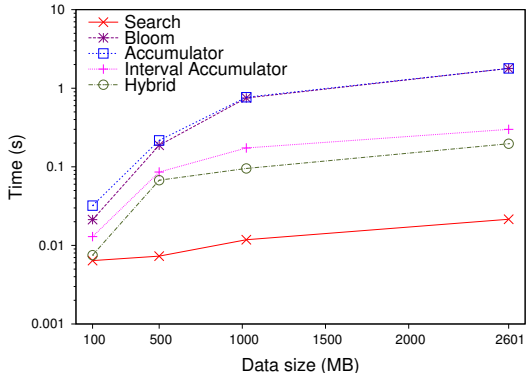

Fig. 5: Comparison of proof generation times for all schemes on Intel Core i7.

1) Comparison of Proof Generation Times: We first compare the performance of all schemes by varying the size of the Enron dataset. Figure 5 illustrates the proof generation times of all schemes, as well as the time for searching the index. Not surprisingly, searching index is much faster than generating proofs, only up to 0.022s. The proposed `Hybrid` scheme performs the best with an average of 0.197s to produce proofs for the whole 2601 MB Enron dataset, which is well enough for online services. `Interval Accumulator` achieves the second best, with an average of 0.300s for the whole Enron dataset. `Bloom` and the baseline `Accumulator` perform similarly, with an average of 1.78s for generating proofs for the whole dataset.

The inferior performance of both `Bloom` and `Accumulator` schemes are caused by generating witnesses for large sets. For instance, one of query used in our evaluation is "Rescheduling Mtg Mary", which corresponds to three inverted indices of size 41,269, 2,795, and 3,227, respectively. These three words all appear in 31 documents. Generating three membership witnesses for those large indices takes a total of 5.6s. In contrast, our interval-based witnesses only take 0.036s to compute. As a result, `Interval Accumulator` and `Hybrid` schemes are up to 83.2% and 88.9% better than both `Bloom` and `Accumulator`.

The `Hybrid` scheme is better than `Interval Accumulator` because of the use of Bloom filter

[1] http://www.cs.cmu.edu/~enron/

[2] http://kdd.ics.uci.edu/databases/20newsgroups/

[3] https://dbappserv.cis.upenn.edu/spell/

based integrity proofs, which are faster to generate than those sets with many check elements in the `Interval Accumulator` scheme.

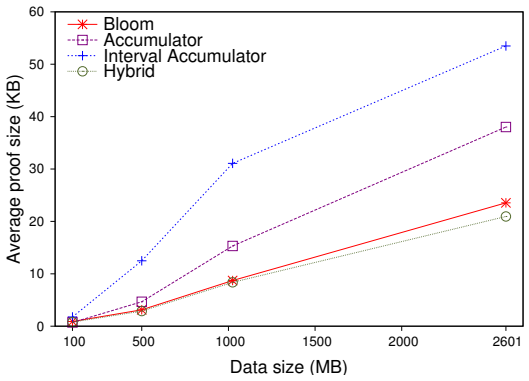

Fig. 6: Comparison of proof sizes for all schemes.

2) Comparison of Proof Sizes: Figure 6 illustrates the proof sizes of all schemes. Again, the proposed `Hybrid` scheme performs the best. The `Hybrid` scheme consumes less space than `Bloom` because for search results with few check elements, aggregated witnesses are more space efficient than Bloom filters. The `Accumulator` scheme suffers from unbounded check elements, thus consuming more space than `Bloom`. Finally, the `Interval Accumulator` uses more space than `Accumulator` due to interval-based witnesses.

TABLE I: The average proof verification times (in seconds) of all queries for the hybrid scheme on Core i7.

	100MB	**500MB**	**1000MB**	**2601MB**
default	0.0083	0.097	0.162	0.457
with prime	0.0052	0.0015	0.016	0.190

3) Verification Times: Table I illustrates the verification times for the `Hybrid` scheme are within 0.5s. Because the current proof does not contain prime representatives for index data, the client needs to compute these primes, resulting in longer verification time than proof generation. In comparison, the table also shows the verification times with these prime representatives, which are indeed shorter than generating proofs. For 500 MB data size, more interval-based proofs are generated, consuming less time than 100 MB case. This experiment shows a tradeoff between verification time and proof size, which we have not explored.

C. Study of Unknown Keyword Optimization

This experiment uses ten unknown keywords to compare the generation times of nonmembership witnesses and interval-based witnesses. Figure 7 illustrates that interval-based witnesses are about two orders of magnitude faster than nonmembership witnesses, using less than one millisecond for all dictionary sizes. This is because we pre-compute witnesses for all intervals and at search time we only need to find the interval containing the unknown keyword. In contrast, the time for nonmembership witness increases with dictionary size, requiring

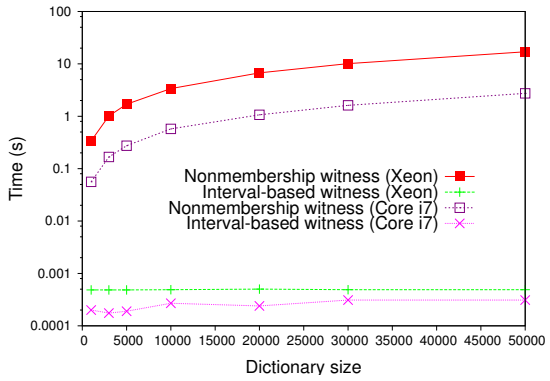

Fig. 7: Comparison of the average proof generation times of ten unknown keywords.

17.04 seconds for a 50,000-word dictionary on the Intel Xeon CPU. The increasing of time is due to exponentiation of larger exponent for bigger dictionaries. Because the server doesn't know the search keywords in advance, these nonmembership witnesses have to be computed online.

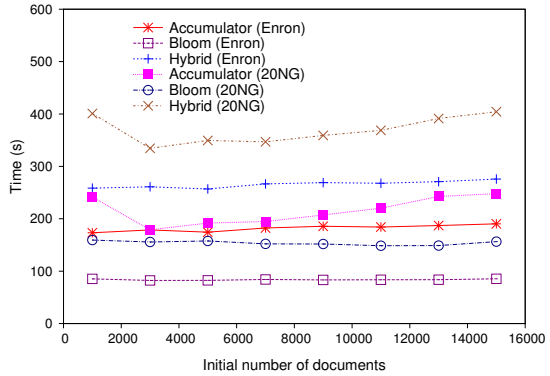

Fig. 8: Time to update accumulators when adding 2,000 new documents on Intel Core i7.

D. Incremental Update to Inverted Indices

We study the time for updating accumulators when adding 2,000 new documents to existing inverted indices and Figure 8 illustrates the results. For all three approaches, the time for updating accumulators remains almost constant with respect to increasing number of existing documents. For the `Accumulator` scheme, this is because updating accumulators only involves the newly added documents, which correspond to roughly the same number of tuples to be added. For the `Bloom` scheme, updating Bloom filters is more efficient than the `Accumulator` scheme and most of the time is for decompressing old Bloom filters and compressing the new ones. The `Hybrid` scheme needs to update both RSA accumulators and Bloom filters, thus requiring more time for processing.

E. Pre-computing

TABLE II: The average times of computing primes for 24 queries on Core i7.

	100MB	500MB	1000MB	2601MB
Time (s)	0.094	0.974	3.961	8.078

1) Effectiveness of Pre-computing: We calculate the average times of computing primes of all queries and the results are shown in Table II. Because we use the pre-computing strategy, this overhead for computing primes is saved for all four schemes. For the proposed hybrid scheme, the saving of proof time is 92.6-97.6%.

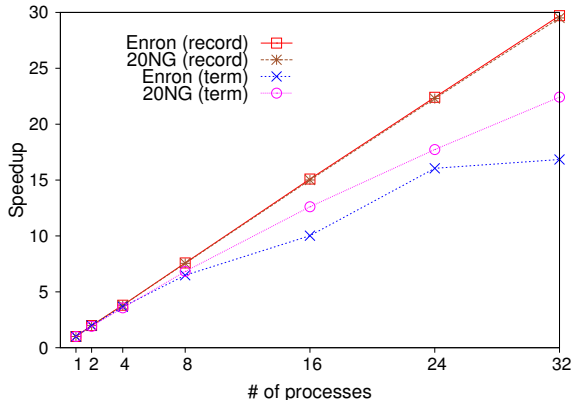

Fig. 9: Speedup of pre-computing primes and accumulators for both the Enron and the 20-newsgroup datasets on the Linux cluster.

2) Speedup of Pre-computing: This experiment studies the effectiveness of parallel computation of prime representatives and accumulators. We compare term-based and record-based load balancing strategies. As illustrated in Figure 9, the record-based strategy achieves close to linear speedup with increasing number of processes, because load is equally distributed on all processes. In comparison, the speedup for term-based strategy is significantly lower for more than 16 processes. This is because the number of records for different terms are skewed, resulting in considerable load imbalance among processes.

VI. Related Work

This section summarizes related work as follows.

A. Verifiable Computing

Verifiable computing was first introduced in [33]. The idea is for a computationally weak client to outsource computation to a server, such that the server performs the computation of a specified function F with given input, and outputs results together with a correctness proof. Such a protocol works by transforming F into a Boolean circuit and using fully-homomorphic encryption (FHE) [34]. Verifiable computing is still impractical despite many recent advances [5]–[9]. For instance, Pinocchio [9] has 10^5-10^6X overhead for generating proofs, and it takes Pinocchio 9.4ms on 2.67 GHz Intel Core i7 to produce proof for SHA-1 hash of a short string. We take a different approach by generating proofs for set intersections, where the keyword search computation has no slowdown. As a result, the overhead for our approach is less than 10X, orders of magnitude faster than Pinocchio. On the other hand, verifiable computing offers a generic solution for many different applications other than keyword search.

Braun et al. [35] point out that previous verifiable computing does not consider states and propose a system called Pantry that includes a block store. Pantry supports a verifiable database using CQL [36], which is a subset of SQL and does not include keyword-based search.

To verify query results from outsourced databases, Sion [16] introduced proofs for each executed *batch* of queries, providing probabilistic assurance that the queries were performed correctly over their entire datasets. In contrast, our work provides proof for each query and does not require the client to possess segments of queried data. The verifiable database introduced by Benabbas et al. [15] allows a client to query any cell values in an outsourced database with proofs, but does not support multi-keyword queries as in this paper.

Finally, Quin [37] delegates client computation to several servers and guarantees a correct answer as long as a single server is honest. In contrast, our verifiable index works with a singer server.

B. Authenticated Data Structures

There have been a number of studies on authenticating membership queries based on different cryptographic assumptions. For instance, several authenticated data structures are based on cryptographic hashing [17], [38], [39], with $O(\log n)$ proof size. Based on the strong RSA assumption, a number of approaches [18], [19] use RSA accumulators to achieve $O(1)$ proof size, which is extensively used in this paper.

Assuming the discrete logarithm problem is hard, zero-knowledge sets provide either membership or nonmembership witnesses for a value [40]. Our verifiable index is based on the aggregated witnesses, in contrast to individual witness of previous work [20], [40]. Proofs for set intersections using compressed Bloom filters and RSA accumulators were presented in [22]. This work differs by using aggregated nonmembership witnesses and interval-based witnesses. Additionally, we have shown that it is more advantageous to use accumulators than Bloom filters when the size of set difference is small. Verifying set operations for bilinear-map accumulators was proposed in [41]. Different from these studies, we use interval-based witnesses to reduce proof time and evaluate keyword search on real datasets.

VII. Conclusion

We have designed and implemented a hybrid approach for quickly verifying cloud search results, where the owner may outsource all the data without keeping local copies. Specifically, the cloud side search is modeled as set intersections. We design aggregated (non)membership witnesses for sets based on RSA accumulators, and use these witnesses to construct proofs of set intersections. In order to reduce both proof

generation time and proof size, we perform a number of optimizations. Experimental results on real data demonstrate that the proposed hybrid scheme generates proofs 83.2% faster than previous Bloom filter approach with a smaller proof size, and supports incremental updates with constant cost.

There are some future directions we plan to explore. This work does not consider the privacy of index data, which we believe can be enhanced with techniques from searchable encryptions [42], [43]. We would also like to compare the performance with another cryptographic system, i.e., bilinear-map accumulators [41].

ACKNOWLEDGMENT

We thank anonymous referees of ACSAC and IPDPS for their helpful comments on earlier drafts of this paper. This work was partially supported by the National Basic Research Program of China (973 Program, No. 2015CB352400), Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), and NSFC (Grant No. 61272099, 61261160502, 61202025, 61428204).

REFERENCES

[1] A. Ferdowsi, "S3 data corruption?" https://forums.aws.amazon.com/thread.jspa?threadID=22709, 2008.

[2] E. Protalinski, "Google fired employees for breaching user privacy," http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, 2010.

[3] L. Babai, "Trading group theory for randomness," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1985, pp. 421–429.

[4] S. Arora and S. Safra, "Probabilistic checking of proofs: a new characterization of NP," *Journal of the ACM*, vol. 45, no. 1, pp. 70–122, Jan. 1998.

[5] G. Cormode, J. Thaler, and K. Yi, "Verifying Computations with Streaming Interactive Proofs," *Proceedings of the VLDB Endowment*, vol. 5, no. 1, pp. 25–36, 2011.

[6] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Proc. of USENIX Security*, Bellevue, WA, 2012, pp. 253–268.

[7] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish, "Resolving the conflict between generality and plausibility in verified computation," in *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 71–84.

[8] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, 2013, pp. 223–237.

[9] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio : Nearly Practical Verifiable Computation," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013, pp. 238–252.

[10] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, 2004, pp. 223–238.

[11] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*, ser. SpringerBriefs in Computer Science. Springer New York, 2011, vol. 10.

[12] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor : Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011, pp. 203–216.

[13] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer : Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, 2005, pp. 1–16.

[14] "Trusted Computing Group (TCG)," http://www.trustedcomputinggroup.org/, 2013.

[15] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable Delegation of Computation over Large Datasets," in *Proc. of CRYPTO*, Santa Barbara, CA, USA, 2011, pp. 111–131.

[16] R. Sion, "Query Execution Assurance for Outsourced Databases," in *VLDB*, Trondheim, Norway, 2005, pp. 601–612.

[17] M. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX)*, 2001, pp. 68–82.

[18] M. T. Goodrich, R. Tamassia, and J. Hasic, "An Efficient Dynamic and Distributed Cryptographic Accumulator," in *Proc. of Information Security Conference (ISC)*, Sao Paulo, Brazil, 2002, pp. 372–388.

[19] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated Hash Tables," in *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, Alexandria, Virginia, USA, 2008, pp. 437–448.

[20] J. Li, N. Li, and R. Xue, "Universal Accumulators with Efficient Nonmembership Proofs," in *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2007, pp. 253–269.

[21] N. Baric and B. Pfitzmann, "Collision Free Accumulators and Fail Stop Signature Schemes Without Trees," in *Proc. of Eurocrypt*, 1997, pp. 480–494.

[22] R. Morselli, B. Bhattacharjee, J. Katz, and P. Keleher, "Trust-Preserving Set Operations," in *IEEE INFOCOM*, 2004, pp. 2231–2241.

[23] J. Benaloh and M. D. Mare, "One-Way Accumulators: A Decentralized Alternative to Digital Signatures," in *Proc. of Eurocrypt*, 1993, pp. 274–285.

[24] R. Gennaro, S. Halevi, and T. Rabin, "Secure Hash and Sign Signatures without the Random Oracle," in *Proc. of Eurocrypt*, 1999, pp. 123–139.

[25] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-3," in *TREC-3*, 1994.

[26] T. Qin, T.-Y. Liu, J. Xu, and H. Li, "LETOR: A Benchmark Collection for Research on Learning to Rank for Information Retrieval," *Information Retrieval Journal*, vol. 13, no. 4, pp. 346–374, 2010.

[27] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[28] M. Mitzenmacher, "Compressed Bloom filters," in *ACM symposium on Principles of distributed computing (PODC)*, Oct. 2001, pp. 144–150.

[29] The Lemur Project, http://www.lemurproject.org/, 2012.

[30] V. Shoup, "NTL: A library for doing number theory," http://www.shoup.net/ntl/.

[31] A. Moffat, "The Arithmetic Coding Page," http://ww2.cs.mu.oz.au/~alistair/arith_coder/, 2013.

[32] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[33] R. Gennaro, C. Gentry, and B. Parno, "Non-Interactive Verifiable Computing : Outsourcing Computation to Untrusted Workers," in *Proc. of CRYPTO*, 2010, pp. 465–482.

[34] C. Gentry, "Computing arbitrary functions of encrypted data," *Communications of the ACM*, vol. 53, no. 3, pp. 97–105, Mar. 2010.

[35] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *SOSP*, 2013, pp. 341–357.

[36] "Cassandra CQL," http://cassandra.apache.org/doc/cql/CQL.html.

[37] R. Canetti, B. Riva, and G. N. Rothblum, "Practical Delegation of Computation using Multiple Servers," in *CCS*, Chicago, Illinois, USA, 2011, pp. 445–454.

[38] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the Correctness of Memories," *Algorithmica*, pp. 90–99, 1995.

[39] M. Naor and K. Nissim, "Certificate Revocation and Certificate Update," in *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 217–228.

[40] S. Micali, M. Rabin, and J. Kilian, "Zero-Knowledge Sets," in *FOCS*, 2003, pp. 80–91.

[41] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Optimal Verification of Operations on Dynamic Sets," in *Advances in Cryptology*, 2011, pp. 91–110.

[42] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of the 13th ACM conference on Computer and Communications Security (CCS)*, Alexandria, Virginia, 2006, pp. 79–88.

[43] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, 2012, pp. 965–976.