Efficient Verification of Web-Content Searching Through Authenticated Web Crawlers

Michael T. Goodrich
UC Irvine
goodrich@ics.uci.edu

Duy Nguyen
Brown University
duy@cs.brown.edu

Olga Ohrimenko
Brown University
olya@cs.brown.edu

Charalampos Papamanthou
UC Berkeley
cpap@cs.berkeley.edu

Roberto Tamassia
Brown University
rt@cs.brown.edu

Nikos Triandopoulos
RSA Laboratories &
Boston University
nikos@cs.bu.edu

Cristina Videira Lopes
UC Irvine
lopes@ics.uci.edu

ABSTRACT

We consider the problem of verifying the correctness and completeness of the result of a keyword search. We introduce the concept of an *authenticated web crawler* and present its design and prototype implementation. An authenticated web crawler is a trusted program that computes a specially-crafted signature over the web contents it visits. This signature enables (i) the verification of common Internet queries on web pages, such as conjunctive keyword searches—this guarantees that the output of a conjunctive keyword search is *correct* and *complete*; (ii) the verification of the content returned by such Internet queries—this guarantees that web data is *authentic* and has not been maliciously altered since the computation of the signature by the crawler. In our solution, the search engine returns a cryptographic proof of the query result. Both the proof size and the verification time are proportional only to the sizes of the query description and the query result, but do not depend on the number or sizes of the web pages over which the search is performed. As we experimentally demonstrate, the prototype implementation of our system provides a low communication overhead between the search engine and the user, and fast verification of the returned results by the user.

1. INTRODUCTION

When we perform a web search, we expect that the list of links returned will be relevant and complete. As we heavily rely on web searching, an often overlooked issue is that search engines are outsourced computations. That is, users issue queries and have no intrinsic way of trusting the results they receive, thus introducing a modern spin on Cartesian doubt. This philosophy once asked if we can trust our senses—now it should ask if we can trust our search results. Some possible attack scenarios that arise in this context include the following:

1. A news web site posts a misleading article and later changes it to look as if the error never occurred.
2. A company posts a back-dated white paper claiming an invention after a related patent is issued to a competitor.
3. An obscure scientific web site posts incriminating data about a polluter, who then sues to get the data removed, in spite of its accuracy.
4. A search engine censors content for queries coming from users in a certain country, even though an associated web crawler provided web pages that would otherwise be indexed for the forbidden queries.

An Internet archive, such as in the Wayback Machine, that digitally signs the archived web pages could be a solution to detecting the first attack, but it does not address the rest. It misses detecting the second, for instance, since there is no easy way in such a system to prove that something didn't exist in the past. Likewise, it does not address the third, since Internet archives tend to be limited to popular web sites. Finally, it does not address the fourth, because such users would likely also be blocked from the archive web site and, even otherwise, would have no good way of detecting that pages missing from a keyword-search response.

From a security point of view, we can abstract these problems in a model where a query request (e.g., web-search terms) coming from an end user, Alice, is served by a remote, unknown and possibly untrusted server (e.g., online search engine), Bob, who returns a result consumed by Alice (e.g., list of related web pages containing the query terms). In this context, it is important that such computational results are *verifiable by the user*, Alice, with respect to their *integrity*. Integrity verifiability, here, means that Alice receives additional authentication information (e.g., a digital

Proceedings of the VLDB Endowment, Vol. 5, No. 10

signature from someone she trusts) that allows her to verify the integrity of the returned result. In addition to *file-level protection*, ensuring that data items (e.g., web contents) remain intact, the integrity of the returned results typically refers to the following three properties (e.g., [11]): (1) *correctness*, ensuring that any returned result satisfies the query specification; (2) *completeness*, ensuring that no result satisfying the query specification is omitted from the result, and (3) *freshness*, ensuring that the returned result is computed on the currently valid, and most recently updated data.

The ranking of query results is generally an important part of web searching. However, in the above scenarios, correctness, completeness, and freshness are more important to the user than a proof that the ranking of the results is accurate. Also, note that it is usually in the best interest of the search engine to rank pages correctly, e.g., for advertising. Thus in our context, we are interested in studying the problem of *integrity verification* of web content. In particular, we wish to design protocols and data-management mechanisms that can provide the user with a cryptographically verifiable proof that web content of interest or query results on this content are *authentic*, satisfying *all* the above three security properties: correctness, completeness, and freshness.

1.1 Challenges in Verifying Web Searching

Over the last decade, significant progress has been made on integrity protection for *management of outsourced databases*. Here, a database that is owned by a (trusted) source, Charles, is outsourced to an (untrusted) server, Bob, who serves queries coming from end users such as Alice. Using an *authenticated index structure*, Bob adds *authentication* to the supported database responses, that is, augments query results with authentication information, or *proof*, such that these results can be cryptographically verified by Alice with respect to their integrity. This authentication is done subject to a *digest* that the source, Charles, has produced by processing the currently valid database version. In the literature, many elegant solutions have been proposed focusing on the authentication of large classes of queries via short proofs (sent to the user) yielding fast verification times.

Unfortunately, translating such existing methods in the authenticated web searching problem is not straightforward, but rather entails several challenges. First, result verifiability is not well defined for web searching, because unlike the database outsourcing model, in web searching there is no clear data source and there is no real data outsourcing by someone like Charles. Of course, one could consider a model where each web-page owner communicates with the online search engine to deliver (current) authentication information about the web pages this owner controls, but clearly this consideration would be unrealistic. Therefore, we need an authentication scheme that is *consistent with the crawling-based current practices of web searching*.

Additionally, verifying the results of search engines seems particularly challenging given the large scale of this data-processing problem and the high rates at which data evolves over time. Indeed, even when we consider the static version of the problem where some portion of the web is used for archival purposes only (e.g., older online articles of the Wall Street Journal), authenticating general search queries for such rich sets of data seems almost impossible: How, for instance, is it possible to verify the completeness of a simple keyword-search query over a large collection of archival web pages? Existing authentication methods heavily rely on a total ordering of the database records on some (primary-key) attribute in order to provide "near-miss" proofs about the completeness of the returned records, but no such total order exists when considering keyword-search queries over text documents. This suggests that to prove completeness to a user the search engine will have to provide the user with "all supporting evidence"—all web contents the engine searched through—and let the user recompute and thus trivially verify the returned result. Clearly, this approach is also not scalable. Generally, Internet searching is a complicated "big data" problem and so is its authentication: We thus need an authentication scheme that produces proofs and incurs verification times that are *web-search sensitive*, that is, they *depend on the set of returned results and not on the entire universe of possible documents*.

Integrity protection becomes even more complicated when we consider web pages that frequently change over time. For instance, collaborative systems store large amounts of scientific data at distributed web-based repositories for the purpose of sharing knowledge and data-analysis results. Similarly, users periodically update their personal or professional web pages and blogs with information that can be searched through web search engines. In such dynamic settings, how is it possible to formally define the notion of freshness? Overall, we need an authentication scheme that is *consistent with the highly dynamic nature of web content*.

1.2 Prior Related Work

Despite its importance and unlike the well-studied problem of database authentication, the problem of web searching authentication has not been studied before in its entirety. To the best of our knowledge, the only existing prior work studies a more restricted version of the problem.

The first solution on the authentication of web searches was recently proposed by Pang and Mouratidis in PVLDB 2008 [20]. This work focuses on the specific, but very important and representative case, where search engines perform *similarity-based document retrieval*. Pang and Mouratidis show how to construct an authentication index structure that can be used by an untrusted server, the search engine, in the *outsourced database model* to authenticate text-search queries over documents that are based on similarity searching. In their model, a trusted owner of a collection of documents outsources this collection to the search engine, along with this authentication index structure defined over the document collection. Then, whenever a user issues a text-search query to the engine, by specifying a set of keywords, the engine returns the top r results (r being a system parameter) according to some well-defined notion of relevance that relates query keywords to documents in the collection. In particular, given a specific term (keyword) an inverted list is formed with documents that contain the term; in this list the documents are ordered according to their estimated relevance to the given term. Using the authentication index structure, the engine is able to return those r documents in the collection that are better related to the terms that appear in the query, along with a proof that allows the user to verify the correctness of this document retrieval.

At a technical level, Pang and Mouratidis make use of an elegant hash-based authentication method: The main idea is to apply cryptographic hashing in the form of a Merkle hash tree (MHT) [13] over each of the term-defined lists. Note

that each such list imposes a total ordering over the documents it contains according to their assigned score, therefore completeness proofs are possible. Pang and Mouratidis observe that the engine only partially parses the lists related to the query terms: At some position of a list, the corresponding document has low enough cost that does not allow inclusion in the top r results. Therefore, it suffices to provide hash-based proofs (i.e., consisting of a set of hash values, similar to the proof in a Merkle tree) only for *prefixes* of the lists related to the query terms. Pang and Mouratidis thus construct a novel *chained sequence of Merkle trees*. This chain is used to authenticate the documents in an inverted list corresponding to a term, introducing a better trade-off between verification time and size of provided proof.

To answer a query, the following process is used. For each term, documents are retrieved sequentially through its document list, like a sliding window, and the scores of these documents are aggregated. A document's score is determined by the frequency of each query term in the document. This parsing stops when it is certain that no document will have a score higher than the current aggregated score.

To authenticate a query, the engine collects, as part of the answer and its associated proof, a verification object that contains the r top-ranked documents and their corresponding MHT proofs, as well as all the documents in between that did not score higher but were potential candidates. For example, consider the following two lists:

$$\mathsf{term}_1: \quad \mathsf{doc}_1, \mathsf{doc}_5, \mathsf{doc}_3, \mathsf{doc}_2$$
$$\mathsf{term}_2: \quad \mathsf{doc}_2, \mathsf{doc}_5, \mathsf{doc}_3, \mathsf{doc}_4, \mathsf{doc}_1$$

If a query is "term_1 term_2", r is 1 and doc_1 has the highest score, then the verification object has to contain doc_2, doc_5, doc_3, doc_4 to prove that their score is lower than doc_1.

We thus observe four limitations in the work by Pang and Mouratidis: (1) their scheme is not fully consistent with the crawling-based web searching, since it operates in the outsourced database model; (2) their scheme is not web-search sensitive because, as we demonstrated above, it returns documents and associated proofs that are not related to the actual query answer, and these additional returned items may be of size proportional to the *number of documents in the collection*; (3) their scheme requires complete reconstruction of the authentication index structure when the document collection is updated: even a simple document update may introduce changes in the underlying document scores, which in the worst case will completely destroy the ordering within one or more inverted lists; (4) it is not clear whether and how their scheme can support query types different from disjunctive keyword searches.

1.3 Our Approach

Inspired by the work by Pang and Mouratidis [20], we propose a new model for performing keyword-search queries over online web contents. In Section 2, we introduce the concept of an *authenticated web crawler*, an authentication module that achieves authentication of general queries for web searching in a way that is consistent with the current crawling-based search engines. Indeed, this new concept is a program that like any other web crawler visits web pages according to their link structure. But while the program visits the web pages, it incrementally builds a space-efficient authenticated data structure that is specially designed to support general keyword searches over the contents of the web pages. Also, the authenticated web crawler serves as a trusted component that computes a signature, an accurate security snapshot, of the current contents of the web. When the crawling is complete, the crawler publishes the signature and gives the authenticated data structure to the search engine. The authenticated data structure is in turn used by the engine to provide verification proofs for any keyword-search query that is issued by a user, which can be verified by having access to the succinct, already published signature.

In Section 3, we present our authentication methodology which provides proofs that are web-search sensitive, i.e., of size that depends linearly on the query parameters and the results returned by the engine and logarithmically on the size of the inverted index (number of indexed terms), but it does not depend on the size of the document collection (number of documents in the inverted lists). The verification time spent by the user is also web-search sensitive: It depends only on the query (linearly), answer (linearly) and the size of the inverted index (logarithmically), not on the size of the web contents that are accessed by the crawler. We stress the fact that our methodology can support general keyword searches, in particular, *conjunctive keyword searches*, which are vital for the functionality of online web search engines today and, accordingly, a main focus in our work.

Additionally, our authentication solution allows for efficient updates of the authenticated data structure. That is, if the web content is updated, the changes that must be performed in the authenticated data structure are only specific to the web content that changes and there is no need to recompute the entire authentication structure. This property comes in handy in more than one flavors: Either the web crawler itself may incrementally update a previously computed authentication structure the next time it crawls the web, or the crawler may perform such an update on demand without performing a web crawling.

So far we have explained a three-party model where the client verifies that the results returned by the search engine are consistent with the content found by the web crawler. (See also Section 2.1 for more details). Our solution can also be used in other common scenarios involving two parties. First, consider a client that outsources its data to the cloud (e.g., to Google Docs or Amazon Web Services) and uses the cloud service to perform keyword-search queries on this data. Using our framework, the client can verify that the search results were correctly performed on the outsourced data. In another scenario, we have a client doing keyword searches on a trusted search engine that delivers the search results via a content delivery network (CDN). Here, our solution can be used by the client to discover if the results from the search engine were tampered with during their delivery, e.g., because of a compromised server in the CDN. (See Section 2.2 for more details on these two-party models).

Our authentication protocols are based on standard Merkle trees as well as on bilinear-map accumulators. The latter are cryptographic primitives that can be viewed as being equivalent to the former, i.e., they provide efficient proofs of set membership. But they achieve a different trade-off between verification time and update time: Verification time is constant (not logarithmic), at the cost that update time is no more logarithmic (but still sublinear). However, bilinear-map accumulators offer a unique property not offered by Merkle trees: They can provide constant-size proofs (and corresponding verification times) for *set disjointness*, i.e., in

order to prove that two sets are disjoint a constant-size proof can be used—this property is very important for proving completeness over the unordered documents that are stored in an inverted index (used to support keyword searches).

In Section 4, we describe implementation details of the prototype of our solution and present empirical evaluation of its performance on the Wall Street Journal archive. We demonstrate that in practice our solution gives a low communication overhead between the search engine and the user, and allows for fast verification of the returned results on the user side. We also show that our prototype can efficiently support updates to the collection. We conclude in Section 5.

1.4 Additional Related Work

A large body of work exists on *authenticated data structures* (e.g., [6, 15, 22]), which provide a framework for designing practical methods for query authentication: Answers to queries on a data structure can be verified efficiently through the computation of some short cryptographic proofs.

Research initially focused on authenticating membership queries [3] and the design of various authenticated dictionaries [8, 15, 24] based on extensions of Merkle's *hash tree* [13]. Subsequently, one-way accumulators [4, 17] were employed to design dynamic authenticated dictionaries [7, 21] that are based on algebraic cryptographic properties to provide optimal verification overheads.

More general queries have been studied as well. Extensions of hash trees have been used to authenticate various types of queries, including basic operations (e.g., select, join) on databases [6], pattern matching in tries [12] and orthogonal range searching [1, 12], path queries and connectivity queries on graphs and queries on geometric objects [9] and queries on XML documents [2, 5]. Many of these queries can be reduced to one-dimensional range-search queries which can been verified optimally in [10, 18] by combining collision-resistant hashing and one-way accumulators. Recently, more involved cryptographic primitives have been used for optimally verifying set operations [22].

Substantial progress has also been made on the design of generic authentication techniques. In [12] it is shown how to authenticate in the static case a rich class of search queries in DAGs (e.g., orthogonal range searching) by hashing over the search structure of the underlying data structure. In [9], it is shown how extensions of hash trees can be used to authenticate decomposable properties of data organized as paths (e.g., aggregation queries over sequences of objects) or any search queries that involve iterative searches over catalogs (e.g., point location). Both works involve proof sizes and verification times that asymptotically equal the complexity of answering queries. Recently, in [23], a new framework is introduced for authenticating general query types over structured data organized and accessed in the relational database model. By decoupling the processes of query answering and answer verification, it is shown how any query can be reduced (without loss of efficiency) to the fundamental problem of set-membership authentication, and that super-efficient answer verification (where the verification of an answer is asymptotically faster than the answer computation) for many interesting problems is possible. Set-membership authentication via collision-resistant hashing is studied in [24] where it is shown that for hash-based authenticated dictionaries of size n, all costs related to authentication are at least logarithmic in n in the worst case.

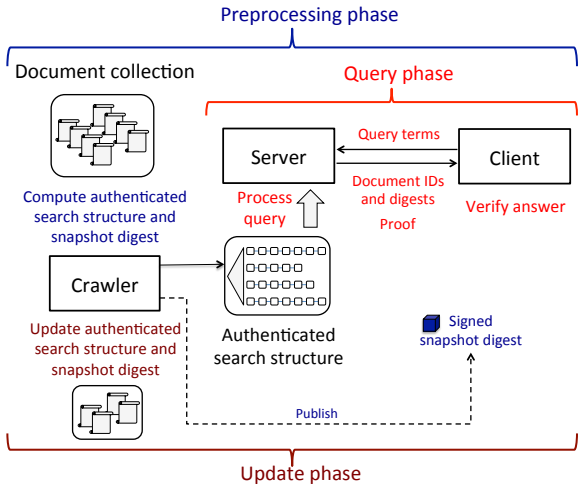

Figure 1: The three-party model as the main operational setting of an authenticated web crawler.

Finally, a growing body of works study the specific problem of authenticating SQL queries over outsourced relational databases typically in external-memory data management settings. Representatives of such works include the authentication of range search queries by using hash trees (e.g., [6, 9]) and by combining hash trees with accumulators (e.g., [10, 18]) or B-trees with signatures (e.g., [16, 19]). Additional work includes an efficient hash-based B-tree-based authenticated indexing technique in [11], the authentication of join queries in [25] and of shortest path queries in [26].

2. OUR MODEL

We first describe two models where our results can be applied allowing users to be able to verify searches over collections of documents of interest (e.g., a set of web pages).

2.1 Three-party Model

We refer to the three-party model of Figure 1. (We note that although relevant, this model is considerably different than the "traditional" three-party model that has been studied in the field of authenticated data structures.) In our model the three parties are called *crawler*, *server*, and *client*. The client trusts the crawler but not the server. The protocol executed by the parties consists of three phases, preprocessing phase, query phase and update phase.

In the *preprocessing phase*, the crawler accesses the collection of documents and takes a snapshot of them. The crawler then produces and digitally signs a secure, collision-resistant digest of this snapshot. This digest contains information about both the identifiers of the documents in the snapshot and their contents. The signed digest is made public so that clients can use it for verification during the query phase. Concurrently, the crawler builds an authenticated data structure supporting keyword searches on the documents in the snapshot. Finally, the crawler outsources to the server the snapshot of the collection along with the authenticated data structure.

In the *query phase*, the client sends a query request, consisting of keywords, to the server. The server computes the query results, i.e., the set of documents in the snapshot that contain the query keywords. Next, the server returns to the

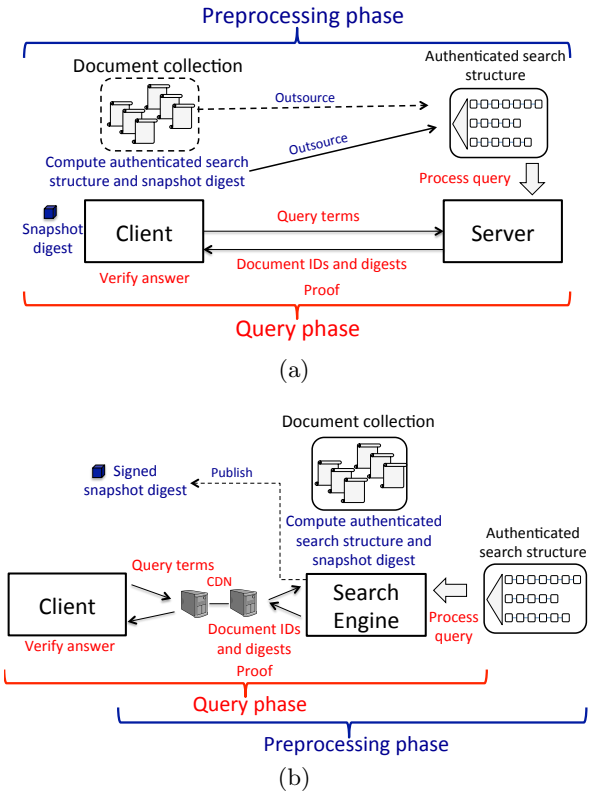

Figure 2: Two-party models. (a) A client outsources a document collection to a server who performs search queries on it. (b) Search engine protects its results from the man-in-the-middle attack.

client an answer consisting of the computed query results and a cryptographic proof that the query results are correct, complete, fresh[1] and that their associated digests are valid. The client verifies the correctness and the completeness of the query results using the proof provided by the server and relying on the trust in the snapshot digest that was signed and published by the crawler. The above interaction can be repeated for another query issued by the client.

In the *update phase*, the crawler parses the documents to be added or removed from the collection and sends to the server any new contents and changes for the authenticated data structure. Additionally, it computes and publishes a signed digest of the new snapshot of the collection.

Note that the query phase is executed after the preprocessing phase. Thus, the system gives assurance to the client that the documents have not been changed since the last snapshot was taken by the crawler. Also, during the update phase, the newly signed digest prevents the server from tampering with the changes it receives from the crawler.

2.2 Two-party Models

Our solution can also be used in a two-party model in two scenarios. In Figure 2(a), we consider a cloud-based model where a client outsources its data along with an authenticated search structure to the cloud-based server but keeps a digest of the snapshot of her data. In this scenario, the server executes keyword-search queries issued by the client and constructs a proof of the result. The client can then verify the results using the digest kept before outsourcing. The model shown in Figure 2(b) protects the interaction between a client and a search engine from a man-in-the-middle attack. The search engine publishes a signed digest of its current snapshot of the web and supplies every search result with a proof. Upon receiving search results in response to a query, the client can use the digest and the proof to verify that the results sent from the search engine were not tampered with.

[1]We note that in the cryptographic literature on authenticated data structures (e.g., [21, 22]) these three integrity properties are combined into a unified *security* property.

2.3 Desired Properties

The main properties we seek to achieve in our solution are *security* and *efficiency*. The system should be secure, i.e., it should be computationally infeasible for the server to construct a verifiable proof for an incorrect result (e.g., include a web page that does not contain a keyword of the query). Our system should also be practical, i.e., we should avoid a solution where the authenticated crawler literally signs every input participating in the specific computation and where the verification is performed by downloading and verifying all these inputs. For example, consider a search for documents which contain two specific terms. Each term could appear in many documents while only a small subset of them may contain both. In this case, we would not want a user to know about all the documents where these terms appear in order to verify that the small intersection she received as a query answer is correct. Finally, we want our solution to support efficient updates: Due to their high frequency, updates on web contents should incur overhead that is proportional to the size of the updated content and not of the entire collection.

We finally envision two modes of operations for our authentication framework. First, the authenticated web crawler operates autonomously; here, it serves as a "web police officer" and creates a global snapshot of a web site. This mode allows the integrity checking to be used as a verification that certain access control or compliance policies are met (e.g., a page owner does not post confidential information outside its organizational domain). Alternatively, the authenticated web crawler operates in coordination with the web page owners (authors); here, each owner interacts with the crawler to commit to the current local snapshot of its web page. This mode allows the integrity proofs to be transferable to third parties in case of a dispute (e.g., no one can accuse a page owner for deliberately posting offending materials).

3. OUR SOLUTION

In this section we describe the cryptographic and algorithmic methods used by all the separate entities in our model for the verification of conjunctive keyword searches. As we will see, conjunctive keyword searches are equivalent with a set intersection on the underlying *inverted index data structure* [27]. Our solution is based on the authenticated data structure of [22] for verifying set operations on outsourced sets. We now provide an overview of the construction in [22]. First we introduce the necessary cryptographic primitives.

3.1 Cryptographic Background

Our construction makes use of the cryptographic primitives of *bilinear pairings* and *Merkle hash trees*.

Bilinear pairings. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p, generated by g. Let also $\mathcal{G}$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G} \times \mathbb{G} \to \mathcal{G}$ be a bilinear pairing with the following properties: (1) Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$; (2) Non-degeneracy: $e(g, g) \neq 1$; (3) Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P, Q \in \mathbb{G}$. We denote with $(p, \mathbb{G}, \mathcal{G}, e, g)$ the bilinear pairings parameters, output by a polynomial-time algorithm on input 1^k.

Merkle hash trees. A Merkle hash tree [13] is an authenticated data structure allowing an untrusted server to vouch for the integrity of a dynamic set of indexed data $\mathcal{T}[0], \mathcal{T}[1], \ldots, \mathcal{T}[m-1]$ that is stored at untrusted repositories. Its representation comprises a binary tree with hashes in the internal nodes, denoted with $\mathsf{merkle}(\mathcal{T})$. A Merkle hash tree is equipped with the following algorithms:

1. $\{\mathsf{merkle}(\mathcal{T}), \mathsf{sig}(\mathcal{T})\} = \mathsf{setup}(\mathcal{T})$. This algorithm outputs a succinct signature of the table $\mathcal{T}$ which can be used for verification purposes and the hash tree $\mathsf{merkle}(\mathcal{T})$. To construct the signature and the Merkle tree, a collision-resistant hash function $\mathsf{hash}(\cdot)$ is applied recursively over the nodes of a binary tree on top of $\mathcal{T}$. Leaf $\ell \in \{0, 1, \ldots, m-1\}$ of $\mathsf{merkle}(\mathcal{T})$ is assigned the value $h_\ell = \mathsf{hash}(\ell || \mathcal{T}[\ell])$, while each internal node v with children a and b is assigned the value $h_v = \mathsf{hash}(h_a || h_b)$. The root of the tree h_r is signed to produce signature $\mathsf{sig}(\mathcal{T})$.

2. $\{\mathsf{proof}(i), \mathsf{answer}(i)\} = \mathsf{query}(i, \mathcal{T}, \mathsf{merkle}(\mathcal{T}))$. Given an index $0 \leq i \leq m-1$, this algorithm outputs a proof that could be used to prove that $\mathsf{answer}(i)$ is the value stored at $\mathcal{T}[i]$. Let $\mathsf{path}(i)$ be a list of nodes that denotes the path from leaf i to the root and $\mathsf{sibl}(v)$ denote a sibling of node v in $\mathsf{merkle}(\mathcal{T})$. Then, $\mathsf{proof}(i)$ is the ordered list containing the hashes of the siblings $\mathsf{sib}(v)$ of the nodes v in $\mathsf{path}(i)$.

3. $\{0, 1\} = \mathsf{verify}(\mathsf{proof}(i), \mathsf{answer}(i), \mathsf{sig}(\mathcal{T}))$. This algorithm is used for verification of the answer $\mathsf{answer}(i)$. It computes the root value of $\mathsf{merkle}(\mathcal{T})$ using $\mathsf{answer}(i)$ and $\mathsf{proof}(i)$, i.e., the sibling nodes of nodes in $\mathsf{path}(i)$, by performing a chain of hash computations over nodes in $\mathsf{path}(i)$. It then checks to see if the output is equal to h_r, a value signed with $\mathsf{sig}(\mathcal{T})$, in which case it outputs 1, implying that $\mathsf{answer}(i) = \mathcal{T}[i]$ whp.

We now describe in detail the individual functionality of the *web crawler*, the *untrusted server* and the *client*.

3.2 Web Crawler

The web page collection we are interested in verifying is described with an inverted index data structure. Suppose there are m terms $q_0, q_2, \ldots, q_{m-1}$ over which we are indexing, each one mapped to a set of web pages S_i, such that each web page in S_i contains the term q_i, for $i = 0, 1, \ldots, m-1$. Assume, without loss of generality, that each web page in S_i can be represented with an integer in $\mathbb{Z}_p^*$, where p is large k-bit prime. For example, this integer can be is a cryptographic hash of the entire web page. However, if we want to cover only a subset of the data in the page, the hash could be applied to cover text and outgoing links, but not the HTML structure. The extraction of the relevant data will be made by a special filter and will depend on the application. For instance, if we are interested in tables of NYSE financial data, there is no need to include images in the hash.

The authenticated data structure is built very simply as follows: First the authenticated crawler picks *random value* $s \in \mathbb{Z}_p^*$ which is kept secret. Then, for each set S_i ($i = 0, 1, \ldots, m-1$), the *accumulation value*,

$$\mathcal{T}[i] = g^{\prod_{x \in S_i} (s+x)},$$

is computed, where g is a generator of the group $\mathbb{G}$ from an instance of bilinear pairing parameters. Then the crawler calls algorithm $\mathsf{setup}(\mathcal{T})$ to compute signature $\mathsf{sig}(\mathcal{T})$ and Merkle hash tree $\mathsf{merkle}(\mathcal{T})$; the former is passed to the clients (to support the result verification) and the latter is outsourced to the server (to support the proof computation).

Intuition. There are *two* integrity-protection levels: First, the Merkle hash tree protects the integrity of the accumulation values $\mathcal{T}[i]$ offering coarse-grained verification of the integrity of the sets. Second, the accumulation value $\mathcal{T}[i]$ of S_i protects the integrity of the web pages in set S_i offering fine-grained verification of the sets. In particular, each accumulation value $\mathcal{T}[i]$ maintains an algebraic structure within the set S_i that is useful in two ways [21, 22]: (1) subject to an authentic accumulation value $\mathcal{T}[i]$ (subset) membership can be proved using a succinct witness, yielding a proof of "correctness," and (2) disjointness between t such sets can be proved using t succinct witnesses, yielding a proof of "completeness." Bilinearity is crucial for testing both properties.

Handling updates. During an update to inverted list S_i of term q_i the crawler needs to change the corresponding accumulation value $\mathcal{T}[i]$ and update the path of the Merkle tree from the leaf corresponding to term q_i to the root. A new page x' is added to an inverted list S_i of term q_i if either a new page x' contains term q_i or the content of page x' that is already in the corpus is updated and now contains q_i. In this case the accumulation value of q_i is changed to $\mathcal{T}'[i] = \mathcal{T}[i]^{(s+x')}$. If some web page $x' \in S_i$ is removed or no longer contains term x' the accumulation value is changed to $\mathcal{T}'[i] = \mathcal{T}[i]^{1/(s+x')}$. It is straightforward to handle updates in the Merkle hash tree (in time logarithmic in the number of inverted lists—see also Section 4).

3.3 Untrusted Server

The untrusted server in our solution stores the inverted index along with the authentication information defined earlier as authenticated data structure $\mathsf{merkle}(\mathcal{T})$. Given a conjunctive keyword-search query $q = (q_1, q_2, \ldots, q_t)$ from a client, the server returns a set of web pages $\mathcal{I}$ where each web page $p \in \mathcal{I}$ contains all terms from q. Namely, it is the case that $\mathcal{I} = S_1 \cap S_2 \cap \ldots \cap S_t$. The server is now going to compute a proof so that a client can verify that all web pages included in $\mathcal{I}$ contain $q_1, q_2, \ldots, q_t$ and ensure that no web page from the collection that satisfies query q is omitted from $\mathcal{I}$. Namely the server needs to prove that $\mathcal{I}$ is the correct intersection $S_1 \cap S_2 \cap \ldots \cap S_t$.

One way to compute such a proof would be to have the server just send *all* the elements in $S_1, S_2, \ldots, S_t$ along with Merkle tree proofs for $\mathcal{T}[1], \mathcal{T}[2], \ldots, \mathcal{T}[t]$. The contents of these sets could be verified and the client could then compute the intersection locally. Subsequently the client could check to see if the returned intersection is correct or not.

The drawback of this solution is that it involves *linear communication and verification complexity*, which could be prohibitive, especially when the sets are large.

To address this problem, in CRYPTO 2011, Papamanthou, Tamassia and Triandopoulos [22] observed that it suffices to certify *succinct relations* related to the correctness of the intersection. These relations have size independent of the sizes of the sets involved in the computation of the intersection, yielding a very efficient protocol for checking the correctness of the intersection. Namely $\mathcal{I} = S_1 \cap S_2 \cap \ldots \cap S_t$ is the correct intersection if and only if:

1. $\mathcal{I} \subseteq S_1 \wedge \ldots \wedge \mathcal{I} \subseteq S_t$ (subset condition);
2. $(S_1 - \mathcal{I}) \cap \ldots \cap (S_t - \mathcal{I}) = \emptyset$ (completeness condition).

Accordingly, for every intersection $\mathcal{I} = \{y_1, y_2, \ldots, y_\delta\}$ the server constructs the proof that consists of four parts:

A Coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ of the polynomial $(s+y_1)(s+y_2)\ldots(s+y_\delta)$ associated with the intersection $\mathcal{I}$;

B Accumulation values $\mathcal{T}[j]$ associated with the sets S_j, along with their respective proofs $\mathsf{proof}(j)$, output from calling algorithm $\mathsf{query}(j, \mathcal{T}, \mathsf{merkle}(\mathcal{T}))$, for $j = 1, \ldots, t$;

C Subset witnesses $\mathcal{W}_{\mathcal{I},j} = g^{P_j(s)}$, for $j = 1, \ldots, t$, where

$$P_j(s) = \prod_{x \in S_j - \mathcal{I}} (x + s)\,;$$

D Completeness witnesses $\mathcal{F}_{\mathcal{I},j} = g^{q_j(s)}$ for $j = 1, \ldots, t$, such that $q_1(s)P_1(s) + q_2(s)P_2(s) + \ldots + q_t(s)P_t(s) = 1$, where $P_j(s)$ are the exponents of the subset witnesses.

Intuition. Part A comprises an encoding of the result (as a polynomial) that allows efficient verification of the two conditions. Part B comprises the proofs for the 1-level integrity protection based on Merkle hash trees. Part C comprises a subset-membership proof for the 2-level integrity protection based on the bilinear accumulators. Part D comprises a set-disjointness proof for the 2-level integrity protection based on the extended Euclidean algorithm for finding the interrelation of irreducible polynomials.

3.4 Client

The client verifies the intersection $\mathcal{I}$ by appropriately verifying the corresponding proof elements described above:

A It first certifies that coefficients $b_\delta, b_{\delta-1}, ..., b_0$ are computed correctly by the server, i.e., that they correspond to the polynomial $\prod_{x \in \mathcal{I}}(s+x)$, by checking that $\sum_{i=0}^{\delta} b_i \kappa^i$ equals $\prod_{x \in \mathcal{I}}(\kappa + x)$ for a randomly chosen value $\kappa \in \mathbb{Z}_p^*$;

B It then verifies $\mathcal{T}[j]$ for each term q_j that belongs to the query $(j = 1, \ldots, t)$, by using algorithm $\mathsf{verify}(\mathsf{proof}(j), \mathcal{T}[j], \mathsf{sig}(\mathcal{T}))$;

C It then checks the subset condition

$$e\left(\prod_{k=0}^{\delta} (g^{s^k})^{b_k}, \mathcal{W}_{\mathcal{I},j}\right) = e\left(\mathcal{T}[j], g\right) \text{ for } j = 1, \ldots, t\,;$$

D Finally, it checks that the completeness condition holds

$$\prod_{j=1}^{t} e\left(\mathcal{W}_{\mathcal{I},j}, \mathcal{F}_{\mathcal{I},j}\right) = e(g, g)\,. \qquad (1)$$

The client accepts the intersection as correct if and only if all the above checks succeed.

Intuition. Step A corresponds to an efficient, high-assurance probabilistic check of the consistency of the result's encoding. Steps C and D verify the correctness of the subset and set-disjointness proofs based on the bilinearity of the underlying group and cryptographic hardness assumptions that are related to discrete-log problems.

3.5 Final Protocol

We now summarize the protocol of our solution.

Web crawler. Given a security parameter:

1. Process a collection of webpages and create an inverted index.
2. Generate a description of the group and bilinear pairing parameters $(p, \mathbb{G}, \mathcal{G}, e, g)$.
3. Pick randomly a secret key $s \in \mathbb{Z}_p^*$.
4. Compute accumulation value $\mathcal{T}[i]$ for each term i in the inverted index.
5. Build a Merkle hash tree $\mathsf{merkle}(\mathcal{T})$ and sign the root of the tree h_r as $\mathsf{sig}(\mathcal{T})$.
6. Compute values $g^{s^1}, \ldots, g^{s^n}$, where $n \geq \max\{m, \max_{i=0,\ldots,m-1}\{|S_i|\}\}$.
7. Send inverted index and $\mathsf{merkle}(\mathcal{T})$ to the server.
8. Publish $\mathsf{sig}(\mathcal{T})$, $(p, \mathbb{G}, \mathcal{G}, e, g)$ and $g^{s^1}, g^{s^2}, \ldots, g^{s^n}$ so that the server can access them to compute the proof and clients can acquire them during verification.

Untrusted server. Given a query $q = \{q_1, q_2, \ldots, q_t\}$:

1. Compute the answer for q as the intersection $\mathcal{I} = \{y_1, y_2, \ldots, y_\delta\}$ of inverted lists corresponding to q.
2. Compute the coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ corresponding to the polynomial $(s + y_1)(s + y_2), \ldots, (s + y_\delta)$.
3. Use $\mathsf{merkle}(\mathcal{T})$ to compute the integrity proofs of $\mathcal{T}[j]$.
4. Compute subset witnesses $\mathcal{W}_{\mathcal{I},j} = g^{P_j(s)}$.
5. Compute completeness witnesses $\mathcal{F}_{\mathcal{I},j} = g^{q_j(s)}$.
6. Send $\mathcal{I}$ and all components of the proof to the client.

Client. Send query q to the server, and given an answer to the query and the proof, accept the answer as correct if all of the following hold:

1. Coefficients of the intersection are computed correctly: Pick a random $\kappa \in \mathbb{Z}_p^*$ and verify that $\sum_{k=0}^{\delta} b_i \kappa^i = \prod_{x \in \mathcal{I}}(\kappa + x)$. (Note that the client can verify the coefficients without knowing the secret key s.)
2. Accumulation values are correct: Verify integrity of these values using $\mathsf{sig}(\mathcal{T})$ and $\mathsf{merkle}(\mathcal{T})$.
3. Subset and completeness conditions hold.

Term ID	Term	Inverted list
0	computer	6,8,9
1	disk	1,2,4,5,6,7
2	hard	1,3,5,7,8,9
3	memory	1,4,7
4	mouse	2,5
5	port	3,5,9
6	ram	5,6,7
7	system	1,7

Table 1: An example of an inverted index.

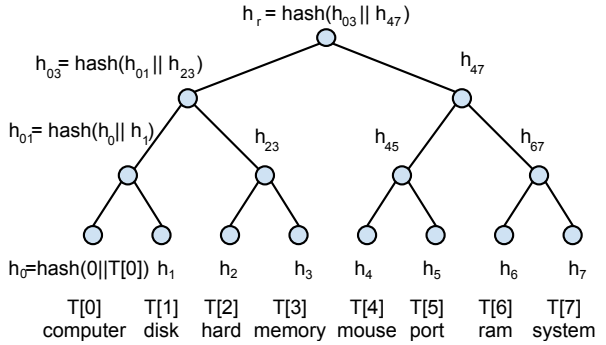

Figure 3: Merkle tree for authenticating the accumulation values of the terms in Table 1.

Example: We now consider how our protocol works on a toy collection. Consider an inverted index in Table 1 where a term is mapped to a set of documents where it appears, i.e., an inverted list. For example, term "mouse" appears in documents 2 and 5 and document 2 contains words "disk" and "mouse". The crawler computes accumulation values $\mathcal{T}[i]$ for each term id i, e.g., an accumulation value for term "memory" is $\mathcal{T}[3] = g^{(s+1)(s+4)(s+7)}$. It then builds a Merkle tree where each leaf corresponds to a term and its accumulation value, as shown in Figure 3. The Merkle tree and the inverted index are sent to the server. The crawler also computes $g^{s^1}, g^{s^2}, \ldots, g^{s^8}$ and publishes them along with a signed root of the Merkle tree, $\mathsf{sig}(\mathcal{T})$.

Given a query q = (*hard* AND *disk* AND *memory*), the result is the intersection $\mathcal{I}$ of the inverted lists for each of the terms in the query. In our case, $\mathcal{I} = \{1, 7\}$. The server builds a proof and sends it to the client along with the intersection. The proof consists of the following parts:

A Intersection proof: Coefficients $b_0 = 7$, $b_1 = 8$ and $b_2 = 1$ of the intersection polynomial $(s + 1)(s + 7)$.

B Accumulation values $\mathcal{T}[0]$, $\mathcal{T}[1]$ and $\mathcal{T}[2]$ with a corresponding proof from the Merkle tree that these values are correct: $(\mathcal{T}[1], \{h_0, h_{23}, h_{47}\})$, $(\mathcal{T}[2], \{h_3, h_{01}, h_{47}\})$, $(\mathcal{T}[3], \{h_2, h_{01}, h_{47}\})$.

C Subset witnesses $g^{P_1(s)}$, $g^{P_2(s)}$ and $g^{P_3(s)}$ for each term in the query where

$$\begin{aligned} P_1(s) &= (s+2)(s+4)(s+5)(s+6)\,, \\ P_2(s) &= (s+3)(s+5)(s+8)(s+9)\,, \\ P_3(s) &= (s+4)\,. \end{aligned}$$

D Completeness witnesses: Using the Extended Euclidean algorithm the server finds values $g^{q_1(s)}$, $g^{q_2(s)}$ and $g^{q_3(s)}$ such that $q_1(s)P_1(s) + q_2(s)P_2(s) + q_3(s)P_3(s) = 1$.

Note that since the server knows values g^{s^i}, it can compute values $g^{P_j(s)}$ and $g^{q_j(s)}$ without knowing a private key s (only the coefficients of the polynomials are required).

To verify the response from the server, the client:

A Picks random $\kappa \in \mathbb{Z}_p^*$ and checks that $\sum_{k=0}^{\delta} b_i \kappa^i = \prod_{x \in \mathcal{I}}(\kappa + x)$, e.g., $7 + 8\kappa + \kappa^2 = (\kappa + 1)(\kappa + 7)$.

B Verifies that each $\mathcal{T}[i]$ is correct, e.g., for $\mathcal{T}[1]$ and its proof $\{h_0, h_{23}, h_{47}\}$ the client checks that

$$h_r \stackrel{?}{=} \mathsf{hash}(\mathsf{hash}(\mathsf{hash}(h_0, \mathsf{hash}(1||\mathcal{T}[1])), h_{23}), h_{47}),$$

such that $\mathsf{sig}(\mathcal{T})$ is a signed root of the Merkle tree h_r.

C Checks subset condition:

$$e\left(\prod_{k=0}^{2}(g^{s^k})^{b_k}, g^{P_j(s)}\right) = e\left(\mathcal{T}[j], g\right) \text{ for } j = 1, 2\,.$$

D Checks completeness: $\prod_{j=1}^{3} e\left(g^{P_j}(s), g^{q_j(s)}\right) = e(g, g)$.

3.6 Security

With respect to the security, we show that given an intersection query (i.e., a keyword search) referring to keywords $q_1, q_2, \ldots, q_t$, any computationally-bounded adversary cannot construct an incorrect answer $\mathcal{I}$ and a proof π that passes the verification test of Section 3.4, except with negligible probability. The proof is by contradiction. Suppose the adversary picks a set of indices $q = \{1, 2, \ldots, t\}$ (wlog), all between 1 and m and outputs a proof π and an *incorrect* answer $\mathcal{I} \neq \mathsf{I} = S_1 \cap S_2 \cap \ldots \cap S_t$. Suppose the answer $\alpha(q)$ contains d elements. The proof π contains (i) *Some* coefficients $b_0, b_1, \ldots, b_d$; (ii) *Some* accumulation values acc_j with *some* respective proofs Π_j, for $j = 1, \ldots, t$; (iii) *Some* subset witnesses W_j with *some* completeness witnesses F_j, for $j = 1, \ldots, t$ (inputs to the verification algorithm).

Suppose the verification test on these values is successful. Then: (a) By the certification procedure, $\beta_0, \beta_1, \ldots, \beta_d$ are *indeed* the coefficients of the polynomial $\prod_{x \in \mathcal{I}}(x + s)$, except with negligible probability; (b) By the properties of the Merkle tree, values acc_j are *indeed* the accumulation values of sets S_j, except with negligible probability; (c) By the successful checking of the subset condition, values W_j are *indeed* the subset witnesses for set $\mathcal{I}$ (with reference to S_j), i.e., $\mathsf{W}_j = g^{P_j(s)}$, except with negligible probability; (d) However, since $\mathcal{I}$ is incorrect then it cannot include *all* the elements and there must be at least one element a that is not in $\mathcal{I}$ and is a *common factor* of polynomials $P_1(s), P_2(s), \ldots, P_t(s)$. In this case, the adversary can divide the polynomials $P_1(s), P_2(s), \ldots, P_t(s)$ with $s+a$ in the completeness relation of Equation 1 and derive the quantity $e(g, g)^{1/(s+a)}$ at the right hand side. However, this implies that the adversary has solved in polynomial time a difficult problem in the target group $\mathcal{G}$, in particular, the adversary has broken the bilinear q-strong Diffie-Hellman assumption, which happens with negligible probability. More details on the security proof can be found in [22].

4. PERFORMANCE

In this section, we describe a prototype implementation of our authenticated crawler system and discuss the results of experimentation regarding its practical performance.

4.1 Performance Measures

We are interested in studying four performance measures:

1. The size of the proof of a query result, which is sent by the server to the client. Asymptotically, the proof size is $O(\delta + t \log m)$ when δ documents are returned while searching for t terms out of the m total distinct terms that are indexed in the collection. This parameter affects the bandwidth usage of the system.

2. The computational effort at the server for constructing the proof. Let N be the total size of the inverted lists of the query terms. The asymptotic running time at the server is $O(N \log^2 N \log\log N)$ [22]. Note that the overhead over the time needed to compute a plain set intersection, which is $O(N)$, is only a polylogarithmic multiplicative. In practice, the critical computation at the server is the extended Euclidean algorithm, which is executed to construct the completeness witnesses.

3. The computational effort at the client to verify the proof. The asymptotic running time at the client is $O(\delta + t \log m)$ for t query terms, δ documents in the query result and an inverted index of m distinct terms.

4. The computational effort at the crawler to update the authenticated data structure when some documents are added to or deleted from the collection. The asymptotic running time at the crawler consists of updating accumulation values and corresponding Merkle tree paths for t' unique terms that appear in n' updated documents, and, hence, it is $O(t'n' + t' \log m)$.

4.2 Prototype Implementation

Our prototype is built in C++ and is split between three parties: authenticated crawler, search engine and client. The interaction between the three proceeds as follows.

The crawler picks a secret key s and processes a collection of documents. After creating the inverted index, the crawler computes an accumulation value for each inverted list. We use cryptographic pairing from [14] for all bilinear pairing computations which are available in DCLXVI library.[2] This library implements an optimal ate pairing on a Barreto-Naehrig curve over a prime field $\mathbb{F}_p$ of size 256 bits, which provides 128-bit security level. Hence, the accumulation value of the documents containing a given term is represented as a point on a curve. Once all accumulation values are computed, the crawler builds a Merkle tree where each leaf corresponds to a term and its accumulation value. We use SHA256 from the OpenSSL library[3] to compute a hash value at each node of the Merkle tree. The crawler also computes values g, g^s, ..., g^{s^n}.

After the authenticated data structure is built, the crawler outsources the inverted index, authentication layer and precomputed values g, g^s, ...,g^{s^n} to the server.

Clients query the search engine via the RPC interface provided by the Apache Thrift software framework. For each query, the server computes the proof consisting of four parts. To efficiently compute the proof, the server makes use of the following algorithms. The coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ and the coefficients for the subset witnesses are computed using FFT. The Extended Euclidean algorithm is used to compute the coefficients $q_1(s)$, $q_2(s), \ldots, q_t(s)$ for the completeness witnesses. We use the NTL[4] and LiDIA[5] libraries for efficient arithmetic operations, FFT and the Euclidean algorithm on elements in $\mathbb{Z}_p^*$, which represent document identifiers. Bilinear pairing, power and multiplication operations on the group elements are performed with the methods provided in the bilinear map library.

We performed the following optimizations on the server. The computation of subset and completeness witnesses is independent of each other and hence is executed in parallel. We also noticed that the most expensive part of the server's computation is the power operation for group elements when computing subset and completeness witnesses. Since the order of these computations is independent from each other we run the computation of $g^{P_j(s)}$ and $g^{q_j(s)}$ in parallel.

The client's verification algorithm consists of verifying the accumulation values using the Merkle tree and running the bilinear pairing procedure over the proof elements.

4.3 Experimental Results

We have conducted computational experiments on a 8-core Xeon 2.93 processor with 8Gb RAM running 64-bit Debian Linux. In the discussion of the results, we use the following terminology.

Total set size: sum of the lengths of the inverted lists of the query terms. This value corresponds to variable N used in the asymptotic running times given in Section 4.1.

Intersection size: number of documents returned as the query result. This value corresponds to variable δ used in the asymptotic running times given in Section 4.1.

4.3.1 Synthetic Data

We have created a synthetic data set where we can control the frequency of the terms as well as the size of the intersection for each query. The synthetic data set contains 777,348 documents and 320 terms. Our first experiment identifies how the size of the intersection, δ, influences the time it takes for the server to compute the proof given that the size of the inverted lists stays the same. We report the results in Figure 4.2 where each point corresponds to a query consisting of two terms. Each term used in a query appears in 2,000 documents. As the intersection size grows, the size of the polynomial represented by the subset witness $P_j(s)$ in Section 3.3 decreases. Hence the time it takes the server to compute the proof decreases as well.

We now measure how the size of the inverted lists affects server's time when the size of the resulting intersection is fixed to $\delta = 100$ documents. Figure 4.2 shows results for queries of two types. The first type consists of queries where both terms appear in the collection with the same frequency. Queries of the second type contain a frequent and a rare term. In each query we define a term as rare if its frequency is ten times less than the frequency of the other term. As the number of terms for subset witnesses grows the time to compute these witnesses grows as well. The dependency is linear, as expected (see Section 4.1). We also note that

[2] `http://www.cryptojedi.org/crypto/`

[3] `http://www.openssl.org/`

[4] A Library for doing Number Theory, V5.5.2.

[5] A library for Computational Number Theory, V2.3.

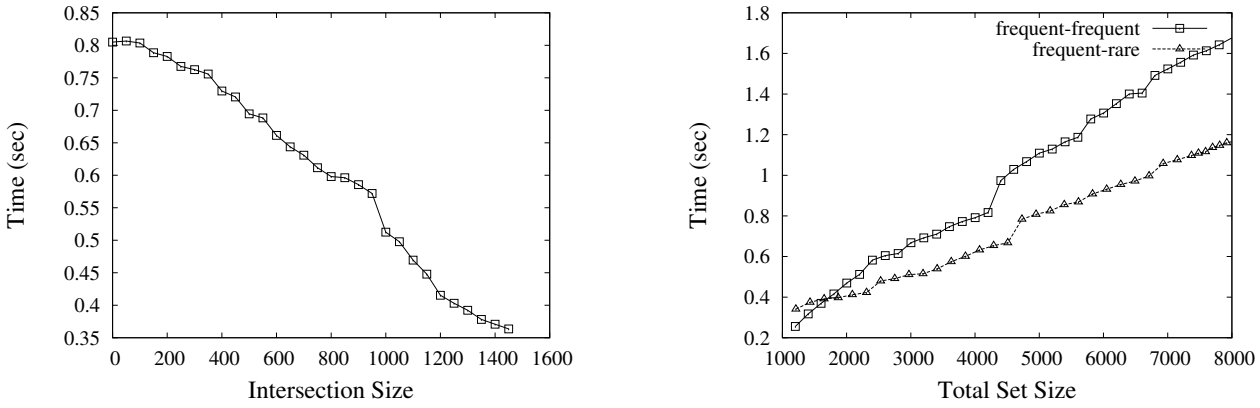

Figure 4: Computational effort at the server on synthetic data. (Left) Proof computation time for a 2-term query as a function of the intersection size (number of returned results) when each query term appears in 2,000 documents. (Right) Proof computation time as a function of the total set size (number of documents in queried inverted lists) for frequent-frequent and frequent-rare 2-term queries each returning 100 documents.

the computation is more expensive when both terms in the query have the same frequency in the collection.

4.3.2 WSJ Corpus

We have also tested our solution on a real data set that consists of 173,252 articles published in the Wall Street Journal from March 1987 to March 1992. After preprocessing, this collection has 135,760 unique terms. We have removed common words, e.g., articles and prepositions, as well as words that appear only in one document. The lengths of the inverted lists follow a power law distribution where 56% of the terms appear in at most 10 documents.

Our query set consists of 200 random conjunctive queries with two terms. We picked queries that yield varying result sizes, from empty to 762 documents. Since each term in a query corresponds to an inverted list of the documents it appears in, we also picked rare as well as frequent terms. Here, we considered a term frequent if it appeared in at least 8,000 documents. The total set size and corresponding intersection size for each query is shown in Figure 5.

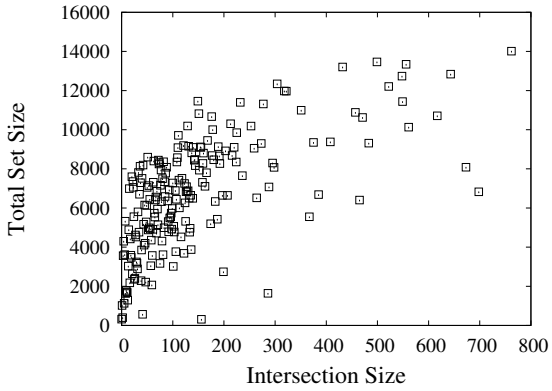

Figure 5: Query set for WSJ corpus. Each point relates the number of returned documents to the number of documents in the queried inverted lists.

Server time. We first measure the time it takes for the server to compute the proof. In Figure 6 (left) we show how the size of the inverted lists of the terms in the query influences server's time. As expected, the dependency is linear in the total set size. However, some of the queries that have inverted lists of close length result in different times. This happens because the intersection size varies between the queries, as can be seen in Figure 5. Furthermore, some of the queries contain different type of terms, e.g., consider a query with one rare and one frequent word, and a query with two semi-frequent words (see Section 4.3.1).

In Figure 6 (right) we show how the intersection size influences server's time. Note that the graph is almost identical to Figure 5, again showing that the time mostly depends on the lengths of the inverted lists and not the intersection size.

Client time and proof size. We now measure the time it takes for the client to verify the proof. Following the complexity analysis of our solution, the computation on the client side is very fast. We split the computation time since verification of the proof consists of verifying that intersection was performed on correct inverted lists (Merkle tree) and that intersection itself is computed correctly (bilinear pairing on accumulation values). In Figure 7, we plot the time it takes to verify both versus the intersection size: It depends only on the intersection size and not on the total set size (the lengths of the inverted lists of the query terms). Finally, the size of the proof sent to the client is proportional to the intersection size as can be seen in Figure 8.

Updates to the corpus. The simulation supports addition and deletion of new documents and updates corresponding authenticated data structures. We pick a set of 1500 documents from the original collection which covers over 14% of the collection vocabulary. In Figure 9 we measure the time it takes for the crawler to update accumulation values in the authenticated data structure. As expected the time to do the update is linear in the number of unique terms that appear in the updated document set. Deletions and additions take almost the same time since the update is dominated by a single exponentiation of the accumulation value of each affected term. Updates to Merkle tree take milliseconds.

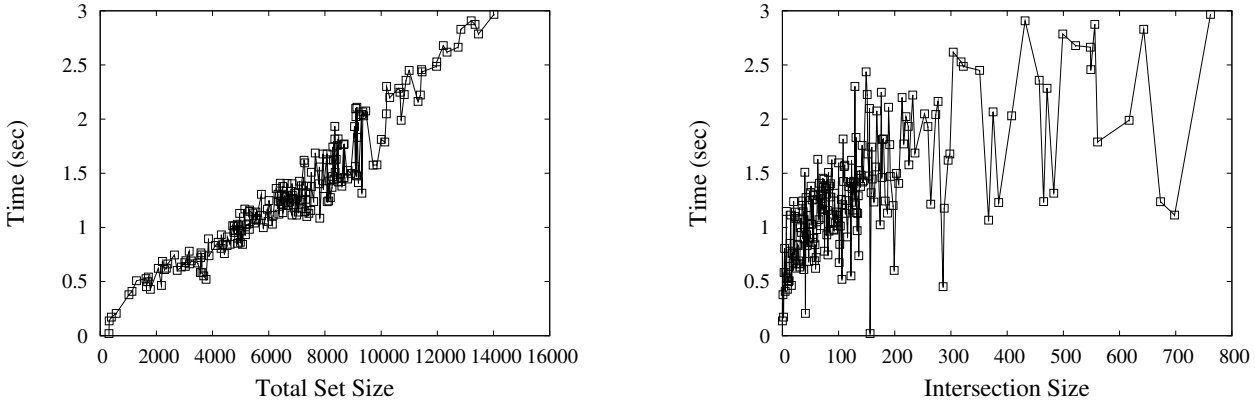

Figure 6: Computational effort at the server for queries with two terms on the WSJ corpus. Time to compute the proof as a function of the total set size (left) and the intersection size (right).

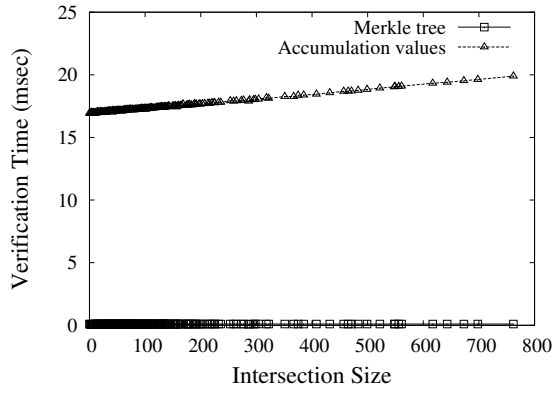

Figure 7: Verification time at the client as a function of the intersection size split into its two components.

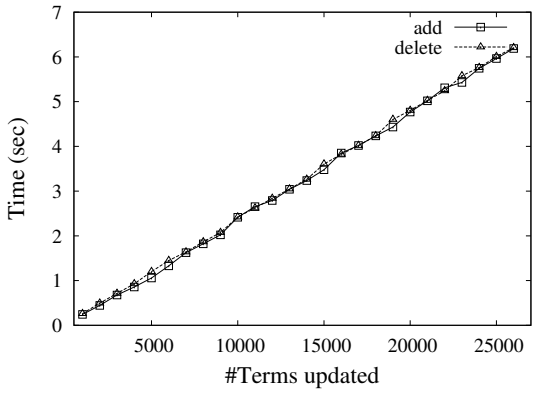

Figure 9: Update time at the crawler as a function of terms contained in the updated documents.

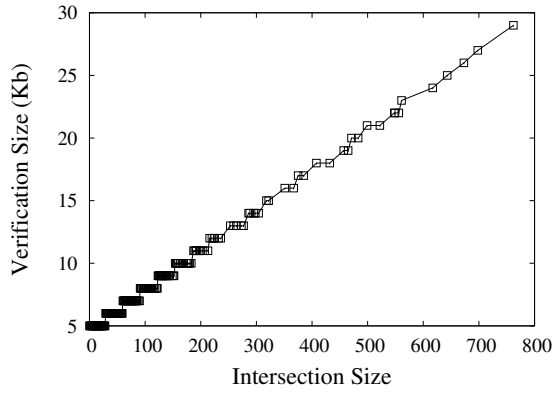

Figure 8: Proof size as function of intersection size.

4.3.3 Comparison with Previous Work

The closest work to ours is the method developed by Pang and Mouratidis [20]. However, their method solves a different problem. Using our method, the user can verify whether the result of the query contains all the documents from the collection that satisfy the query and no malicious documents have been added. The method of [20] proves that the returned result consists of top-ranked documents for the given query. However, it does not assure the completeness of the query result with respect to the query terms.

The authors of [20] show that their best technique achieves below 60 msec verification time,[6] and less than 50 Kbytes in proof size for a result consisting of 80 documents. Using our method the verification time for a result of 80 documents takes under 17.5 msec (Figure 7) and the corresponding verification object is of size under 7 Kbytes (Figure 8). The computation effort by the server reported in [20] is lower than it is for our method, one second versus two seconds.

We also note that updates for the solution in [20] require changes to the whole construction, while updates to our authenticated data structures are linear in the number of unique terms that appear in new documents.

4.3.4 Improvements and Extensions

From our experimental results, we observe that the most expensive part of our solution is the computation of subset and completeness witnesses at the server. This is evident when a query involves frequent terms with long inverted

[6]Dual Intel Xeon 3GHz CPU with 512MB RAM machine.

lists, where each term requires a call to a multiplication and power operation of group elements in $\mathbb{G}$. However, these operations are independent of each other and can be executed in parallel. Our implementation already runs several multiplication operations in parallel. However, the number of parallel operations is limited on our 8-core processor.

In a practical deployment of our model, the server is in the cloud and has much more computational power than the client, e.g., the server is a search engine and the client is a portable device. Hence, with a more powerful server, we can achieve faster proof computation for frequent terms.

Our implementation could use a parallel implementation of the Extended Euclidean Algorithm, however, the NTL library is not thread-safe and therefore we could not perform this optimization for our current prototype.

5. CONCLUSION

We study the problem of verifying the results of a keyword-search query returned by a search engine. We introduce the concept of an authenticated web-crawler which enables clients to verify that the query results returned by the search engine contain all and only the web pages satisfying the query. Our prototype implementation has low communication overhead and provides fast verification of the results.

Our method verifies the correctness and completeness of the results but does not check the ranking of the results. An interesting extension to our method would be to efficiently verify also the ranking, i.e., return to the client r pages and a proof that the returned set consists of the top-r results.

6. ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under grants CNS-1012060, CNS-1012798, CNS-1012910, OCI-0724806 and CNS-0808617, by a NetApp Faculty Fellowship, and by Intel through the ISTC for Secure Computing. We thank Isabel Cruz, Kristin Lauter, James Lentini, and Michael Naehrig for useful discussions.

7. REFERENCES

[1] M. J. Atallah, Y. Cho, and A. Kundu. Efficient data authentication in an environment of untrusted third-party distributors. *ICDE*, 696–704, 2008.

[2] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *Trans. Knowl. Data Eng.*, 16(10): 1263-1278, 2004.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. *CCS*, 9–18, 2000.

[4] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. *CRYPTO*, 61–76, 2002.

[5] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. *CCS*, 136–145, 2001.

[6] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the Internet. *J. Comput. Security*, 11(3):291–314, 2003.

[7] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. *ISC*, 372–388, 2002.

[8] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. *DISCEX II*, 68–82, 2001.

[9] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Authenticated data structures for graph and geometric searching. *Algorithmica*, 60(3):505–552, 2010.

[10] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. *CT–RSA*, 407–424, 2008.

[11] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. *SIGMOD*, 121–132, 2006.

[12] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41, 2004.

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

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

[15] M. Naor and K. Nissim. Certificate revocation and certificate update. *USENIX Security*, 217–228, 1998.

[16] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. *DASFAA*, 420–436, 2006.

[17] L. Nguyen. Accumulators from bilinear pairings and applications. *CT-RSA*, 275–292, 2005.

[18] G. Nuckolls. Verified query results from hybrid authentication trees. *DBSec*, 84–98, 2005.

[19] H. Pang, A. Jain, K. Ramamritham, and K. L. Tan. Verifying completeness of relational query results in data publishing. *SIGMOD*, 407–418, 2005.

[20] H. Pang and K. Mouratidis. Authenticating the query results of text search engines. *PVLDB*, 1(1):126–137, 2008.

[21] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. *CCS*, 437–448. 2008.

[22] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. *CRYPTO*, 91–110, 2011.

[23] R. Tamassia and N. Triandopoulos. Certification and authentication of data structures. In *AMW*, 2010.

[24] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. *ICALP*, 153–165, 2005.

[25] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. *SIGMOD*, 5–18, 2009.

[26] M. L. Yiu, Y. Lin and K. Mouratidis. Efficient verification of shortest path search via authenticated hints. *ICDE*, 237–248, 2010.

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