Practical Authenticated Pattern Matching with Optimal Proof Size

Dimitrios Papadopoulos
Boston University
dipapado@cs.bu.edu

Charalampos Papamanthou
University of Maryland
cpap@umd.edu

Roberto Tamassia
Brown University
rt@cs.brown.edu

Nikos Triandopoulos
RSA Laboratories & Boston University
nikolaos.triandopoulos@rsa.com

ABSTRACT

We address the problem of authenticating pattern matching queries over textual data that is outsourced to an untrusted cloud server. By employing cryptographic accumulators in a novel optimal integrity-checking tool built directly over a suffix tree, we design the first authenticated data structure for verifiable answers to pattern matching queries featuring *fast generation of constant-size proofs*. We present two main applications of our new construction to authenticate: (i) pattern matching queries over text documents, and (ii) exact path queries over XML documents. Answers to queries are verified by proofs of size at most 500 bytes for text pattern matching, and at most 243 bytes for exact path XML search, independently of the document or answer size. By design, our authentication schemes can also be parallelized to offer extra efficiency during data outsourcing. We provide a detailed experimental evaluation of our schemes showing that for both applications the times required to compute and verify a proof are very small—e.g., it takes less than 10μs to generate a proof for a pattern (mis)match of 10^2 characters in a text of 10^6 characters, once the query has been evaluated.

1. INTRODUCTION

The advent of cloud computing has made data outsourcing common practice for companies and individuals that benefit from delegating storage and computation to powerful servers. In this setting, integrity protection is a core security goal. Ensuring that information remains intact in the lifetime of an outsourced data set and that query processing is handled correctly, producing correct and up-to-date answers, lies at the foundation of secure cloud services.

In this work we design protocols that cryptographically guarantee the correct processing of *pattern matching* queries. The problem setting involves an outsourced textual database, a query containing a text *pattern*, and an answer regarding the presence or absence of the pattern in the database. In its most basic form, the database consists of a single text $\mathcal{T}$ from an alphabet Σ, where a query for pattern p, expressed as a string of characters, results in answer "match at i", if p occurs in $\mathcal{T}$ at position i, or in "mismatch" otherwise. More elaborate models for pattern matching involve queries expressed as regular expressions over Σ or returning multiple occurrences of p, and databases allowing search over multiple texts or other (semi-)structured data (e.g., XML data). This core data-processing problem has numerous applications in a wide range of topics including intrusion detection, spam filtering, web search engines, molecular biology and natural language processing.

Previous works on authenticated pattern matching include the schemes by Martel et al. [28] for text pattern matching, and by Devanbu et al. [16] and Bertino et al. [10] for XML search. In essence, these works adopt the same general framework: First, by hierarchically applying a cryptographic hash function (e.g., SHA-2) over the underlying database, a short secure description or *digest* of the data is computed. Then, the answer to a query is related to this digest via a proof that provides step-by-step reconstruction and verification of the *entire answer computation*. This approach typically leads to large proofs and, in turn, high verification costs, proportional to the number of computational steps used to produce the answer. In fact, for XML search, this approach offers no guarantees for efficient verification, since certain problem instances require that the proof includes almost the entire XML document, or a very large part of it, in order to ensure that no portions of the true (honest) answer were omitted from the returned (possibly adversely altered) answer.

On the other hand, recent work on *verifiable computing* (e.g., [32, 13, 8]) allows verification of general classes of computation. Here also, verification is based on cryptographic step-by-step processing of the entire computation, expressed by circuits or RAM-based representations. Although special encoding techniques allow for constant-size proofs and low verification costs, this approach cannot yet provide practical solutions for pattern matching, as circuit-based schemes inherently require complex encodings of all database searches, and RAM-based schemes result in very high proof generation costs. Indeed, costly proof generation comprises the main bottleneck in all existing such implementations.

Our goal. We wish to design schemes that offer an efficient answer-verification process for authenticated pattern matching. That is, answer correctness is based on a proof that is *succinct*, having size independent of the database size and the query description, and that can be *quickly generated and verified*. We emphasize that our requirement to support pattern matching verification with easy-to-compute constant-size proofs is in practice a highly desired property. First, it contributes to *high scalability in query-intensive applications* in settings where the server that provides querying service for outsourced databases receives incoming requests by several clients at high rates; then obviously, faster proof generation and transmission of constant-size proofs result in faster response

Proceedings of the VLDB Endowment, Vol. 8, No. 7

	space	setup	proof size	query time	verification time	assumption
[23]	n	n	$m \log \Sigma$	$m \log \Sigma + \kappa$	$m \log \Sigma + \kappa$	collision resistance
this work	n	n	1	$m + \kappa$	$m \log m + \kappa$	ℓ-strong DH
[10, 16] (any path)	n	n	n, d	n, d	n, d	collision resistance
this work (exact path)	n	n	1	m	$m \log m + s$	ℓ-strong DH

Table 1: Asymptotic complexities of our scheme for text pattern matching and XML exact path queries: n is the size of the document, m the pattern length, κ the number of occurrences, Σ the alphabet size, s the answer size, and d the number of valid paths.

times and higher throughputs. But it also promotes *storage efficiency in data-intensive applications* in settings where the proof for a (mis)match of any pattern query over a database must be persistently retained in storage for a long or even unlimited time duration; then, minimal-size proofs result in the minimum possible storage requirements, a very useful feature in big-data environments.

An example of a data-intensive application where pattern matching proofs might be permanently stored, is the problem of *securing the chain of custody* in forensic and intrusion detection systems used by enterprises today. Such systems often apply big-data security analytics (e.g., [41]) over terabytes of log or network-traffic data (collected from host machines, firewalls, proxy servers, etc.) for analysis and detection of impending attacks. Since any data-analytics tool is only as useful as the quality (and integrity) of its data, recent works (e.g., [40, 12]) focus on the integrity of the data consumed by such tools, so that any produced security alert carries a cryptographic proof of correctness. To support a verifiable chain of custody,[1] these proofs must be retained for long periods of time. As big-data security analytics grow in sophistication, authenticated pattern matching queries will be crucial for effective analytics (e.g., to match collected log data against known high-risk signatures of attacks), hence storing only constant-size associated proofs will be important in the fast-moving area of information-based security.

Contributions. We present the first authentication schemes for pattern matching over textual databases that achieve the desired properties explained above. Our schemes employ a novel authenticated version of the suffix tree data structure [18] that can provide *precomputed (thus, fast to retrieve), constant-size* proofs for any basic-form pattern matching query, at no asymptotic increase of storage.

Our first contribution is the design and implementation of an authentication scheme for pattern matching such that:

- The size of the proof is $O(1)$; specifically, it always contains at most 10 bilinear group elements (described in Section 2).
- The time to generate the proof that a query pattern of size m is found in κ occurrences is $O(m+\kappa)$, and *very short* in practice, as it involves *no* cryptographic operations but only assembling of *precomputed* parts—e.g., it takes less than 90μs to respond to a query of size 100 characters: 80μs to simply find the (mis)match and less than 10μs to assemble the proof.

We extend our scheme to also support regular expressions with a constant number of wildcards. Our second contribution is the application of our main scheme above to the authentication of *pattern matching queries over collections of text documents* (returning the indices of documents with positive occurrences), and *exact path queries* over XML documents. By design, these schemes also achieve *optimal* communication overhead: On top of the requested answer, the server provides only a constant number of bits (modulo the security parameter)—e.g., for XML search and 128-bit security level, proofs can be made as small as ∼178 bytes. Unlike existing hash-based authentication schemes [10, 16, 23], our authentication schemes support fully parallelizable set-up: They can be constructed in $O(\log n)$ parallel time on $O(n/\log n)$ processors in the EREW model, thus maintaining the benefits of known parallel algorithms for (non-authenticated) suffix trees [19, 22]. While the use of precomputed proofs best matches static text databases, we also present efficient fully or semi-dynamic extensions of our schemes. Our last contribution is the implementation of our schemes for authenticated pattern matching search on text and XML documents along with an experimental evaluation of our verification techniques that validates their efficacy and practicality.

Construction overview. Our solution is designed in the model of *authenticated data structures* (ADS) [34], which has a prominent role in the literature due to its generality, expressive power and relevance to practice. An ADS is best described in the following *three-party* setting: A data set (the text in our case) that originates at a trusted *owner* is outsourced to an untrusted *server* which is responsible for answering queries coming from *clients*. Along with an answer to a query returned by the server, a client is provided with a *proof* that can be used to verify, in a cryptographic sense, the answer correctness, using a public key issued by the owner. That is, subject to some cryptographic hardness assumption, verifying the proof is a reliable attestation that the answer is correct.

We follow the framework of [35]: Our schemes first define and encode *answer-specific* relations that are sufficient for *certifying* (unconditionally) that an answer is correct and, then, cryptographically authenticate these relations using optimal-size proofs. We achieve this by employing in a novel way the bilinear-map (BM) accumulator [27],[2] over a special encoding of the database with respect to a suffix tree, used to find the pattern (mis)match. The encoding effectively takes advantage of the suffix tree where patterns in the database share common prefixes, which in turn can be succinctly represented by an accumulator. For the XML query application, we use the same approach, this time over a trie defined over all possible paths in the document, and we link each path with the respective XML query answer (i.e., all reachable XML elements).

Related work. Table 1 summarizes our work as compared to [10, 16, 23].[3] In [23] a general technique is applied to the suffix tree, that authenticates every step of the search algorithm, thus obtaining proof size proportional to the length of the pattern, which is not optimal. Moreover, due to the use of sequential hashing, this solution is inherently not parallelizable. The authors of [16] authenticate XPath queries over a simplified version of XML documents by relying on the existence of a document type definition (DTD) and applying cryptographic hashing over a trie of all possible semantically distinct path queries in the XML document. An alternative

[1] Informally, any security alert—carrying important forensic value—can be publicly and with non-repudiation verified—thus, carrying also legal value when brought as evidence to court months, or even years, after the fact.

[2] An *accumulator* [9] is a cryptographic primitive for securely proving set-inclusion relations *optimally* via $O(1)$-size proofs verifiable in $O(1)$ time.

[3] We note that our ADS schemes operate with any accumulator, not just the BM accumulator. In fact, using the RSA accumulator [14] reduces verification cost to $O(m)$. However, a recent experimental comparison demonstrates that the BM accumulator is more efficient in practice [37].

approach is taken in [10], where similar XML queries are authenticated by applying cryptographic hashing over the XML tree structure. As discussed above, both these approaches suffer from very bad worst-case performance, e.g., yielding verification proofs/costs that are proportional to the size of the XML tree. However, these works are designed to support *general* path queries, not only *exact*, as our works does. Recently, the authors of [17] presented a protocol for verifiable pattern matching that achieves security and secrecy in a very strong model, hiding the text even from the responding server. While that work offers security in a much more general model than ours, it has the downside that the owner that outsourced the text is actively involved in query responding and that it makes use of heavy cryptographic primitives, the practicality of which remains to determined. There is a large number of ADS schemes in the database and cryptography literature for various classes of queries (e.g., [24, 15, 21]). Also related to our problem is keyword search authentication, which has been achieved efficiently, e.g., in [29, 39, 30]. As previously discussed, verifiable computation systems such as [32, 13] can be used for the verification of pattern matching; although optimized to provide constant-size proofs these constructions remain far from practical. Finally, parallel algorithms in the context of verifiable computation have only recently been considered. In [36, 33] parallel algorithms are devised for constructing a proof for arithmetic-circuit computations.

2. CRYPTOGRAPHIC TOOLS

We review some known cryptographic primitives that we use as building blocks.

Bilinear-map (BM) accumulator [27]. A BM accumulator succinctly and securely represents a set of elements from $\mathbb{Z}_p$, operating in the following setting of bilinear groups. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p, generated by g. Let also $\mathbb{G}_T$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be a bilinear pairing (or mapping) such that: (1) $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$ (bilinearity); (2) $e(g, g) \neq 1$ (non-degeneracy); (3) $e(P, Q)$ is efficiently computable for all $P, Q \in \mathbb{G}$ (computability). Let also $(p, \mathbb{G}, \mathbb{G}_T, e, g)$ be a tuple of *public bilinear parameters* as above, selected uniformly at random. The BM accumulator represents any set $\mathcal{A}$ of n elements from $\mathbb{Z}_p$ by its *accumulation value*, namely

$$\mathsf{acc}(\mathcal{A}) = g^{\prod_{a \in \mathcal{A}}(s+a)} \in \mathbb{G} ,$$

i.e., a single element in $\mathbb{G}$, where $s \in \mathbb{Z}_p$ is trapdoor information that is kept secret. Note that given values $g, g^s, \ldots, g^{s^n}$ (and without revealing the trapdoor s), $\mathsf{acc}(\mathcal{A})$ can be computed in time $O(n \log n)$ with polynomial interpolation using FFT techniques.

The *ℓ-strong DH or, here, ℓ-sDH* assumption for bilinear groups is a variant of the well-known computational Diffie-Hellman assumption applied to bilinear groups. The ℓ-sDH assumption was introduced in [11] and has since been widely used in the cryptographic literature as a standard assumption. Under the ℓ-sDH assumption, the BM accumulator is shown to provide two security properties: (1) The accumulation function $\mathsf{acc}(\cdot)$ is *collision resistant* [31] (i.e., it is computationally hard to find unequal sets with equal accumulation values); and (2) It allows for *reliable verification of subset containment* [30] using short, *size optimal*, computational proofs; namely, subject to $\mathsf{acc}(\mathcal{A})$, the proof for relation $\mathcal{B} \subseteq \mathcal{A}$ is defined as the *subset witness* $\mathsf{W}_{\mathcal{B},\mathcal{A}} = g^{\prod_{a \in \mathcal{A}-\mathcal{B}}(s+a)}$ (i.e., $\mathcal{B} \subseteq \mathcal{A}$ can be efficiently validated via checking the equality $e(\mathsf{W}_{\mathcal{B},\mathcal{A}}, g^{\prod_{b \in \mathcal{B}}(s+b)}) \stackrel{?}{=} e(\mathsf{acc}(\mathcal{A}), g)$ given accumulation value $\mathsf{acc}(\mathcal{A})$, set $\mathcal{B}$ and public values $g, g^s, \ldots, g^{s^\ell}$, where ℓ is an upper bound on $\mathcal{A}$'s size n; but it is computationally hard to produce a fake subset witness that is verifiable when $\mathcal{B} \subseteq \mathcal{A}$ is false).

Authenticated data structure scheme. We use the model of *authenticated data structures* (ADS). This model assumes three parties: an *owner* holding a data structure D who wishes to outsource it to a *server* who is, in turn, responsible for answering queries issued by multiple *clients*. The owner preprocesses D, producing some cryptographic authentication information $auth(D)$ and a succinct digest d of D, and signs d. The server is untrusted, i.e., it may modify the returned answer, hence it is required to provide a proof of the answer, generated using $auth(D)$, and the signed digest d. A client with access to the public key of the owner can subsequently check the proof and verify the integrity of the answer.

DEFINITION 1 (ADS [30]). *An* ADS *scheme $\mathcal{A}$ is a collection of four algorithms as follows: (i) $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$ outputs secret and public keys* sk *and* pk*, given the security parameter k; (ii) $\{\mathsf{auth}(D), d\} \leftarrow \mathsf{setup}(D, \mathsf{sk}, \mathsf{pk})$ computes the authenticated structure $\mathsf{auth}(D)$ and its respective digest d, given a plain data structure D, the secret and public keys* sk*,* pk*; (iii) $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D, \mathsf{auth}(D), \mathsf{pk})$ returns an answer $\alpha(q)$, along with a proof $\Pi(q)$, given a query q on data structure D and the authenticated data structure $\mathsf{auth}(D)$; (iv) $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d, \mathsf{pk})$ outputs either* accept *or* reject*, on input a query q, an answer α, a proof Π, a digest d and public key* pk*.*

Like a signature scheme, an authenticated data structure scheme must satisfy correctness and security properties. Namely, (i) as long as the adversary does not tamper with the data and the algorithms of the scheme (therefore correct answers are returned), verification algorithms always *accept*; (ii) a computationally-bounded adversary cannot persuade a verifier for the validity of an incorrect answer.

Accumulation trees. The *accumulation tree* is an ADS scheme introduced in [31] that supports *set membership* queries. It can be viewed as an alternative to a Merkle hash tree [24]. The main differences are that proofs have constant size (instead of logarithmic in the set size), internal nodes have larger degrees (making the trees "flat"), and they rely on different cryptographic primitives (accumulators versus hash functions). During setup, a parameter $0 < \epsilon \leq 1$ is chosen that dictates the degree of inner nodes. E.g., an accumulation tree with $\epsilon = 1/2$ has 2 levels and each internal node has degree $O(\sqrt{n})$. Accumulation trees build upon BM accumulators, storing at each internal node the accumulation value of its children (appropriately encoded as elements of $\mathbb{Z}_p$). As shown in [31], an accumulation tree $\mathcal{AT}$ for a set of n elements can be built in time $O(n)$ and yields proofs of $O(1)$ group elements, verifiable with $O(1)$ group operations. If the tree has height 1, proofs are computed in time $O(1)$, otherwise in time $O(n^\epsilon \log n)$. Finally, insertions to and deletions from the set take $O(n^\epsilon)$ time.

3. PATTERN MATCHING QUERIES

The problem of pattern matching involves determining whether a pattern appears within a given text. In its basic form, assuming an alphabet Σ of size $|\Sigma| = \sigma$, a n-character text $\mathcal{T} \in \Sigma^n$ and a pattern $p \in \Sigma^m$ of length m, the problem is expressed as "*is there position $1 < i \leq n - m + 1$ such that $\mathcal{T}[i + j] = p[j]$ for $j = 0, \ldots m - 1$?*", where $\mathcal{T}[i]$ is the character at the i-th position in the text, and likewise for pattern p. If there exists such i, the answer is "match at i", otherwise "mismatch".

Answering pattern matching queries is an arduous task, if done naively. For instance, to check the occurrence of p in T, one could sequentially test if p at any position i (i.e., if it is a prefix of some suffix of $\mathcal{T}$), for all positions in $\mathcal{T}$. Such a successful test would

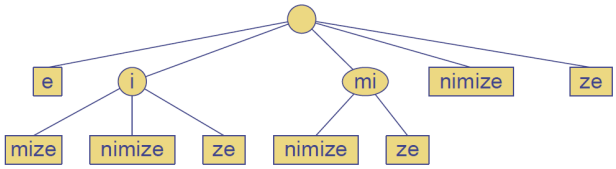


Figure 1: Suffix tree for minimize **storing suffixes** minimize**,** inimize**,** nimize**,** imize**,** mize**,** ize**,** ze**,** e **as eight overlapping paths.**

imply a match but would require $O(n)$ work. However, with some preprocessing of $O(n)$ work, one can organize patterns in a *suffix tree* [38], reducing the complexity of pattern matching query from $O(n)$ to $O(m)$. A suffix tree is a data structure storing *all* the suffixes of $\mathcal{T}$ in a way such that any repeating patterns (common prefixes) of these suffixes are stored once and in an hierarchical way, so that every leaf of the suffix tree corresponds to a suffix of the text $\mathcal{T}$. This allows for (reduced) $O(m)$ search time while maintaining (linear) $O(n)$ space usage. We provide next a more detailed description of the suffix tree data structure, represented as a directed tree $G = (V, E, \mathcal{T}, \Sigma)$. We refer to the example of Figure 1 depicting the suffix tree for the word minimize.

Each leaf of G corresponds to a distinct suffix of $\mathcal{T}$, thus G has exactly n leaves. We denote with $S[i]$ the i-th suffix of T, that is, $S[i] = \mathcal{T}[i] \dots \mathcal{T}[n]$, for $i = 1, \dots, n$. Internal tree nodes store common prefixes of these n suffixes $S[1], S[2], \dots, S[n]$, where the leaves themselves store any "remainder" non-overlapping prefixes of T's suffixes. If leaf v_i corresponds to suffix $S[i]$, then $S[i]$ is formed by the *concatenation* of the contents of all nodes in the root-to-leaf path defined by v_i, where the root conventionally stores the empty string. For instance, in Figure 1, $S[4] =$ imize and $S[6] =$ ize, respectively associated with the paths defined by the second and fourth most left leaves, labelled by mize and ze (having as common parent the node labelled by i).

Additionally, every node $v \in V$ stores the following information that will be useful in the case of the mismatch: **(a)** the *range* $r_v = (s_v, e_v)$ of v, which corresponds to the start (s_v) and end (e_v) position of the string stored in v in the text (we pick an arbitrary range if v is associated with multiple ranges); **(b)** the *depth* d_v of v, which corresponds to the number of characters from the root to v, i.e., the number of characters that are contained in the path in G that consists of the ancestors of v; **(c)** the *sequel* $\mathcal{C}_v$ of v, which corresponds to the set of initial characters of the children of v. For example in Figure 1, for the node v labelled mi, it is $s_v = 1$ and $e_v = 2$ (or $s_v = 5$ and $e_v = 6$), $d_v = 0, \mathcal{C}_v = \{n, z\}$.

Traversing a suffix tree. Since all matching patterns must be a prefix of some suffix, the search algorithm beings from the root and traverses down the tree incrementally matching pattern p with the node labels, until it reaches some node v where either a mismatch or a complete match is found. We model this search on suffix tree $G = (V, E, \mathcal{T}, \Sigma)$ by algorithm $(v, k, t) \leftarrow \mathsf{suffixQuery}(p, G)$, returning: (1) the *matching node* v, i.e., the node of G at which the algorithm stopped with a match or mismatch, (2) the *matching index* k, $s_v \le k \le e_v$, i.e., the index (with reference to the specific range (s_v, e_v)) where the last matching character occurs (for successful matching searches, k coincides with the index of the last character of p within v), and (3) the *prefix size* $t \le m$, i.e., the length of maximum matching prefix of p (m in case of a match). Figure 2 shows the relation of variables k and t for both cases.

Characterization of pattern matching queries. The following lemmas *characterize* the correct execution of algorithm suffixQuery, by providing necessary and sufficient conditions for checking the consistency of a match or mismatch of p in T with the output (v, k, t) produced by suffixQuery. In the next section, we will base the security of our construction on proving, in a *cryptographic* manner, that these conditions hold for a given query-answer pair. Namely, the structure of these relations allows us to generate *succinct* and efficiently verifiable proofs. In the following we denote with xy the concatenation of two strings x, y (order is important).

LEMMA 1 (PATTERN MATCH). *There is a* match *of p in $\mathcal{T}$ if and only if there exist two suffixes of $\mathcal{T}$, $S[i]$ and $S[j]$, with $i \le j$, such that $S[i] = pS[j]$.*

LEMMA 2 (PATTERN MISMATCH). *There is a* mismatch *of p in $\mathcal{T}$ if and only if there exist a node $v \in G$, an integer $k \in [s_v, e_v]$ and an integer $t < m$ such that $S[s_v - d_v] = p_1 p_2 \dots p_t S[k+1]$ and $p_{t+1} \neq \mathcal{T}[k+1]$, if $k < e_v$, or $p_{t+1} \notin c_v$ if $k = e_v$.*

With reference to Figure 1, the match of the pattern $p =$ inim, can be shown by employing the suffixes $S[2] =$ inimize and $S[6] =$ ize. Note that indeed $S[2] = pS[6]$. This is a match (as in Lemma 1). More interestingly, observe the case of mismatch for the string $p =$ minia. For this input, algorithm suffixQuery returns the node v labelled by nimize where the mismatch happens, matching index $k = 4$ and prefix size $t = 4$. For node v, we have $s_v = 3$ and $e_v = 8$ and also $d_v = 2$. To demonstrate the mismatch, it suffices to employ suffixes $S[s_v - d_v] = S[1] =$ minimize and $S[k+1] = S[5] =$ mize as well as symbols p_{t+1} and $\mathcal{T}[k+1]$. The concatenation of the prefix mini of p (of size $t = 4$) with the suffix $S[5]$ is $S[1]$, and also $p_{t+1} =$ a $\neq \mathcal{T}[k+1] =$ m. This is a mismatch (as the first case considered in Lemma 2, since $k < e_v$). Finally, to demonstrate the mismatch of the string $p =$ mia we proceed as follows. Algorithm suffixQuery returns the node v labelled by mi where the mismatch happens, matching index $k = 6$ and prefix size $t = 2$. For node v, we have $s_v = 5$ and $e_v = 6$ (alternatively we can also have $s_v = 1$ and $e_v = 2$) and also $d_v = 0$. It suffices to employ suffixes $S[s_v - d_v] = S[5] =$ mize and $S[k+1] = S[7] =$ ze as well as symbol p_{t+1} and sequel (set) c_v. Note that indeed the concatenation of the prefix mi of p (of size $t = 2$) with the suffix $S[7]$ is $S[5]$, and that also $p_{t+1} =$ a $\notin c_v = \{$n, z$\}$. This is a mismatch (as in Lemma 2, since $k = e_v$).

4. MAIN CONSTRUCTION

We now present our main ADS scheme for verifying answers to pattern matching queries. Our construction is based on building a suffix tree over the outsourced text and proving in a secure way the conditions specified in Lemmas 1 and 2 for the cases of match and mismatch respectively. The main cryptogaphic tool employed is the BM accumulator, which will be use to authenticate the contents of a suffix tree in a structured way, allowing the server to prove the existence of appropriate suffixes in the text and values in the tree that satisfy the conditions in the two lemmas. Moreover, due to the properties of the BM accumulator, the produced proofs will be independent of the size of the text and the pattern, consisting only of a *constant* number of bilinear group elements.

Key generation. The text owner first computes public bilinear parameters $\mathsf{pub} = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ for security parameter 1^k. He then picks a random $s \in \mathbb{Z}_p$ and computes $\mathbf{g} = [g, g^s, \dots, g^{s^\ell}]$. Finally, the key pair is defined as $\mathsf{sk} = s$, $\mathsf{pk} = (\mathsf{pub}, \mathbf{g})$.

Setup. The setup process is described in pseudo-code in Algorithm 1 and we provide a detailed explanation of each step here. The owner first computes a *suffix accumulation* for each suffix in the text with a linear pass. This value encodes information about the text contents of the suffix, its starting position and its leading character. In particular, $\mathsf{acc}(S[i])$ is denoted as $\mathsf{acc}_i := \mathsf{acc}(\mathcal{X}_{i1} \cup$

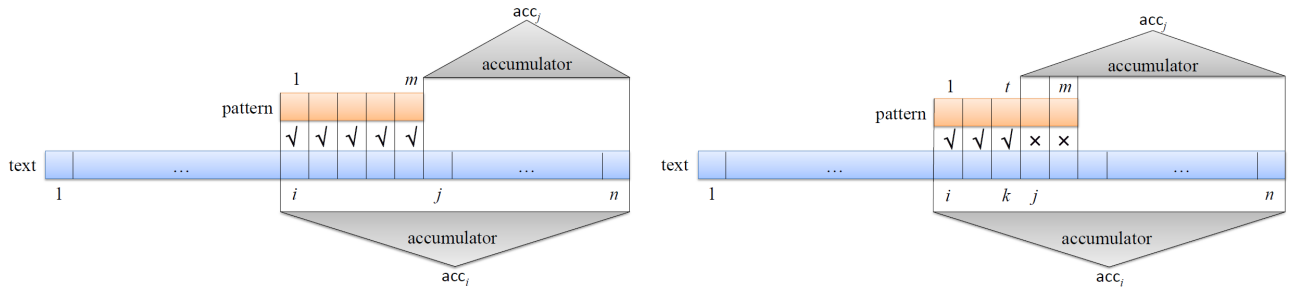


Figure 2: (Left) Pattern matching in our scheme for pattern p ($|p| = m$), using suffixes $S[i]$ and $S[j]$, where $S[i] = pS[j]$. (Right) Pattern mismatch in our scheme, using suffixes $S[i]$ and $S[j]$, where $S[i] = p_1p_2 \dots p_tS[j]$ and $t < m$.

$\mathcal{X}_{i2} \cup \mathcal{X}_{i3}$), where **(a)** $\mathcal{X}_{i1}$ is the set of *position-character* pairs in suffix $S[i]$, i.e., $\mathcal{X}_{i1} = \{(\mathsf{pos}, i, \mathcal{T}[i]), \dots, (\mathsf{pos}, n, \mathcal{T}[n])\}$; **(b)** $\mathcal{X}_{i2}$ is the *first character* of $S[i]$, i.e., $\mathcal{X}_{i2} = \{(\mathsf{first}, \mathcal{T}[i])\}$; and **(c)** $\mathcal{X}_{i3}$ is the *index* of $S[i]$, i.e., $\mathcal{X}_{i3} = \{(\mathsf{index}, i)\}$. Also, for each suffix $S[i]$ he computes a *suffix structure accumulation* $t_i = \mathsf{acc}(\mathcal{X}_{i1})$, i.e. it contains only the position-character pairs in the suffix and its use will be discussed when we explain the verification process of our scheme. Structure accumulations are a very important part for the security of our construction. Observe that the suffix structure accumulation t_i encompasses only a *subset* of the information encompassed in acc_i. As shown in Section 2, the security of the BM accumulator makes proving a false subset relation impossible, hence no efficient adversary can link t_i with acc_j for $j \neq i$.

Following this, the owner builds a suffix tree $G = (V, E, \mathcal{T}, \Sigma)$ over the text and computes a *node accumulation* for each $v \in G$. This value encodes all the information regarding this node in G, i.e., the range of $\mathcal{T}$ it encompasses, its depth in the tree and the leading characters of all its children nodes (taken in consecutive pairs). More formally, for a node v with values $(s_v, e_v), d_v, c_v$, its accumulation is defined as $\mathsf{acc}_v := \mathsf{acc}(\mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3})$, where: **(a)** $\mathcal{Y}_{v1}$ is the *range* of v , i.e., $\mathcal{Y}_{v1} = \{(\mathsf{range}, s_v, e_v)\}$; **(b)** $\mathcal{Y}_{v2}$ is the *depth* of v, i.e., $\mathcal{Y}_{v2} = \{(\mathsf{depth}, d_v)\}$; and **(c)** $\mathcal{Y}_{v3}$ is the *sequel* of v defined as the set of consecutive pairs $\mathcal{Y}_{v3} = \{(\mathsf{sequel}, c_i, c_{i+1}) | i = 1, \dots, \ell-1\}$ where $\mathcal{C}_v) = \{c_1, c_2, \dots, c_\ell\}$ is the alphabetic ordering of the first characters of v's children. He also computes a *node structure accumulator* $t_v = \mathsf{acc}(\mathcal{Y}_{v3})$ (similar to what we explained for suffixes). Finally, for each sequel c_i, c_{i+1}, compute a subset witness $\mathsf{W}_{\mathcal{P},\mathcal{Y}}$ (as defined in Section 2) where $\mathcal{P} = \{(\mathsf{sequel}, c_j, c_{j+1})\}$ and $\mathcal{Y} = \mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3}$. This will serve to prove that the given sequel of characters are leading characters of consecutive children of v.

Note that, the keywords pos, first, index, range, depth and sequel are used as *descriptors* of the value that is accumulated. Without loss of generality one can view the elements of sets $\mathcal{X}_{ij}$, $\mathcal{Y}_{ij}$, $j \in \{1, 2, 3\}$ as distinct k-bit strings (each less than the group's prime order $p \in O(2^k)$), simply by applying an appropriate deterministic encoding scheme $\mathbf{r}(\cdot)$. Therefore, when we accumulate the elements of these sets, we are in fact accumulating their *numerical* representation under encoding $\mathbf{r}$. This allows us to represent all accumulated values as distinct elements of $\mathbb{Z}_p$, achieving the necessary domain homogeneity required by the BM accumulator.

At the end of this procedure, each suffix $S[i]$ has its suffix accumulation acc_i and its suffix structure accumulation t_i. Also, each node $v \in G$ is associated with its node accumulation acc_v, its node structure accumulation t_v and one subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$ for each consecutive pair of its children. We denote with $\mathcal{V}, \mathcal{S}$ the sets of node and suffix accumulations acc_v and acc_i, respectively. As a final step, the owner builds two accumulation trees $\mathcal{AT}_\mathcal{V}, \mathcal{AT}_\mathcal{S}$, using the BM accumulator described by the key pair. Let d_V, d_S be their respective digests. He sends to the server the text $\mathcal{T}$, as well as authentication information $auth(\mathcal{T})$ consisting of the suffix tree G, the two accumulation trees $\mathcal{AT}_\mathcal{V}, \mathcal{AT}_\mathcal{S}$ and all values $\mathsf{acc}_i, \mathsf{acc}_v, t_i, t_v, \mathsf{W}_{\mathcal{P},\mathcal{C}}$, and publishes $\mathsf{pk}, d_\mathcal{V}, d_\mathcal{S}$.

Algorithm 1: $\mathsf{setup}(\mathcal{T}, pk, sk)$
1. **For** suffix $i = 1, \dots, n$
2. Compute suffix structure accumulation t_i
3. Compute suffix accumulation acc_i
4. Build suffix tree $G = (V, E, \mathcal{T}, \Sigma)$
5. **For each** node $v \in G$
6. Compute node structure accumulation t_v
7. Compute node accumulation acc_v
8. **For each** consecutive pair of children of v
9. Compute subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$
10. Build accumulation trees $\mathcal{AT}_\mathcal{V}, \mathcal{AT}_\mathcal{S}$
11. Send $\mathcal{T}, auth(\mathcal{T})$ to the server and publish $\mathsf{pk}, d_\mathcal{V}, d_\mathcal{S}$

Proof generation. We next describe proof generation for pattern matching queries, i.e., matches and mismatches. The process varies greatly for the two cases as can be seen in Algorithm 2 below. The role of each proof component will become evident when we discuss the verification process in the next paragraph. In both cases, let (v, k, t) be the matching node in G, the matching index and the prefix size returned by algorithm $(v, k, t) \leftarrow \mathsf{suffixQuery}(p, G)$ (as described in Section 3 and Figure 2).

Proving a match. In this case the answer is $\alpha(q) =$ "match at i". Let $p = p_1p_2 \dots p_m$ be the queried pattern. The server computes $i = s_v - d_v$ and $j = i + m$. By Lemma 1, suffixes $S[i]$ and $S[j]$ are such that $S[i] = pS[j]$ and $i \leq j$. The corresponding indexes are easily computable by traversing the suffix tree for p. The server returns i, j along with characters $\mathcal{T}[i], \mathcal{T}[j]$ as well as suffix structure and suffix accumulations $\mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j$. Finally, using accumulation tree $\mathcal{AT}_\mathcal{S}$, he computes proofs π_i, π_j for validating that $\mathsf{acc}_i, \mathsf{acc}_j \in \mathcal{S}$.

Proving a mismatch. In this case the answer to the query q is $\alpha(q) =$ "mismatch". Let (s_v, e_v), d_v and c_v be the range, depth and sequel of v. The server computes $i = s_v - d_v$ and $j = i + k + 1$ and returns $s_v, e_v, d_v, k, t, i, j, \mathcal{T}[i], \mathcal{T}[j]$ along with accumulations $\mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j$ with proofs π_v, π_i, π_j and structure accumulation values t_v, t_i, t_j. Finally, if $k = e_v$ he also returns $\mathsf{W}_{\mathcal{P},\mathcal{C}}$ where $\mathcal{P}$ contains sequel c, c' such that $c < p_{t+1} < c'$.

Verification. Here we describe the verification algorithm of our scheme. Below we provide the pseudo-code in Algorithm 3 and an intuitive explanation for the role of each component of the proof. In both cases, the verification serves to check the conditions stated in Lemmas 1, 2, which suffices to validate that the answer is correct.

Algorithm 2: $\mathsf{query}(q, \mathcal{T}, auth(\mathcal{T}), \mathsf{pk})$

1. Call $\mathsf{suffixQuery}(p, G)$ to receive (v, k, t)
2. Set $i = s_v - d_v$
3. **If** $t = m$ **Then**
4. Set $a(q) =$ "match at i" and $j = i + m$
5. Lookup $\mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j$ in $auth(\mathcal{T})$
6. Compute $\mathcal{AT}_{\mathcal{S}}$ proofs π_i, π_j for $\mathsf{acc}_i, \mathsf{acc}_j$
7. Set $\Pi(q) = (j, \mathcal{T}[i], \mathcal{T}[j], \mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j, \pi_i, \pi_j)$
8. **Else**
9. Set $a(q) =$ "mismatch" and $j = i + k + 1$
10. Lookup $\mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j, t_v, t_i, t_j$ in $auth(\mathcal{T})$
11. Compute $\mathcal{AT}_{\mathcal{S}}$ proofs π_i, π_j for $\mathsf{acc}_i, \mathsf{acc}_j$
12. Compute $\mathcal{AT}_{\mathcal{V}}$ proof π_v for acc_v
13. Set $\mathsf{aux} = (s_v, e_v, d_v, i, j, k, t)$
14. Set $\Pi(q) = (\mathsf{aux}, \mathcal{T}[i], \mathcal{T}[j], \mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j, t_v, t_i, t_j, \pi_v, \pi_i, \pi_j)$
15. **If** $k = e_v$ **Then**
16. Traverse the sequels of v to find pair c, c' s.t. $c < p_{t+1} < c'$
17. Let $\mathcal{P} = \{(\mathsf{sequel}, c, c')\}$
18. Lookup subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$
19. Set $\Pi(q) \cup \{\mathcal{P}, \mathsf{W}_{\mathcal{P},\mathcal{C}}\}$
20. Output $a(q), \Pi(q)$

Algorithm 3: $\mathsf{verify}(q, \alpha(q), \Pi(q), d, \mathsf{pk})$

1. Verify $\mathsf{acc}_i, \mathsf{acc}_v$ with respect to $d_{\mathcal{S}}$, with π_i, π_j
2. Compute $g^{\mathbf{x}}$ for $\mathbf{x} = (s + \mathbf{r}(\mathsf{first}, \mathcal{T}[i]))(s + \mathbf{r}(\mathsf{index}, i))$
3. Compute $g^{\mathbf{y}}$ for $\mathbf{y} = (s + \mathbf{r}(\mathsf{first}, \mathcal{T}[j]))(s + \mathbf{r}(\mathsf{index}, j))$
4. Verify that $e(t_i, g^{\mathbf{x}}) = e(\mathsf{acc}_i, g)$ and $e(t_j, g^{\mathbf{y}}) = e(\mathsf{acc}_j, g)$
5. **If** $\alpha(q) =$ "match at i" **Then**
6. Compute $g^{\mathbf{p}}$ for $\mathbf{p} = \prod_{l=1}^{m}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$
7. Verify that $e(t_i, g) = e(t_j, g^{\mathbf{p}})$
8. **Else**
9. Compute $g^{\mathbf{p}}$ for $\mathbf{p} = \prod_{l=1}^{t}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$
10. Verify that $e(t_i, g^{\mathbf{p}}) = e(t_j, g)$
11. Verify that $i = s_v - d_v$ and $s_v \leq k \leq e_v$ and $j = i + k + 1$
12. Verify acc_v, with respect to $d_{\mathcal{V}}$, with π_v
13. Compute $g^{\mathbf{z}}$ for $\mathbf{z} = (s + \mathbf{r}(\mathsf{range}, s_v, e_v))(s + \mathbf{r}(\mathsf{depth}, d_v))$
14. Verify that $e(t_v, g^{\mathbf{z}}) = e(\mathsf{acc}_v, g)$
15. **If** $k < e_v$ **Then** verify that $p_{t+1} \neq \mathcal{T}[j]$
16. **Else**
17. Verify that $c < p_{t+1} < c'$ (alphabetically)
18. Compute $g^{\mathbf{w}}$ for $\mathbf{w} = s + \mathbf{r}(\mathsf{sequel}, c, c')$
19. Verify that $e(\mathcal{W}_{\mathcal{P},\mathcal{C}}, g^{\mathbf{w}}) = e(\mathsf{acc}_v, g)$
20. **If** any check fails **Then** output `reject`, **Else** `accept`

Verifying a match. Recall that, by Lemma 1, it suffices to validate that there exist suffixes $S[i], S[j]$ in the text, such that $S[j] = pS[i]$. First the client verifies that $\mathsf{acc}_i, \mathsf{acc}_j \in \Pi(q)$ are indeed the suffix accumulations of two suffixes of $\mathcal{T}$ using proofs π_i, π_j (Line 1). Then, it checks that the corresponding structure accumulations are indeed t_i, t_j (Lines 2-4). It remains to check that the "difference" between them is p (Lines 6-7), by first computing the pattern accumulation value $g^{\mathbf{p}}$) for $\mathbf{p} = \prod_{l=1}^{m}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$. A careful observation shows that this is indeed the "missing" value between the honestly computed structure accumulations t_i, t_j. This can be cryptographically checked by a single bilinear equality testing $e(t_i, g) = e(t_j, g^{\mathbf{p}})$. This last step can be viewed as an accumulator-based alternative to chain-hashing using a collision-resistant hash function. It follows from the above that, if all these checks succeed, the conditions of Lemma 1 are met.

Verifying a mismatch. The case of a mismatch is initially similar to that of a match, however it eventually gets more complicated. The client begins by verifying the same relations as for the case of a match for two indices i, j (Lines 1-4). In this case, these positions correspond to two suffixes $S[i], S[j]$ such that $S[j] = p'S[i]$, where p' is a prefix of p, i.e., their difference is a beginning part of the pattern (Lines 9-10). Unfortunately, this is not enough to validate the integrity of the answer. For example, a cheating adversary can locate the occurrence of such a prefix of p in the text, and falsely report its position, ignoring that the entire p appears in $\mathcal{T}$ as well. We therefore need to prove that p' is the maximal prefix of p appearing in the text and here is where the properties of the suffix tree become useful. In particular, if two characters appear consecutively within the same node of G, it must be that every occurrence of the first one in $\mathcal{T}$ is followed by the second one. Hence, if the server can prove that the part of $\mathcal{T}$ corresponding to the final part of p' as well as the consequent character, both fall within the same node and said consequent character is not the one dictated by p, it must be that p' truly is the maximal prefix of $p \in \mathcal{T}$. This is done by checking the relation between the node accumulation and the node structure accumulation of the returned node v (Lines 11-15).

This however does not cover the case where the consequent character, after p', falls within the a child node of v (i.e., the part of $\mathcal{T}$ corresponding to p', ends at the end of the range of v). To accommodate for this case, the server needs to prove that the next character in p, does not appear as the leading character of any of v's children. Since, all these characters have been alphabetically ordered and accumulated in consecutive pairs, it suffices to return the corresponding pair $\mathcal{P}$ that "covers" this consequent character. The validity of this pair is guaranteed by providing the related pre-computed witness, the relation of which to node v is tested by checking a bilinear equality (Lines 17-19).

We can now state our main result (proof is included in the full version of our paper).

THEOREM 1. *The algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{query}, \mathsf{verify}\}$ *are a* correct ADS *scheme for pattern matching queries that is* secure *under the* ℓ-sDH *assumption.*

Complexity analysis. The running time of algorithm setup is $O(n)$. This follows immediately from the following: (i) the construction of G takes $O(n)$ and the produced tree contains $O(n)$ nodes, (ii) all suffix and suffix structure accumulations can be computed with a single pass over $\mathcal{T}$, (iii) node and node structure accumulation values can be computed in time linear to the number of the node's children (using sk); since each node has a unique parent node, all node accumulations are also computable in time $O(n)$, and (iv) an accumulation tree over n elements can be constructed in time $O(n)$. The running time of algorithm query is $O(m)$, because all proof components in $\Pi(q)$ are pre-computed (including $\mathcal{AT}$ proofs if the accumulation trees are of height 1), hence the only costly component is the suffix tree traversal which takes $O(m)$. For algorithm verify the runtime is $O(m \log m)$. This holds because verification of $\mathcal{AT}$ proofs can be done with $O(1)$ operations, accumulating a set of m elements, with pk alone, takes $O(m \log m)$ operations and only a constant number of checks is made. The proof consists of a *constant* number of bilinear group elements (at most ten, corresponding to the case of a mismatch at the end of a node). Finally the overall storage space for $auth(\mathcal{T})$ is $O(n)$.

Handling wildcards. Our construction can be easily extended to support pattern matching queries expressed as limited regular expressions. In particular, it can accommodate queries with patterns containing a constant number of "wildcards" (e.g., $*$ or ?). To achieve this we proceed as follows. Partition p into segments associated with simple patterns, with the wildcards falling between them. Proceed to run proof generation and verification for each segment individually. For the mismatch case, it suffices for the server to demonstrate that just one of these segments does not appear in $\mathcal{T}$. For the match case, the server proves existence for all segments and the clients verifies each one separately. He then checks that the positions of occurrence (expressed as the i, j indices of each

segment) are "consistent", i.e., they fall in the correct order within the text (or they have the specified distance in case there is a corresponding restriction in the query specification).

Parallel setup. We also show how to derive parallel implementations for the setup algorithm, assuming $O(n/\log n)$ processors in the *exclusive-read-exclusive-write* (EREW) model [19].

BM accumulator setup. Given the trapdoor information s, the accumulation $\mathsf{acc}(\mathcal{X})$ of a set $\mathcal{X}$ of size n, can be computed in $O(\log n)$ parallel time. This is achieved with an algorithm for summing n terms in parallel (where sum is replaced with multiplication) [19].

Suffix products. Suffix accumulations can be computed by simply using a parallel prefix sums algorithm, in $O(\log n)$ parallel time.

Accumulation trees. Accumulation trees on a set of n elements, can also be constructed in parallel. First, partition the elements of the set in $O(n/\log n)$ buckets of size $O(\log n)$ and then compute the accumulations of the buckets in $O(\log n)$ parallel time. Next, for a fixed ϵ, build the accumulation tree on top of the $B = n/\log n$ buckets. Specifically, the accumulations ($O(B^{1-\epsilon})$ in total) of all internal nodes (of degree $O(B^{\epsilon})$) at a specific level can be computed independently from one another. Therefore, by the parallel accumulation setup algorithm (presented in the beginning of this section), the accumulation tree can be computed in $O(\log B^{\epsilon}) = O(\log n)$ parallel time using $O(B^{1-\epsilon}B^{\epsilon}/\log B) = O(n/\log n)$ processors, similarly with a Merkle tree. It follows that setup takes $O(\log n)$ parallel time with $O(n/\log n)$ processors.

5. APPLICATIONS

In this section we discuss two practical applications of our construction. We first show how our scheme can be used to accommodate pattern matching queries over a collection of documents and then explain how our BM authentication technique can be modified to support a class of queries over semi-structured data, namely XML documents. Finally, we discuss how our construction can be extended to efficiently handle modifications in the dataset.

5.1 Search on collection of text documents

We generalize our main construction to handle queries over multiple documents. By adding some modifications in the suffix tree authentication mechanism, we build a scheme that supports queries of the form "return all documents that contain pattern p". This enhancement yields a construction that is closer to real-world applications involving querying a corpus of textual documents.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_\tau$ be a collection of τ documents, with content from the same alphabet Σ. Without loss of generality, assume each of them has length n, and let N be the sum of the lengths of all $\mathcal{T}_i$, i.e., $N = \tau n$. We assume a data structure that upon input a query q, expressed as string pattern p from Σ, returns the index set $\mathcal{I} := \{i | p$ appears in $\mathcal{T}_i\}$, i.e., the indices of all documents that contain the pattern. Using our construction as a starting point, one straightforward solution for authenticating this data structure is to handle each $\mathcal{T}_i$ separately, building and authenticating a corresponding suffix tree. Consequently, in order to prove the integrity of his answer, the server replies with τ separate proofs of our main construction (one for each document) which are separately verified by the client. This approach is clearly not efficient since τ can be very large in practice; shorter proofs are not possible, since a server can cheat simply by omitting the answer for some documents and the client has no way to capture this unless he receives a proof for all of them.

Main idea. We handle all documents as a single document $\mathcal{T} = \mathcal{T}_1 * \mathcal{T}_2 * \ldots * \mathcal{T}_\tau$ expressed as their concatenation, where $*$ is a special character $\notin \Sigma$ marking the end of a document. We define extended alphabet $\Sigma^* = \Sigma \cup \{*\}$ and build a single authenticated suffix tree $G = (V, E, \mathcal{T}, \Sigma^*)$ as in our main construction. Observe now that the query can be reduced to answering a single pattern matching query for p in $\mathcal{T}$, asking for *all* its occurrences (as opposed to our main scheme where we were interested with a single occurrence). This can be easily achieved with the following observation about suffix trees: for a pattern p for which $\mathsf{suffixTree}$ outputs node $v \in G$, the number of occurrences of $p \in \mathcal{T}$, is the number of children of G. For example, in Figure 1, the pattern i appears three times in the text, and the pattern mi appears twice.

Construction overview. The above relation can be incorporated in our main construction, by encoding in each node v not a single range (s_v, e_v) but the indices of all these ranges (s_u^v, e_u^v), one for each child node u of v. In fact, the information will consist of triples (i, s_u^v, e_u^v) where i is the index of the document $\mathcal{T}_i$ within which s_v falls. This can performed in time $O(n)$ in three steps. Initially, the owner sets up an efficient dictionary structure with key-value pairs formed by document indices and corresponding starting positions. Then he sets up a suffix tree G for $\mathcal{T}$ and with a post-order traversal computes all ranges for each node (with lookups to the dictionary). He finally runs the setup for our main construction with the modified node information explained above.

Regarding proof generation and verification, we distinguish between the two cases. If p does not appear in $\mathcal{T}$, then the proof is same as in our basic scheme and the same holds for verification[4]. For the case of positive response, the server must return a proof that consists of three parts: (i) a match proof exactly as in our main construction, with boundaries i, j; (ii) a node accumulation acc_v (with its accumulation tree proof and structure accumulation) for the node v corresponding to p and all its ranges; (iii) the indices of all documents where p appears. With access to all this information, the client verifies that p indeed appears in $\mathcal{T}$, it corresponds to v (because there must exist one range of v that covers position j) and that the returned indices correspond exactly to *all* documents containing p. Observe that the special character $*$ makes it impossible for an adversary to cheat by finding two consecutive documents, the first of which ends with a prefix of p and the second of which begins with the corresponding suffix (as long as $\{*\} \notin p$).

5.2 Search on XML documents

We now turn our attention to queries over XML documents. We consider the standard tree-based representation of XML data: An XML document X consists of n elements $e_1, \ldots, e_n$ organized in a hierarchical structure, via sub-element relations, which, in turn, imposes a well-defined representation of X as a tree $\mathcal{X}_T$ having elements as nodes and sub-element relations expressed by edges. Each element has a *label* that distinguishes its data type, *attributes* and corresponding *attribute values* and actual *textual content* (which can be viewed as an additional attribute).[5] We also assume that each element of $\mathcal{X}_T$ is associated with a unique numerical identifier stored as an element attribute. Figure 3 provides one such simplified tree-based representation.

Each node e in $\mathcal{X}_T$ is defined (or reachable) by a single *label path* that is the concatenation of the labels of e's ancestor nodes in the order that they must be traversed starting from the root of $\mathcal{X}_T$ in order to reach e. In general, many elements may share the same label path. We abstract away the details of the (often elaborate) querying process of an XML document by considering generic *path queries* that return a subset of the elements of $\mathcal{X}_T$ (in fact,

[4]The node accumulations must include separately a single range for v (randomly chosen in the case of multiple occurrences) and the collection of all ranges described above.

[5]We do not consider reference attributes relating elements to arbitrary nodes in $\mathcal{X}_T$) or processing instructions.

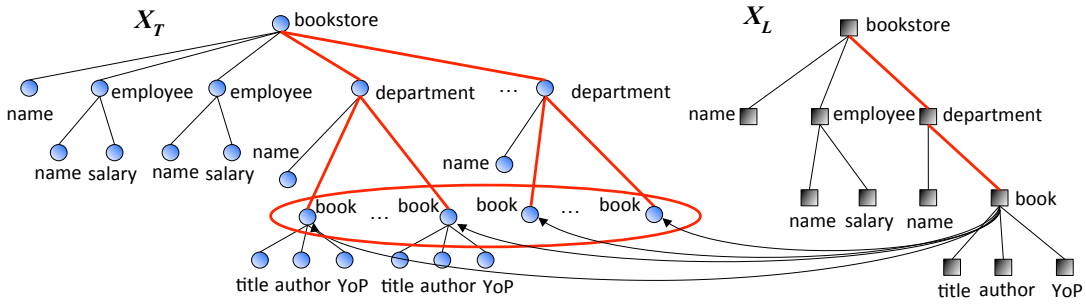


Figure 3: (Left) Tree $\mathcal{X}_T$ containing all the elements of XML document X. Element attributes can be included as a different type of node, directly below the corresponding element. (Right) Trie $\mathcal{X}_L$ containing all the distinct label paths that appear at X.

a forest of subtrees in $\mathcal{X}_T$). A *path query* is generally a regular expression over the alphabet $\mathcal{L}$ of valid labels returning all nodes reachable by those label paths conforming to the query, along with the subtrees in $\mathcal{X}_T$ rooted at these nodes. An *exact path query* is related to a label path L of length m, i.e., $L \in \mathcal{L}^m$, returning the subtrees reachable in $\mathcal{X}_T$ by L. This abstraction fully captures the basic notion of path query as identified in various XML query languages, e.g., XPath, XML-QL. As an example a query of the form \bookstore\department\book will return all the books that appear in $\mathcal{X}_T$ as shown in Figure 3 *with* the corresponding subtree of each book element (i.e., nodes title, author, YoP).

Main idea. Similar to the case of text pattern matching, our goal is to identify the relations among the elements of XML document that are sufficient to succinctly certify the correctness of exact-path XML queries. Our main approach is to *decouple locating the queried elements* from *validating their contents*. We achieve this through a direct reduction to our (authenticated) suffix tree construction from the previous section: Given an XML document X in its tree-like representation $\mathcal{X}_T$, we construct a *trie* $\mathcal{X}_L$ that stores *all the distinct label paths that appear in* $\mathcal{X}_T$. Compared to our main scheme, $\mathcal{X}_L$ can be viewed as an *uncompressed suffix tree (trie)* with the alphabet being the element label space $\mathcal{L}$ associated with X and the "text" over which it is defined being all label paths in $\mathcal{X}_T$. Each node in $\mathcal{X}_L$ is associated with a valid label path according to $\mathcal{X}_T$ and also with the set of elements in $\mathcal{X}_T$ that are reachable by this label path, through back pointers. For example, the query \bookstore\department\book in Figure 3 will reach one node in $\mathcal{X}_L$ which points back to the elements reachable by the queried path. We define, encode and authenticate three types of certification relations (corresponding to edges in Figure 3):

1. *Subtree contents:* This relation maps nodes in $\mathcal{X}_T$ with the elements (and their attributes) in $\mathcal{X}_T$ that belong in the subtrees in $\mathcal{X}_T$ defined by these nodes.

2. *Label paths*: This relation maps nodes in $\mathcal{X}_L$ with their corresponding label paths. Here, we make direct use of our results from Section 4; however, since we no longer have a tree defined over all possible suffixes of a text, suffix accumulations are no longer relevant (instead, we use node accumulations).

3. *Element mappings:* This relation maps nodes in $\mathcal{X}_L$ with the corresponding elements in $\mathcal{X}_T$ that are reachable by the same label path (associated with these nodes).

We next describe how to cryptographically encode the above relations by carefully computing accumulations or hash values over sets of data objects related to the nodes in $\mathcal{X}_L$ and $\mathcal{X}_T$.

Notation. We denote by e_{id} the identifier, by $A_e = \{(a_i, \beta_i) | i = 1, \ldots, |A_e|\}$ the attribute values, by $lb(e)$ the label, and by $C_e = \{c_i | i = 1, \ldots, |C_e|\}$ the children of element e in $\mathcal{X}_T$. Also, for node $v \in \mathcal{X}_L$, we denote by L_v, $lb(v)$, C_v, and E_v its label path, label, children set, and respectively the set of elements e_i in $\mathcal{X}_T$ that are reachable by L_v. Finally, let d be the height of $\mathcal{X}_T$.

Subtree labels. Subtree contents in $\mathcal{X}_T$ are encoded using a special type of node-specific values. If h is a cryptographic collision-resistant hash function, then for any $e \in \mathcal{X}_T$ we let h_e denote the *hash content* of e $h_e = h(e_{id} \| (a_1, \beta_1) \| \ldots \| (a_{|A_e|}, \beta_{|A_e|}))$. Then, for $e \in \mathcal{X}_T$ we define two different ways for recursively computing node-specific *subtree labels* $\mathsf{sl}(e)$ that *aggregate the hash labels of all the descendant nodes of* e *in* $\mathcal{X}_T$:

1) Hash based: If e is leaf in $\mathcal{X}_T$ then $\mathsf{sl}_1(e) = h_e$, otherwise $\mathsf{sl}_1(e) = h(h_e \| \mathsf{sl}_1(c_1) \ldots \| \mathsf{sl}_1(c_{|C_e|}))$;

2) Accumulation based: $\mathsf{sl}_2 = \mathsf{acc}(\mathcal{Z}_e)$ where if e is leaf then $\mathcal{Z}_e = h_e$, otherwise $\mathcal{Z}_e = \{h_e, \mathcal{Z}_{c_1}, \ldots, \mathcal{Z}_{c_{|C_e|}}\}$;

Node accumulations. Label paths and element mappings in $\mathcal{X}_L$ are encoded using node accumulations. We associate with $v \in \mathcal{X}_L$ three sets of data objects: **(a)** The *label path* L_v of v; let $\mathcal{Y}_{v1} = \{(\mathsf{label}, i, l_i) : i = 1, \ldots, |L_v|\}$; **(b)** The *label sequels* of v is $lb(c_1), \ldots, lb(c_{|C_v|})$, the sequence of the alphabetically ordered labels of v's children; let $\mathcal{Y}_{v2} = \{(\mathsf{sequel}, lb(c_i), lb(c_{i+1})) : i = 1, \ldots, |C_v| - 1\}$; and **(c)** The *XML elements hash* is the hash value of the set E_v of elements of $\mathcal{X}_T$ that correspond to L_v; let $\mathcal{Y}_{v3} = \{(\mathsf{hash}, h(\mathsf{sl}(e_1), \ldots, \mathsf{sl}(e_{|E_v|})\}$ (alternatively, this hash can be computed as the accumulation of values $\mathsf{sl}(e_i)$ using a BM accumulator). Then the *node accumulation* for node $v \in \mathcal{X}_L$ is defined as $\mathsf{acc}(\mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3})$.

Construction overview. Here we discuss the operation of the algorithms of our scheme. A more detailed description can be found at the full version of our paper. The genkey algorithm is exactly the same as the one for our main construction (plus generating a collision resistant hash function if sl_1 is chosen for subtree labels). For setup, the owner first builds $\mathcal{X}_L$, the trie containing all distinct paths appearing in the $\mathcal{X}_T$. He then computes subtree labels $\mathsf{sl}(e)$ for each element $e \in \mathcal{X}_T$ in a bottom up way starting from the leaves and node accumulations acc_v for each node $v \in \mathcal{X}_L$. He also computes for v, two structure accumulations t_v, s_v, the first of which contains only information regarding the labels of its children nodes and the second contains all the node information except for the label path L_v. Moreover, he computes a subset witness for each consecutive pair of children of v (ordered alphabetically based on their label), *exactly* as in the main scheme. He finally builds a single accumulation tree over the set $\mathcal{V}$ of node accumulations acc_v and sends all components to the server.

With respect to query we again distinguish the two cases. If the queried path L does not appear in $\mathcal{X}_T$, proof generation is identical to the mismatch case of our main construction. The server simply needs to prove the existence of a prefix of L, and that none of the children of the node v in $\mathcal{X}_L$ corresponding to this prefix has the necessary next label. This is achieved by providing the length of the prefix, the corresponding acc_v (with its accumulation proof), the structure accumulation s_v, and the corresponding pair of children labels with its subset witness. The client, first checks the validity of acc_v, then verifies it corresponds to the given prefix of L using the structure accumulation and, finally checks whether the next label in L is covered by the given label pair, as well as the fact that it is a well-formed pair using the given witness. Observe that, in contrast to our main construction, since $\mathcal{X}_L$ is uncompressed, the mismatch will always happen "at the end" of a node.

If L appears in $\mathcal{X}_T$, the answer consists of all elements e_i in the document that have label paths corresponding to L as well as the subtrees of $\mathcal{X}_T$ that have e_i as roots. Note that, since the result consists of a forest of subtrees, their structure (i.e., the parent-children relations of elements) is also explicitly part of the answer. Proof generation proceeds as follows. If v is the node in $\mathcal{X}_L$ that corresponds to L, the server only needs to provide acc_v (with its accumulation proof) and the structure accumulation t_v. The client first validates that acc_v is a correct node accumulation and then checks that it corresponds to L and all provided elements e_i using the structure accumulation t_v. To achieve the latter, he first computes subset label $\mathsf{sl}(e_i)$ for each element in the answer E_v and their hash value $\eta = h(\mathsf{sl}(e_1), \ldots, \mathsf{sl}(e_{|E_v|}))$. He then computes $g^{\mathbf{x}}$ for $\mathbf{x} = (s + \mathbf{r}(\mathsf{hash}, \eta)) \prod_{i=1}^{|L_v|} (s + \mathbf{r}(\mathsf{label}, i, l_i))$ and finally checks whether $e(g^{\mathbf{x}}, t_v) = e(\mathsf{acc}_v, g)$. This simultaneously validates that v corresponds to L and that *all* elements of $\mathcal{X}_T$ (including subtrees) have been returned. For the latter, observe that sl is a secure cryptographic representation, hence no elements may be omitted.

Parallel setup. Our XML construction also supports parallelizable setup, in $O(\log n)$ parallel time using $O(n/\log n)$ processors.
Label paths. For a node v of a rooted tree T of size n, let x_v denote the information stored at v and $\mathsf{path}(v)$ denote the path between v and the root. Let prefix accumulations of tree T, be computed as $\mathsf{accP}_v(T) = g^{\prod_{u \in \mathsf{path}(v)} (s+x_u)}$ for $v \in T$. These can be computed in $O(\log n)$ parallel time, by computing a *suffix accumulation over the Euler tour of* T (using our approach for text pattern matching), that is appropriately refined to accumulate $(s+x_v)$ modulo p in the exponent when the tour encounters the left side of v and $(s+x_v)^{-1}$ modulo p when the tour encounters the right side of v.
Subtree labels. If the labels are accumulation-based, they can be modeled as $\mathsf{accS}_v(T) = g^{\prod_{w \in \mathsf{subtree}(v)} (s+x_w)}$ for $v \in T$, where $\mathsf{subtree}(v)$ is the set of nodes contained in the subtree rooted on node v. Note that in order to compute $\mathsf{accS}_v(T)$ for all $v \in T$, it suffices to compute the products $\prod_{w \in \mathsf{subtree}(v)} (s + x_w)$ for all $v \in T$. Such a parallel algorithm running in $O(\log n)$ parallel time was originally presented as an application of tree contraction [25].

5.3 Dynamic datasets

So far we have only dealt with the case of static datasets, where the data owner outsources the data once, with no further changes. However, in many cases the owner may wish to update the dataset by inserting or removing data. When this occurs, the owner can of course run the entire setup process again, but here we investigate more efficient updates for the two applications presented above.

Collection of text documents. For our scheme we build a single suffix tree on the collection, hence our update efficiency will crucially depend on this data structure's behavior. In practice, a single modification in any of the documents may change the suffix tree entirely and the best we can do for updates is to re-run setup, in time $O(n\tau)$. One way to accommodate updates more efficiently is the following. We first split the documents in $\sqrt{\tau}$ groups, each with $\sqrt{\tau}$ documents, and then run our scheme separately for each group. A given query now decomposes into a separate query for each group. In this setting, an update –in the form of a document insertion or removal– will only cause the re-computation of one of the suffix trees (and the corresponding ADS) in time $O(n\sqrt{\tau})$ instead of $O(n\tau)$. On the other hand, this increases the cost for proof generation/verification and size by a multiplicative $\sqrt{\tau}$ factor, but in settings with frequent updates, this trade-off may be favorable.

XML documents. In this setting, we discuss updates in the form of element insertion or removal from the document, that do not change the structure of the label trie $\mathcal{X}_L$ (i.e. they do not introduce a new label path in the document). Otherwise, we face the same difficulties as in the previous application. We focus on leaf element insertions; in order to insert more than one element (building a new subtree in $\mathcal{X}_T$) the process is repeated accordingly. Updates of this form can be efficiently handled as follows: First, the new element's subtree label is computed and the subtree labels of all its ancestors in $\mathcal{X}_T$ are re-computed. Second, the node accumulation value of the corresponding node $v \in \mathcal{X}_L$ is updated by inserting the subtree label of the new element in the *XML elements hash*. Then, the second structure accumulation and children witnesses for v are updated, and the accumulation tree is updated accordingly.

Let us now calculate the efficiency of the above process. Computing the subtree labels takes $O(d)$ operations and recomputing the node and structure accumulations and children witnesses requires $O(|C_v|)$ exponentiations (assuming the *XML elements hash* is computed with a BM accumulator). We stress that $|C_v|$ is the number of distinct labels the siblings of the inserted element have, and not the number of its XML element siblings; for all practical purposes $|C_v|$ can be viewed as a constant. Finally, by the properties of the accumulation tree, the last step can be run in time $O(|\mathcal{X}_L|^{\epsilon})$, where $\epsilon \in (0, 1]$ is a chosen parameter. The same holds for the case of element removal. Hence the overall update cost is $O(d + |C_v| + |\mathcal{X}_L|^{\epsilon})$, which is much less than the setup cost.

6. PERFORMANCE EVALUATION

In this section, we present an experimental evaluation of our two authenticated pattern matching applications from Section 5. All scheme components were written in C++, by building on a core BM accumulator implementation [37] developed by Edward Tremel, as well as using library DCLXVI [2] for bilinear pairings, library FLINT [3] for modular arithmetic, Crypto++ [1] for implementing SHA-2, and the pugiXML [4] XML parser. The code was compiled using g++ version 4.7.3 in C++11 mode. Our goal is to measure important quantities related to the execution of our scheme: verification time for the clients, proof generation time for the server, setup time for the data owner, and the size of the produced proof.

Experimental setup. For our collection of documents application, we used the Enron e-mail dataset [20] to build collections of e-mail documents (including headers) with total size varying between 10,000 and 1,000,000 characters. We set the public key size to be equal to 10% of the text size at all times (this can be seen as an upper bound on the size of patterns that can be verified). For the exact path XML application, we experimented with five XML documents of various sizes from the University of Washington XML repository [5], as well as a large synthetic XML document generated using the XMark benchmark tool [6]. A list of the documents and their sizes can be found in Table 2. Special characters were

XML document	size (MB)	# of elements	# of paths	setup (sec)
SIGMOD	0.5	11,526	11	0.4
Mondial	1	22,423	33	0.7
NASA	23	476,646	95	8.9
XMark	100	2,840,047	514	35.2
DBLP	127	3,332,130	125	68.9
Protein sequence	683	21,305,818	85	381.5

Table 2: XML documents used for experiments and setup time.

text size	setup (sec)
100	1.4
1,000	10.7
10,000	99.6
100,000	976.5
1,000,000	10,455

(a) setup time

proof type	proof (KB)	optimal (B)
positive	3.4	435
negative	3.4	435
neg. end node	4	500
xml positive	1.2	178
xml negative	1.7	243

(b) proof size

Table 3: Setup cost for text documents and size of proofs.

escaped both in the e-mail documents and the text content within XML elements. For computing subset labels and XML elements hashes within trie nodes, we used the hash-based approach with SHA-2. In both cases we constructed accumulation trees of height 1 for the authentication of suffix and node accumulations. All quantities were measured ten times and the average is reported.

Working with pairings over elliptic curves. As mentioned in Section 2 the BM accumulator employs a pairing e defined over two bilinear groups. For simplicity of presentation, we previously defined $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$, i.e., both its inputs come from the same group (known in the literature as an *symmetric* pairing). In practice however, *asymmetric* pairings of the form $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$ are groups of the same prime order but $\mathbb{G}_1 \neq \mathbb{G}_2$, are significantly faster. The DCLXVI library we use here, makes use of such a pairing over an elliptic curve of 256 bits, and offering bit-level security of 128 bits (corresponding to the strong level of 3072-bit RSA signatures according to NIST [7]). Elements of $\mathbb{G}_2$ (corresponding to *witnesses* in our scheme) are defined over an extension of the field corresponding to elements of $\mathbb{G}_1$(resp. *accumulations*). The former are twice as large as the latter and arithmetic operations in $\mathbb{G}_2$ are roughly 2-3 times slower.

Setup cost. Table 3(a) shows the setup time versus the total length of the documents and depicts a strong linear relation between them. This is expected because of the suffix accumulation computations and the fact that the suffix tree has linearly many nodes. The practical cost is quite large (e.g., roughly 3 hours for a text of 1,000,000 characters). However, this operation only occurs once when the outsourcing takes place. For the XML case, Table 2 contains the necessary setup time for the documents we tested. The time grows with the size of the document but is quite small in practice, even for very large documents (e.g., a little above 6 minutes for a document of size 683MB). This happens because the crucial quantity is the number of distinct paths in the document (that will form the nodes of $\mathcal{X}_L$), and not the number of elements in the document itself.

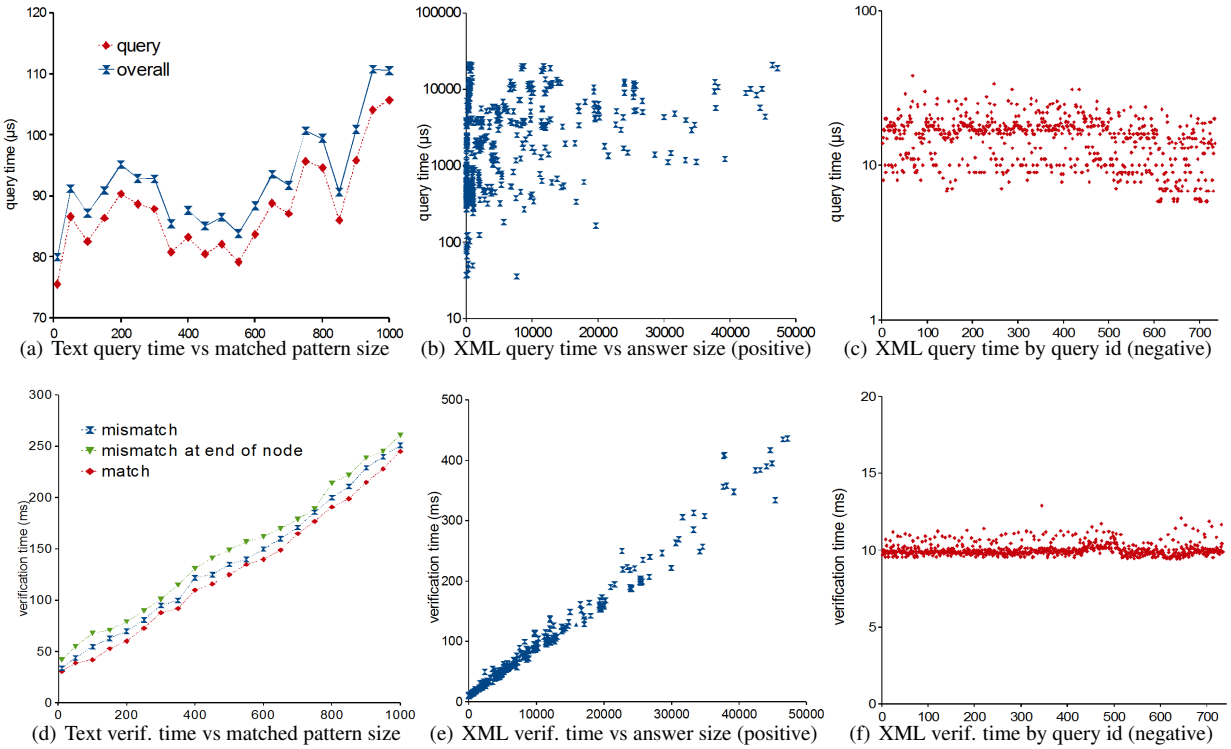
Query time. Figures 4(a), (b) and (c) show the server's overhead for answer computation and proof generation, for text and XML pattern matching. For text pattern matching we experimented with pattern lengths of 10 to 1,000 characters at a text of 1,000,000 characters. To test the query time at the server, we focused on queries with negative answers and prefix matches finishing at the end of a node, which is the most demanding scenario (e.g., a pattern that starts with a letter that does not even appear in the text is answered by simply looking at the children of the root node of the suffix tree). To produce such queries, we identified matches at ends of various nodes, and "built" progressively larger patterns that ended with them. We plot the overall time for query evaluation and proof generation versus the size of the found prefix. As can be seen, the cost is in the order of a few microseconds (μs) at all times. In the case of XML queries, we present findings both for the positive and negative case in Figures 4(b) and (c) respectively, for the NASA, XMark and DBLP datasets (note the different y-axis scales). For queries with positive answers, we tested on all existing label paths, whereas for negative ones we inserted a "junk" label at a random point along a valid path. In the first case the plot is versus the size of the answer; for the second case where the answer size is zero, we plotted the times across the x-axis by simply assigning an arbitrary id (1-742) to each query. The overhead is again very low, less than 1 millisecond for most instances in the positive case and less than 20μs in the negative. This discrepancy occurs because the server must compile the answer subtrees into a new pugiXML document (that will be sent to the client) for a positive answer –which does not entail any cryptographic operations. Finally, in both applications the plots are quite noisy. This follows because the answer computation time varies greatly with the topology of the trees (in both cases) and the size of node contents (for the XML case).

Comparison with query-evaluation time. In both cases the server's overhead for proof generation is very low in our scheme since, once the answer is computed, he simply performs a constant number of lookups in his local database to find the corresponding accumulations and witnesses. This is highlighted in Figure 4(a) where the lower data series corresponds to the time it takes to simply evaluate the query (without any proof of integrity). As can be inferred, the pure cost for proof generation is less than 10μs at all times. This is also true for the XML case, but due to the different plot type, it was not easy to depict in a figure. In essence, in our scheme the server only performs exactly the same operations as if there was no authentication plus a constant number of memory look-ups, for both applications which makes it ideal for scenarios where a dedicated server needs to handle great workload at line-speed.

Verification time. In Figures 4(d),(e) and (f) we demonstrate the verification cost for clients for the text and XML pattern matching applications. In the first case the time is measured as a function of the queried pattern length (or matching prefix in the case of a negative answer) and in the second as a function of the answer size (as before, for negative responses we plot versus an arbitrary id).

To test the verification time for our text application, we report findings for all three possible cases (match, mismatch and mismatch at end of node). We observe a strong linear correlation between the verification time and the length of the matched pattern. This follows because the main component of the verification algorithm is computing the term $g^{\mathbf{z}}$. Observe that verification for the positive case of a match is slightly faster, which corresponds to our protocol description. In that case, the client needs to perform operations over accumulations and witnesses related only to suffixes, without getting involved with suffix tree nodes. On the other hand, the case where a mismatch occurs at the end of a suffix tree node is slightly more costly than that of a simple mismatch since the client needs to also verify a received sequel with a corresponding witness. The verification overhead remains below 300ms even for arguably large pattern sizes consisting of up to 1,000 characters.

For XML path matching, we report findings for answer sizes of up to 50,000 elements. Observe again the strong linear correlation between the answer size and the verification time, for positive answers. This follows from the fact that the client performs one hash operation per element in the answer, followed by a constant number of bilinear pairings. The total overhead is very small, less than half a second even for large answer sizes. If the answer is negative



(a) Text query time vs matched pattern size (b) XML query time vs answer size (positive) (c) XML query time by query id (negative)

(d) Text verif. time vs matched pattern size (e) XML verif. time vs answer size (positive) (f) XML verif. time by query id (negative)

Figure 4: Query (top) and verification (bottom) time for text and XML pattern matching.

(again, note the different y-axis scale) the overhead comes mostly from the fixed number of pairings and is much smaller.

Proof size and optimizations. With the DCLXVI library, bilinear group elements are represented by their Jacobian coordinates, i.e., three values per element. As described in [26], each coordinate of an element in $\mathbb{G}_1$ is represented by a number of double-precision floating-point variables. The total representation size is 2304 bits for elements of $\mathbb{G}_1$ and 4608 bits for elements of $\mathbb{G}_2$. In our scheme, proofs also contain additional structural information (e.g., position of match/mismatch in text, depth of edge, etc.) which was less than 50 bytes for all tested configurations.

Table 3(b) contains the proof sizes produced by our scheme for both applications. Recall that these numbers are independent of dataset, pattern, or answer size. At all times the proof size is below 4Kb and as low as 1.2Kb for positive XML proofs. While these sizes are very attractive for most applications, further improvements (not implemented here) are possible. Elements can be instead represented by their two affine coordinates (x, y). Moreover, there is no need to transmit y-coordinates as all elements lie on the curve with equation $y^2 = x^3 + 3$, which is part of the public parameters of the scheme. Given x, the y-coordinate can be inferred by a single bit indicating which square root of $x^3 + 3$ it corresponds to. The result of these optimizations can be seen on the third column of the table. The proof size is as low as 435 bytes for text pattern matching and 178 bytes for XML path search. On the other hand, these techniques introduce a small additional overhead at the client (for computing y and transforming to Jacobian coordinates again). When reduced communication bandwidth is essential or proof caching occurs, this extra cost may be acceptable.

Discussion and comparison with alternative schemes. The above results highlight the practicality of our constructions. In particular for the server, who would have to handle the largest workload, the fact that all proof components are pre-computed implies only a small fixed overhead between simply evaluating a query and authenticating the answer with a proof on top of that. Verification time is also appealing for most real-world scenarios making our scheme ideal for settings with "thin" clients or even mobile devices. One component of our scheme that can be improved significantly is the one-time setup operation; pre-computing all proof components takes its toll, especially for the text pattern matching application. Finally, while proofs are arguably very short, they can be further compressed by the optimization discussed above.

To the best of our knowledge, the only other known constructions to achieve constant-size proofs rely on general verifiable computation schemes. As discussed previously, state-of-the-art implementations fall under two categories: circuit or RAM-based. For the former, (e.g., [32]) the proof generation cost is always at least as large as parsing a circuit that has the entire document as input. The latter are asymptotically better than the former, but still incur prohibitive costs for the server. In particular, as shown in [42], performing a BFS over a graph of roughly 9,000 edges takes 270 hours with [13] and 50 hours with [8] for proof generation. For comparison, in our text pattern matching experiment, we tested patterns of up to 1,000 elements and an alphabet of 256 characters. Assuming a binary search tree at each node for finding children nodes matching the pattern, this corresponds to 8,000 memory reads in the worst case, and proof generation took less than 10μs. A different line of work for authenticated pattern matching is based entirely on cryptographic hashes (e.g, [16, 23, 10]). There is no existing built system for concrete comparison but, due to the different nature of operations, we expect these schemes to have faster setup and slightly better verification time than ours. However, the proofs grow with the pattern size for text pattern matching, and with the size of the *entire* document (in the worst case) for XML queries.

7. CONCLUSION

We presented a novel approach for verifying pattern matching queries on text and XML documents that yields constant-size proofs, using careful encoding of answer-specific certification relations with cryptographic accumulators. We demonstrated the practicality of our schemes by experimenting on real datasets. In this work we focused on exact pattern matching, leaving for future work the authentication of more general related query types, such as patterns expressed by regular expressions or pattern matching on graphs.

Acknowledgments

We thank all the anonymous reviewers for their detailed comments and suggestions. We also thank Edward Tremel for many insightful discussions in early stages of our work and for making his BM accumulator code [37] available to us, portions of which we used as a library. Research supported in part by the U.S. National Science Foundation under CNS grants 1012798, 1012910, and 1228485.

8. REFERENCES

[1] Crypto++ Library. `http://www.cryptopp.com/`.

[2] DCLXVI Library. `http://cryptojedi.org/`.

[3] FLINT Library. `http://www.flintlib.org/`.

[4] PugiXML. `http://pugixml.org/`.

[5] University of Washington XML data repository. `http://www.cs.washington.edu/research/xmldatasets/`.

[6] XMark. `http://www.xml-benchmark.org/`.

[7] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST recommendation for key management Part 1: General (revision 3), July 2012.

[8] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[9] J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sginatures (extended abstract). In *EUROCRYPT*, 1993.

[10] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *IEEE TKDE*, 16(10):1263–1278, 2004.

[11] D. Boneh and X. Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[12] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *RAID*, 2014.

[13] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, 2013.

[14] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, 2002.

[15] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, 2014.

[16] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. *Journal of Computer Security*, 6:841–864, 2004.

[17] S. Faust, C. Hazay, and D. Venturi. Outsourced pattern matching. In *ICALP*, 2013.

[18] M. T. Goodrich and R. Tamassia. *Algorithm design - foundations, analysis and internet examples*. Wiley, 2002.

[19] J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

[20] B. Klimt and Y. Yang. The Enron corpus: A new dataset for email classification research. In *ECML*, 2004.

[21] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM TISSEC*, 13(4):32, 2010.

[22] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. ERA: Efficient serial and parallel suffix tree construction for very long strings. *PVLDB*, 5(1):49–60, 2011.

[23] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[24] R. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[25] G. L. Miller and J. H. Reif. Parallel tree contraction, part 2: Further applications. *SICOMP*, 20(6):1128–1147, 1991.

[26] M. Naehrig, R. Niederhagen, and P. Schwabe. New software speed records for cryptographic pairings. In *LATINCRYPT*, 2010.

[27] L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, 2005.

[28] R. Ostrovsky, C. Rackoff, and A. Smith. Efficient consistency proofs for generalized queries on a committed database. In *ICALP*, 2004.

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

[30] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, 2011.

[31] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, pages 1–49, 2015.

[32] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symp. on Security and Privacy*, 2013.

[33] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, 2013.

[34] R. Tamassia. Authenticated data structures. In *ESA*, 2003.

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

[36] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud*, 2012.

[37] E. Tremel. Real-world performance of cryptographic accumulators. Undergraduate Honors Thesis, Brown University, 2013.

[38] P. Weiner. Linear pattern matching algorithms. In *IEEE SWAT*, 1973.

[39] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD*, 2009.

[40] A. A. Yavuz, P. Ning, and M. K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *FC*, 2012.

[41] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. K. Robertson, A. Juels, and E. Kirda. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *ACSAC*, 2013.

[42] Y. Zhang, C. Papamanthou, and J. Katz. ALITHEIA: Towards practical verifiable graph processing. In *CCS*, 2014.