Verification of Web-Content Searching Through Authenticated Web Crawler

Project Report

Duy Nguyen

Department of Computer Science

Brown University

Abstract

In this project, we study the security properties of web-content searching by using *three-party authenticated data structure* model. We present an implementation of an *authenticated web crawler*, a trusted program that computes a *digest* of a collection of web pages. This digest is later used to check the integrity of result of *conjunctive keyword search* queries, an important kind of query supported by search engines.

1. Introduction

Modern web search engines (e.g. Google, Bing, Yahoo) play an essential part in our life. Without them, it is difficult to imagine how people find information on Internet these days. However, if we look at them from the security point of view, it is natural to ask a question: Should we trust the results from search engines? This question is important for people searching for sensitive information, such as NYSE financial data. In this project, we aim to build a prototype that addresses this issue using three-party authenticated data structure model.

In three-party authenticated data structure model, we have the following entities:

- *Owner* owns the data (e.g. a collection of web pages in our context). The owner has a trusted crawler that produces an *inverted index*, an authenticated data structure and a *digest* on top of the collection.
- *Server* stores all web pages in the collection as well as the inverted index, authenticated data structure and digest computed by the crawler, as mentioned above. Server also returns answers to queries coming from clients along with cryptographic proofs for the correctness of its answers.
- *Client* queries server and verifies the results by using the proofs sent by the server.

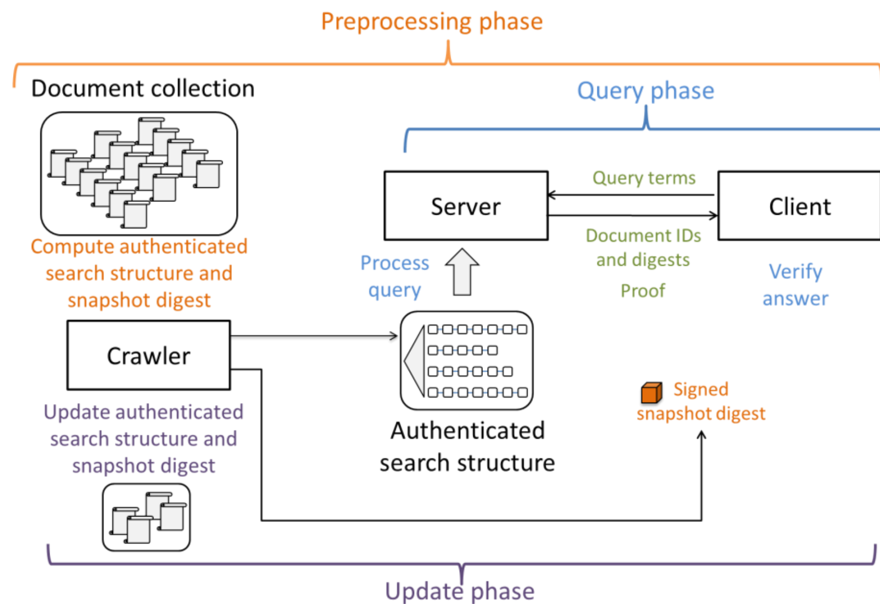

Figure 1: The three-party authenticated data structure model

2. Cryptographic background

We use the theoretical results of Papamanthou, Triandopoulos and Tamassia in CRYPTO 2011 [9]. We outline here the main ideas and components of the protocol. For detail construction and proof, please refer to their paper.

In inverted index data structure, each query term is mapped to a *posting list*, which is just a list of IDs of the documents containing the term. In order to answer a conjunctive keyword search query, we need to find a set intersection of posting lists of all terms in the query. We can think of a posting list as a set of integers. Each integer is the document ID, as described above.

The authenticated data structure computed by the crawler is composed of two parts:

- *Accumulators,* which are *bilinear-map accumulators,* for all terms (sets) in the inverted index.
- A *Merkle tree* [5] built on top of these accumulators. Each leaf node of the tree is an *SHA-256* digest of an accumulator.

The server, when answering conjunctive keyword queries, first uses inverted index to find the intersection. After that, it uses the authenticated data structure described above to compute the proof for the intersection. The proof is composed of four parts:

- Coefficients of the polynomial representing the intersection.
- Accumulators of all terms of the query. Each accumulator is attached with its full path in Merkle tree.
- *Subset witnesses*, which are also bilinear-map accumulators, serve as a proof that the intersection is a subset of all sets (terms) involved in the query.
- *Completeness witnesses*, which are also bilinear-map accumulators, serve as a proof that if we take the intersection from each set, all sets are disjoint.

The client is supposed to have very limited computing resources. It stores only the root node of the Merkle tree. After receiving the intersection and proof from the server, it verifies the result as below:

- With the intersection, it uses a randomized algorithm to verify the coefficients of the polynomial representing that intersection.
- With the root node of the Merkle tree, it verifies all accumulators of all terms of the query. Recall that each accumulator is attached with its full path in Merkle tree, as mentioned above.
- It uses *bilinear pairing* algorithm to verify subset witnesses
- It uses bilinear pairing algorithm to verify completeness witnesses

Again, for detail constructions and proofs of correctness of the protocol please refer to [9].

3. Implementation

Our prototype makes use of several open source libraries for cryptographic primitives and client/server communication. The following packages are used:

- OpenSSL [8]: SHA-256 used to build Merkle tree
- DCLXVI [2]: bilinear pairing algorithm
- NTL [7]: polynomial arithmetic operations
- LiDIA [4]: arithmetic operations in prime field
- Apache Thrift [1]: RPC calls between clients & server

3.1. Merkle Tree

Implementing Merkle tree requires a collision resistant hash function to recursively compute value for each node of the tree. We found that *SHA-256* is sufficient for our purposes, and OpenSSL is the de-factor standard library for classical cryptographic primitives, so we chose it to build our prototype.

3.2. *Zp* and polynomials over *Zp*

One of the key ideas of our protocol is to represent sets as polynomials over prime field *Zp* and to make use of *Extended Euclidean* algorithm for computing witnesses. So the implementation of *Zp* and polynomials over *Zp* must be very efficient so that the whole system can achieve practical performance.

Implementing software for multi precision arithmetic operations on big numbers has been an active research area for years. At this moment, GNU GMP [3], an open source library, is the de-factor standard for this purpose. It provides primitives for higher-level development of complex feature, such as prime field *Zp* and polynomials over *Zp*.

Currently, NTL is the fastest implementation of polynomials over *Zp* we are aware of. Although it supports several features in Number Theory, polynomial arithmetic is still its strongest one. It uses GMP as "kernel" for handling multi precision arithmetic, and provides optimal implementation of common operations on polynomials: addition, multiplication, Extended Euclidean… Our prototype uses it at the server where we compute the witnesses without knowing the secret key. However, it has one drawback: it is not thread-safe. So two operations required by our protocol, multiplication and Extended Euclidean, cannot be parallelized.

LiDIA [4] is another library for doing Number Theory. Like NTL, it also makes use of GMP to handle operations on big numbers. It also supports *Zp* as well as polynomials over *Zp*. However, its polynomial module is actually the early version of NTL which only supports basic algorithms. But unlike NTL, it is thread safe, so we chose it to implement our crawler where we need fast performance in *Zp* to compute accumulators for terms during setup and update phases.

3.3. Bilinear pairing

Because our protocol (especially the verification step at the client) is primarily based on bilinear pairing, this section outlines the design decision made when implementing this important primitive.

Recall that in bilinear pairing, $G1$ and $G2$ are cyclic multiplicative groups of prime order p having generators $g1$ and $g2$ respectively. GT is also a cyclic multiplicative group with the same order p. If $G1$ and $G2$ are the same, we have a *symmetric bilinear pairing*. Otherwise, we have *asymmetric bilinear pairing*.

A bilinear pairing is a function $e: G1 \times G2 \rightarrow GT$ with the following properties:

1. Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P \in G1$, $Q \in G2$ and $a,b \in Zp$
2. Non-degeneracy: $e(g1, g2) \neq 1$
3. Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P \in G1$, $Q \in G2$

A bilinear pairing library should provide users abilities to perform arithmetic operations in $G1$, $G2$, GT, and ideally, in Zp as well. However, most libraries implement Zp in a very simple form to speedup performance. Hence, users must use external library for arithmetic operations in Zp and manage to make these libraries talk to each other.

The development of bilinear pairing software in open source and academic community is still at its early stage. There are a few libraries available, some are faster than others, some offer more features than others. Among them, DCLXVI supports all operations required by our protocol and has acceptable performance, so we chose it for our implementation. However, the APIs are designed in an abstract manner to let us switch to new library easily if there is such one available.

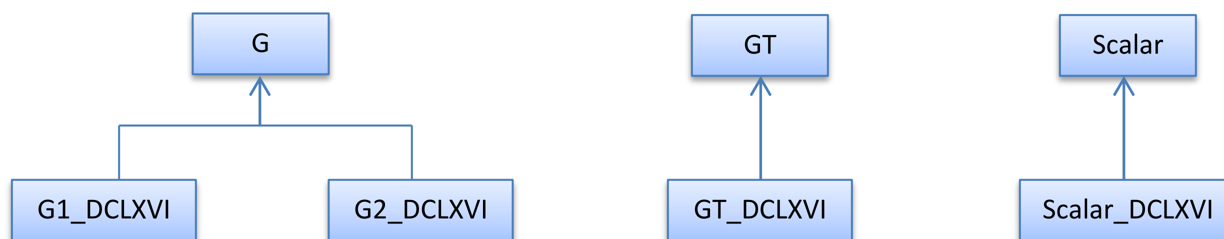

Figure 2: Class diagrams of major components of bilinear pairing implementation

We implement *G* and *GT* as *abstract interfaces*, they define the methods required by our protocol. Any bilinear pairing library that supports these methods can be used in our prototype. There are two packages satisfying our requirements: Stanford PBC [9] and DCLXVI. We chose the latter because it is 10 times faster than the first one to compute a bilinear pairing operation. *G1_DCLXVI*, *G2_DCLXVI* and *GT_DCLXVI* are C++ wrapper classes of DCLXVI library which implements all methods defined by interfaces *G* and *GT*

As mentioned above, prime field *Zp* is often implemented in bilinear pairing library in a very simple form. In most libraries, with prime order of 256 bits, an element in *Zp* is represented by an array of four 64-bit unsigned integers. However, our protocol requires some computation in *Zp*, so we define an abstract interface for it, called *Scalar*. Similarly, *Scalar_DCLXVI* is C++ wrapper class of DCLXVI implementation of *Zp*.

The abstract interface of *G* is defined as below:

```
/*
 * Abstract interface of an element in group G1/G2
 */

class G {
    public:
        // Constructor, the object is initialized as the generator
        G();

        // Destructor
        virtual ~G() = 0;

        // Assignment operator
        virtual G& operator=(const G& e) = 0;

        // Call this function to become a random element of the group
        virtual void generateRandom() = 0;

        // Call this function to become the identity element of the group
        virtual void becomeIdentity() = 0;

        // Call this function to become the generator element of the group
        virtual void becomeGenerator() = 0;

        // Check if this object and e are the same element
        virtual int  isEqual(const G *e) = 0;

        // Multiplication: result = this * e
        virtual void doMultiplication(G *e, G *result) = 0;

        // Power: result = this ^ scalar
        virtual void doPower(const Scalar *scalar, G *result) = 0;

        // Import the underlying implementation to this object
        virtual void importObject(const void *obj) = 0;

        // Export this object to the underlying implementation
        virtual void exportObject(void *obj) const = 0;

        // Size of this object in byte
        virtual size_t getSize() const = 0;
```

```
        // Serialize this object to a read-only byte buffer
        virtual char* getByteBuffer() const = 0;

        // Import object from file
        virtual void readFromFile(ifstream &inFile) = 0;

        // Export object to file
        virtual void writeToFile(ofstream &outFile) const = 0;
};
```

The abstract interface of *GT* is defined as below:

```
/*
 * Abstract interface of an element in group GT
 */

class GT {
    public:
        // Constructor
        GT();

        // Destructor
        virtual ~GT() = 0;

        // Assignment operator
        virtual GT& operator=(const GT& e) = 0;

        // Multiplication: result = this * e
        virtual void doMultiplication(GT *e, GT *result) = 0;

        // Power: result = this ^ scalar
        virtual void doPower(const Scalar *scalar, GT *result) = 0;

        // Check if this object and e are the same element
        virtual int isEqual(const GT *e) = 0;

        // Import the underlying implementation to this object
        virtual void importObject(const void *obj) = 0;

        // Export this object to the underlying implementation
        virtual void exportObject(void *obj) const = 0;

        // Import object from file
        virtual void readFromFile(ifstream &inFile) = 0;

        // Export object to file
        virtual void writeToFile(ofstream &outFile) const = 0;
};
```

The abstract interface of *Scalar* is defined as below:

```
/*
 * Abstract interface of an element in Zp
 */

class Scalar {
    public:
        // Constructor
        Scalar();

        // Destructor
        virtual ~Scalar() = 0;

        // Assignment operator
        virtual Scalar& operator=(const Scalar& s) = 0;

        // Generate a random number
        virtual void generateRandom() = 0;

        // Get the bit at position pos
        virtual int getBit(unsigned int pos) const = 0;

        // How many bits used to represent this number
        virtual int getSize() const = 0;

        // Import the underlying implementation to this object
        virtual void importObject(const void *obj) = 0;

        // Export this object to the underlying implementation
        virtual void exportObject(void *obj) const = 0;

        // Import an NTL object to this object
        virtual void importNTLObject(const NTL::ZZ_p &obj) = 0;

        // Export this object to an NTL object
        virtual void exportNTLObject(NTL::ZZ_p &obj) const = 0;

        // Import an LiDIA object to this object
        virtual void importLiDIAObject(const LiDIA::bigmod &obj) = 0;

        // Export this object to an LiDIA object
        virtual void exportLiDIAObject(LiDIA::bigmod &obj) const = 0;

        // Print this number in raw hex data
        virtual void print() const = 0;

        // Print this number in user-friendly format
        virtual void printPretty() const = 0;
```

```
        // Import object from file
        virtual void readFromFile(ifstream &inFile) = 0;

        // Export object to file
        virtual void writeToFile(ofstream &outFile) const = 0;
};
```

Note that this interface has 2 groups of methods for importing/exporting *Zp* object implemented by external libraries. In our implementation, we use NTL and LiDIA libraries for arithmetic operations in *Zp* and polynomial over *Zp*. These two have their own implementation of *Zp*, so we have to be able to interact with them.

With these interfaces defined, it's straightforward to define the bilinear function:

```
class ASCAlgorithms{
    public:
        static ASCAlgorithms* getInstance();

        void pairing(GT *e, const G *e1, const G *e2);

    protected:
        ASCAlgorithms();

    private:
        static ASCAlgorithms *_instance;
};
```

ASCAlgorithms is the class that implements all algorithms required by our protocol. It is designed using *Singleton* design pattern and thread-safe. We only list *pairing()* method here, please refer to source code for the rest of the definition. Below is a toy example demonstrating how to use bilinear pairing:

```
  G *P = new G1DCLXVI(),  // New an element in G1, initialized as generator
    *Q = new G2DCLXVI();  // New an element in G2, initialized as generator
  P->generateRandom();    // P becomes a random element in G1
  Q->generateRandom();    // Q becomes a random element in G2

  Scalar *a = new ScalarDCLXVI(),  // New an element in Zp, initialized as 0
         *b = new ScalarDCLXVI();  // New an element in Zp, initialized as 0
  a->generateRandom();
  b->generateRandom();

  GT *E1 = new GTDCLXVI(),    // New an element in GT
     *E2 = new GTDCLXVI();    // New an element in GT

  ASCAlgorithms::getInstance()->pairing(E1, P, Q);    // E1 = e(P, Q)
  E1->doPower(a, E1);    // Compute e(P, Q)^a
  E1->doPower(b, E1);    // Compute e(P, Q)^ab
```

```
P->doPower(a, P);    // Compute Pᵃ
Q->doPower(b, Q);    // Compute Qᵇ
ASCAlgorithms::getInstance()->pairing(E2, P, Q);    // E2 = e(Pᵃ, Qᵇ)

if(E1.isEqual(E2)){
    cout<<"Correct pairing"<<endl;
} else {
    cout<<"Incorrect pairing"<<endl;
}
```

3.4. Client-Server communication

As described, our prototype includes 3 parts: crawler, client and server. The crawler produces authenticated data structure offline, and then outsources it to the untrusted server, which is usually deployed on the cloud. The clients send queries to server over untrusted network links (LAN, Internet) and verify the results by using the proofs, both sent by the server.

In our model, the connection between clients and server is stateless. That is, any client can query server at any time, and queries are independent from each other. This property suggests a straightforward approach for clients and server to communicate: *Remote Procedure Call (RPC)*, and our implementation chose *Apache Thrift* for this purpose.

Apache Thrift (originally developed by Facebook) is an open source software framework for scalable cross-language RPC. Like other frameworks, it requires users to provide a file which defines message format in its pre-defined syntax for all messages exchanged by clients and server. After that, a compiler parses this file, and generates skeletons for both server and client which take care of almost everything from networking communication to data marshaling. Users then provide the "real" implementations for both server and client.

Below is the service declaration and message format defined in Thrift syntax:

```
namespace cpp Query

// An element in Zp
struct TZp {
    1: list<i64> data
}

// An element in G1/G2 defined by DCLXVI library
struct TG{
    1: list<double> m_x,
    2: list<double> m_y,
    3: list<double> m_z,
    4: list<double> m_t
}
```

```
// Merkle tree node and full-path
struct TMerkleNode{
    1: i32 offset,
    2: list<byte> hash
}
struct TMerklePath{
    1: list<TMerkleNode> hashes
}

// Proof returned by server
struct TQueryProof{
    1: byte errorCode,          // Handling errors
    2: list<i64> docList,       // The intersection
    3: list<string> titleList, // URL titles, just for debugging
    4: TG iacc,                 // Accumulator of the intersection
    5: list<TZp> coeffs,        // Coefficients of polynomial
    6: list<TG> accs,           // Accumulators of query terms
    7: list<TG> sws,            // Subset witnesses
    8: list<TG> cws,            // Completeness witnesses
    9: list<TMerklePath> accHashPaths // Merkle tree paths of query terms

}

// The RPC service
service Query {
    TQueryProof query(1: string queryStr)
}
```

Here we define an RPC service named *Query,* having a single RPC call *query* which takes a string *queryStr* as input, and return a proof of type *TQueryProof*. Structure of the proof is self-explained in the listing above.

3.5. Parallel computation of witnesses

In our model, the server is the entity that handles all expensive computations. The main overhead lies in computation of subset witnesses and completeness witnesses of the proof. When the query terms are common words, their posting lists can be very large which make powering and multiplication operations in *G1*/*G2* time consuming.

Recall that both kinds of witnesses have the form of $g^{f(s)}$ where g is the generator of the group (*G1* for subset witnesses, *G2* for completeness witnesses), and $f(s)$ is a polynomial representing the set difference of term's posting list and the intersection of all posting lists in case subset witnesses or computed by Extended Euclidean algorithm in case completeness witnesses.

When computing a witness, the server, without the secret key s, first uses *Fast Fourier Transform (FFT)* algorithm to find coefficients of polynomial $f(s)$. After that, for each coefficient, given its degree and value, with access to the public key (which is an array of

elements in G1/G2), it raises corresponding element in public key to the power of coefficient. Finally, it multiplies all intermediate elements into a final one, which is the witness.

We observe that subset witnesses and completeness witnesses are independent from each other. So the first optimization is to compute them in parallel. Our implementation uses two *POSIX threads* for this computation.

Another observation is, when computing $g^{f(s)}$, after we have coefficients computed by FFT algorithm, powering each public key element to the power of corresponding coefficient can also be done in parallel because all coefficients are independent. However, we cannot allocate a POSIX thread for each powering operation. The reason is that if the posting list is large and the intersection is small, we will end up with a large degree polynomial which may have thousands of coefficients. Allocating too many threads will slow down the system because of too much context switching as well as resource exhaustion. Our approach for this problem is to allocate just N worker threads, N is configurable. If the polynomial has degree of M, each worker thread is responsible for M/N powering operations and $(M/N - 1)$ multiplication operations.

These optimizations greatly speed up server's time to compute the proof. The more CPU cores we have the better performance we achieve. In our experiment on an 8-core machine, using this worker thread model, performance is 8x faster than non-multithread implementation. Note that in an ideal deployment where the server is in the cloud, it would have access to much more computing resources, we can achieve better performance.

4. Experimental results

Our experiments are performed on an 8-core Xeon 2.93 CPU with 8GB of RAM with 64-bit Debian Linux installed. The following two parameters are used as metrics in our tests:

- *Total set size*: sum of the size of all posting lists of the query terms
- *Intersection size*: the size of the intersection of all posting lists of the query terms

We use data set from *Wall Street Journal (WSJ)* articles published from 03/1987 to 03/1992. This collection has 173,252 articles including 135,760 unique terms.

4.1. Server time

The main job of server is to compute the proof. This is the extra cost added to a standard search engine when it adopts our model, hence, we measure time spent for this task.

We build a query set consisting of 200 random conjunctive queries with two terms. Each query has intersection size in range [0, 762]. Queries include both rare and frequent terms. A term is considered frequent if it appears in 8000 documents or more.

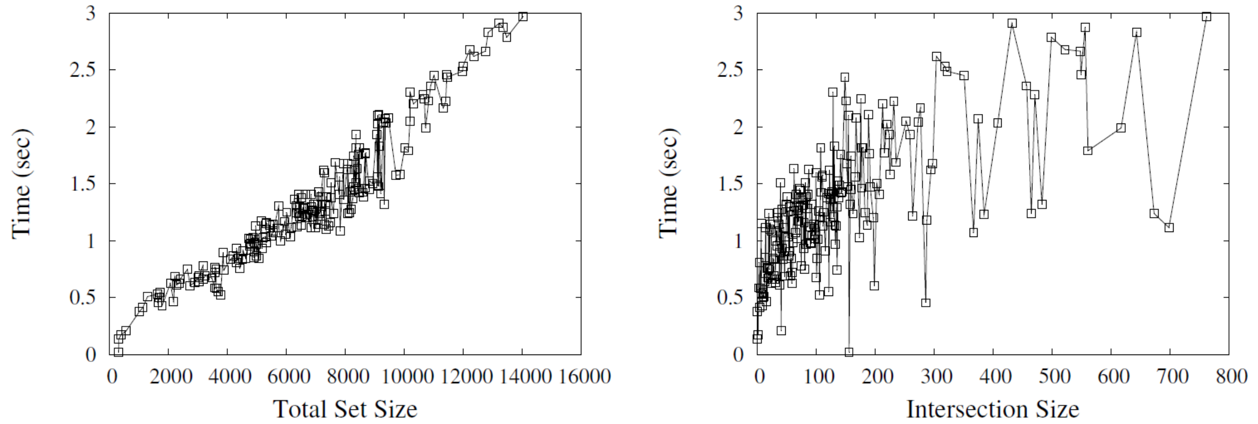

Figure 3: Server time measured in terms of total set size and intersection size

Figure 3 shows how total set size and intersection size affect server performance. We can see that server time linearly depends on total set size and does not depend on the intersection size.

4.2. Client time and proof size

When measuring time at client side, we split the time into two parts: time spent for verifying the integrity of posting lists (by using Merkle tree) and time spent to verify the correctness of intersection (by using bilinear pairing). In addition, we also measure the proof size sent by the server to clients.

As we can see in the Figure 4, both client time and proof size only depend on the intersection size, not on the total set size.

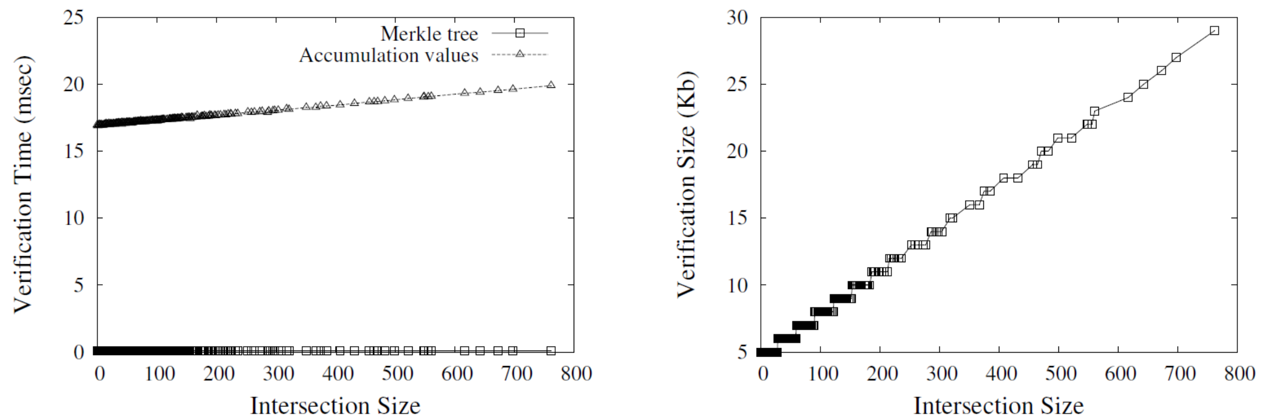

Figure 4: Client time and proof size only depend on intersection size

4.3. Update time

Our prototype supports updating the collection, namely adding documents and deleting documents. This update is done by the crawler and consists of 2 main tasks: updating the

accumulators of the affected terms and updating the Merkle tree from leaf node up to the root for all affected accumulators.

We pick a set of 1500 documents in WSJ collection which covers over 14% of the collection vocabulary. The following figure shows the time spent for updating as a function of number of terms to be updated. We can see that the time is linear in the number of terms.

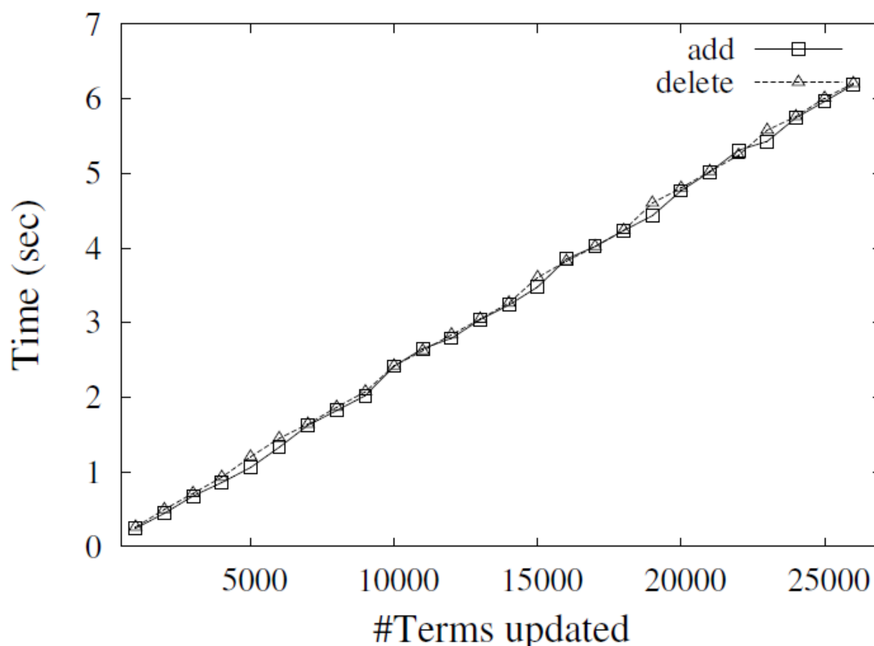

Figure 5: Time spent for updating authenticated data structure

5. Conclusion

We successfully implemented a prototype addressing the problem of verifying the integrity of results of conjunctive keyword searches. Our prototype shows a practical performance at the server side when being deployed on commodity hardware platform, and shows a very fast and bandwidth efficient at the client side in verification of the results.

6. Acknowledgment

I would like to thank Prof. Tamassia, Olya Ohrimenko and Babis Papamanthou for their patience and guidance during two semesters I've been working on this project. Especially, I would like thank Olya for her comments and ideas in various parts of my implementation. The experiment mentioned in section 4 of this report is done by her using my code. The graphs are extracted from the VLDB paper (in submission) in which we are co-authors.

7. References

[1] Apache Thrift (http://thrift.apache.org).

[2] DCLXVI library (http://www.cryptojedi.org/crypto). V. 20110718.

[3] GMP Lib (http://gmplib.org): The GNU MP Bignum Library

[4] LiDIA (http://www.cs.sunysb.edu/~algorith/implement/lidia/implement.shtml): A library for computational number theory. V. 2.3.

[5] R. C. Merkle. A certified digital signature. *CRYPTO*, 218-238, 1989.

[6] M. Naehrig, R. Niederhagen, and P. Schwabe. New software speed records for cryptographic pairings. *LATINCRYPT*, 109-123, 2010.

[7] NTL (http://www.shoup.net/ntl): A Library for doing Number Theory. V. 5.5.2.

[8] OpenSSL (http://www.openssl.org).

[9] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verication of operations on dynamic sets. *CRYPTO*, 91-110, 2011.

[10] PBC Library (http://crypto.stanford.edu/pbc): The Pairing-Based Cryptography Library.

[11] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.