Efficient Authenticated Multi-Pattern Matching

Zhe Zhou[1], Tao Zhang[1], Sherman S.M. Chow[1], Yupeng Zhang[2], Kehuan Zhang[1]
[1]Department of Information Engineering, The Chinese University of Hong Kong
[2]Department of Electrical and Computer Engineering, University of Maryland, College Park
[1]{zz113, zt112, sherman, khzhang}@ie.cuhk.edu.hk, [2]zhangyp@umd.edu

ABSTRACT

Multi-pattern matching compares a large set of patterns against a given query string, which has wide application in various domains such as bio-informatics and intrusion detection. This paper shows how to authenticate the classic Aho-Corasick multi-pattern matching automation, without requiring the verifier to store the whole pattern set, nor downloading a proof for every single matching step. The storage complexity for the authentication metadata at the server side is the same as that of the unauthenticated version. The communication overhead is minimal since the proof size is linear in the query length and does not grow with the sizes of query result nor the pattern set. Our evaluation has shown that the query and verification times are practical.

Keywords

Authenticated Data Structure; Verifiable Computation; Pattern Matching

1. INTRODUCTION

Cloud services emancipate organizations and individuals from maintaining a large amount of storage and conducting resource-consuming computations, as they can elastically rent storage and computation capability to satisfy their dynamically fluctuating demand. As paying customers, they may not fully trust the cloud service provider, and it is crucial to have a mechanism to verify the authenticity of the cloud computation. Ideally, the outsourcing mechanism should allow a *data owner* to outsource the data, possibly with some additional meta-data for authentication, to the *cloud server* for answering the queries from *clients* with proofs, which the clients can verify the proof to assert their correctness, without interacting with the data owner nor downloading a large set of the original data.

In this paper, we consider the *authenticated multi-pattern matching* problem — Given a *pattern set* $\mathcal{P}$ which contains a large amount of patterns formed by alphabets Σ, and some queries which are *texts* from Σ, it reports all the occurrences of patterns in each of the given query, as well as the corresponding proof which can be used to authenticate the correctness of the answers.

Multi-pattern matching query has a lot of deployed applications such as anti-virus scanning, bio-informatics, business analytics, intrusion detection, natural language processing, web search engines, *etc.* Take bio-informatics as an example, when a doctor wants to know if a patient has genetic disease, he can query to the disease-causing gene pool for DNA pieces collected from the patient.

Real world multi-pattern matching applications occupies a lot of storage resulted from the accommodation of the huge pattern set (simply consider pools for human gene or virus). It is a perfect example for leveraging the cloud for outsourcing the patterns storage and decreasing the query complexity as much as possible, ideally, making it independent of the size of the pattern set. Yet, since the results are not computed by the user themselves, the users may have doubt in the trustworthiness of the cloud service, unless there are cryptographic proofs provided which assert the correctness.

In this work, we present the *first* solution of efficient authenticated multi-pattern matching, which is based on Aho-Corasick algorithm (AC automaton) [5], to be described in Section 2.2. We first propose a scheme for small alphabet size, and then a tailor-made one for large alphabet size. Our schemes have the following desirable features:

- *Short proof*: The proof size is only proportional to the size of the query n. It is independent of both the pattern set size and the result size m.
- *Low query overhead*: The proof generation time complexity is $O(n + m \log m \log n)$, where n is the query string length and m is the number of matched patterns. There is only a logarithmic overhead compared to that of the original AC-automaton without authentication, which is $O(n + m)$. It is also independent of the pattern set size.
- *Efficient verification*: The verification time is $O(n + m \log m)$, which is efficient, as the client at least needs to read the query and the answer which already add up to $O(n + m)$. Again, it is independent of the pattern set size.

1.1 Scheme Overview

Our schemes are based on AC automaton built on trie. In the original AC automaton, a path is explored according to the query string. For every node lying on the path, the

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

ISBN 978-1-4503-4233-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897906

Table 1: Complexity comparison with generic solutions in the context of pattern matching: N is the total number of patterns outsourced, $\bar{l}$ is the average length of these patterns, m is the number of matched patterns, n is the length of queried text.

	Space	Setup	Proof Size	Query Time	Verification Time	Assumption
Naïve Signature Scheme	$N\bar{l}$	$N\bar{l}$	$N\bar{l}$	$N\bar{l}$	$N\bar{l}$	Digital Signature
Generic VC	$N\bar{l}$	$N\bar{l}$	1	$N \log N$	n	Non-falsifiable Assumptions
Generic ADS	$N\bar{l}$	$N\bar{l}$	$(n+m)\log N$	$(n+m)\log N$	$(n+m)\log N$	Collision Resistant Hash
Our Schemes	$N\bar{l}$	$N\bar{l}$	n	$n + m \log m \log n$	$n + m \log m$	q-Strong Diffie-Hellman

automaton outputs a set of matching patterns. The complete result pattern set is the multi-set union of those sets for every node.

To enable users to verify the result, our schemes attach a cryptographic accumulator as well as a signature to every node in the trie during the setup. With the help of the signatures, the node can be authenticated and the path can be verified thereby. An accumulator stores all the patterns which are the outputs for a node. So, users can verify if the accumulation of all the accumulators on a path matches with the result.

For better performance and shorter proof, we used an accumulation tree structure to enable users to verify the union relationship.

1.2 Related Works

Verifiable Computation (VC). General-purpose techniques for verifiable computation [19, 9, 10, 20, 11, 15] and their implementations [30, 29] are potential solutions to the problem. To use these systems, the computation of the multi-pattern matching is either modeled as a circuit [28, 16] or as a RAM program [14, 31, 6, 7, 15]. In circuit-based VC, the whole pattern set must be hardcoded into the circuit, which makes the computation on the server side inefficient (quasi-linear in the circuit size). RAM-based VC partially solves this problem, but their practicality is still questionable. Besides, the security of these systems is often based on non-standard assumptions or non-falsifiable assumptions.

Authenticated Data Structure (ADS). Naor and Nissim [25] proposed a generic solution to construct an ADS for arbitrary data structure. Martel *et al.* [23] improves the performance for tree-based data structures. The idea is to build a Merkle tree [24] on top of the whole data structure, and provide a proof for each step of the query process. The query time and the proof size are thus both quasi-linear in the number of steps to generate the result, which is inefficient in the multi-pattern matching case. In particular, the query time, proof size, and verification time are all $O((n+m)\log N)$, where n is the queried string length, m is the number of matching patterns, and N is the total number of patterns. Note that m could be as much as n^2 in the worst case. In contrast, the proof size of our schemes is $O(n)$.

ADS for Pattern Matching. Papadopoulos *et al.* [27] proposed an authenticated single pattern matching scheme with optimal proof size by employing cryptographic accumulators [26, 4]. Faust *et al.* proposed a verifiable pattern matching scheme [18] which further achieves query privacy, yet it requires the data owner's online presence. XML file search is a typical pattern matching application. Existing schemes [17, 8] support single pattern matching on XML file, where each query shows whether the *queried pattern* occurs in the XML file. All these schemes only support single pattern matching, which is essentially a Applying these schemes to the multi-pattern matching problem gives even worse complexity than the naïve approach of signing all the patterns and returning them with the corresponding signatures. different problem. In their model, a long string is outsourced to the server, and a client queries whether a particular pattern occurs in the string. While in our case, a large set of patterns are outsourced and a client queries what patterns occur in the queried string. To the best of our knowledge, we are the first to consider this problem.

A detailed comparison is presented in Table 1.

2. PRELIMINARIES

This section reviews the pattern matching algorithm and some cryptographic primitives used in our proposed system.

2.1 Trie

Trie is a data structure to store a large amount of strings formed by alphabet Σ, also named as a prefix tree. It is a $|\Sigma|$-ary tree where every string is represented as a path from the root to a node, and each node represents a common prefix of some strings. Root node represents a null string while other node represents a prefix that is constructed by appending the character represented by the incoming edge, to the prefix that its parent node represents.

Figure 1 shows a trie of the set {"adv","d","re","read","rec"}. Prefixes which are complete strings are marked "gray", *e.g.*, Node 6 is in gray since the prefix "re" it represents is in the string set. We remark that dashed and dotted edges shown in Figure 1 do not belong to the trie, and will be used by the AC automaton (to be explained later).

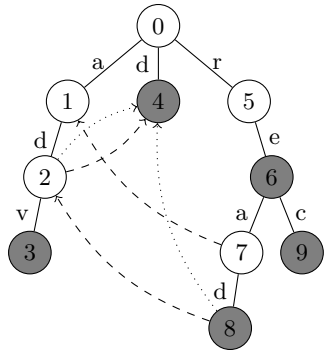


Figure 1: Illustration of trie and AC automaton of {"adv","d","re","read","rec"}: gray node denotes an ending of a pattern, dotted link represents *suffixP* pointer, and dashed link represents *fail* pointer. Pointers to the root are omitted.

To set up a trie $\mathcal{T}$, strings should be added to the trie

Table 2: Matching Flow for "breadv"

Step	Prefix	Longest Suffix	Movement	Action
1	"b"	""	$0 \to 0$	The root has no edge for 'b', so stay at the root.
2	"br"	"r"	$0 \to 5$	The root has an edge for 'r', so move along the link to Node 5.
3	"bre"	"re"	$5 \to 6$	Node 5 has edge 'e', move to Node 6. Node 6 was marked as pattern ending, so output a match "**re**" here.
4	"brea"	"rea"	$6 \to 7$	Node 6 has edge 'a', move to Node 7.
5	"bread"	"read"	$7 \to 8$	Node 7 has edge 'd', move to Node 8. Node 8 is pattern ending, output "**read**". Node 8 has *suffixP*, traverse along the pointer, and include "**d**" as a matched output.
6	"breadv"	"adv"	$8 \to 3$	Node 8 has no edge 'v', use *fail* to traverse to Node 2. Node 2 has edge 'v', move along the edge to Node 3, output "**adv**" as it is an ending.

one by one. Specifically, for each string, traverse $\mathcal{T}$ from the root according to the character sequence of the string to be inserted, until the current node does not have an outgoing edge representing the current character for continuing the walk, which is the node for adding the rest of the string. Specifically, we add a new edge representing the missing character and direct it to a new node, and then continue the rest of the walk. After finishing walking, the current node should be at the end of the path representing the new string, and is marked as the ending of a string. The node identifier (ID) is set to be the insertion timestamp, as illustrated in Figure 1.

Trie has a lot of good properties enabling it to be a good data structure for storing strings. Firstly, all the common prefixes in the string set are stored only once. So, it can achieve compression to some extent. Even in the worst case, it requires less space than simply storing all the strings. It can be further compressed by aggregating nodes in the intermediate path without branches. Besides, trie enables searching strings with prefix quickly. Many algorithms employ trie as a base data structure for both compression and fast searching, and AC automaton is such an algorithm [5].

2.2 Aho-Corasick String Matching Algorithm

A simple (and naïve) multi-pattern matching algorithm works as follows. Given a query text, enumerate all the patterns and check if the pattern occurs by checking if the pattern is a prefix of any suffix of the text. For the last step, advanced pattern matching algorithm like KMP [22] or Boyer-Moore string searching algorithm [13] can be used. Yet, the pattern enumeration remains. For a huge number of patterns, the query text probably matches only a small portion of them, but enumeration means all of them are accessed at least once, which is a waste.

Aho-Corasick string matching algorithm (AC automaton) does not need to access all the patterns while correctness can still be guaranteed. The searching philosophy is to traverse all the prefixes of the query text from shorter to longer (e.g., "b", "br", $\cdots$, "bread", "breadv"), and for each prefix, find all its suffixes which are in the pattern set (e.g., "read" and "d" for "bread"). Correctness comes from the fact that all suffixes have been conceptually covered. Now we can divide our problem into two sub-problems, namely, given a prefix of the query, how to do faster than trying all suffixes, and how to output those suffixes.

Matching Flow. According to the query string, the automaton traverses the trie starting from the root. It sequentially takes one character from the query string each time and tries to move from the present node to another node according to the character. If the present node has an edge for the character, it moves along the edge to the lower node.

If however the present node has no such edge, it iteratively uses the *fail* pointers to try to find a node that has an eligible edge and then move along that edge. Along traversing the *fail* pointers, if the root node is reached and it still cannot find an eligible edge, the automaton just stays at the root. This part will be elaborated in the paragraph for "Efficient Transition".

Every time the automaton moves to a new node according to the rules, it tries to output the matched patterns if possible. If the current node is an ending of a pattern, the pattern must be output. Besides, if the *suffixP* pointer points to a node rather than the root, the automaton iteratively outputs nodes using *suffixP*. This part will be elaborated in the paragraph for "Efficient Output".

For example, if the pattern set is what showed in Figure 1 and the query string is "breadv", the automaton starts from the root and follows the steps showed in Table 2.

Efficient Output. The *suffixP* pointer of a node guarantees that it is pointed to the longest suffix (except itself) of the node which is in the pattern set. For the i-th long prefix of query text q_i, if the longest suffix of q_i on the trie (may not be a pattern ending) is represented by node v, we can efficiently output all the matches for q_i by backtracing from v along *suffixP* pointer until the root is reached. (See Step 5 in Table 2 as an example.)

Backtracing the path does not miss any result and guarantees that all the suffixes for the node were accessed. This step can be regarded as a sub-routine$\mathsf{TracePatterns}(v)$, which outputs all the hit patterns when node v is accessed. Our proposed scheme will use this sub-routine.

Efficient Transition. The AC automation introduces a *fail* pointer to realize fast transition from a longer suffix to a shorter one (may not be a pattern). Specifically, suppose for the i-th long prefix q_i, node v in the trie represents its longest eligible suffix, but we failed to find the longest eligible suffix for q_{i+1} by moving along any outgoing edge from node v. We can still use the *fail* pointers to quickly locate the node for a shorter suffix, which may lead to the longest eligible suffix for q_{i+1}. (See Step 6 in Table 2 as an example.)

Efficiency. The AC automaton is very efficient. First, note that the additional links are just two pointers for each node which do not increase the space complexity of the trie. For setup, the time complexity is $O(N\bar{l})$, including constructing the trie, the *fail* pointers as well as the *suffixP* pointers for each node of the trie. For searching, $O(n+m)$ nodes are accessed where n nodes are accessed for stepping down, and the nodes accessed for outputting result (backtracing) is $O(m)$.

2.3 Bilinear Pairings

Bilinear pairing is a powerful cryptographic primitive which

maps a pair of base group elements to a target group element. Let $\mathbb{G}$ and $\mathbb{G}_T$ be two cyclic groups of prime order p. A (type-I) bilinear pairing $e(\cdot,\cdot) : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ satisfies:

- *Bilinearity.* For all $u, v \in \mathbb{G}$, $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$.
- *Non-degeneracy.* $e(g, g) \neq 1$ where g is a generator of $\mathbb{G}$.
- *Efficiency.* For all $u, v \in \mathbb{G}$, $e(u, v)$ is efficiently computable, but the inverse is not.

We also say $(\mathbb{G}, \mathbb{G}_T, e, p, g)$ defines a bilinear group.

2.4 q-Strong Diffie-Hellman Assumption

The q-Strong Diffie-Hellman (q-SDH) assumption was first introduced by Boneh and Boyen [12]. The q-SDH problem states that: given a bilinear group $(\mathbb{G}, \mathbb{G}_T, e, p, g)$, and a $(q+1)$-tuple $(g, g^x, g^{x^2}, \cdots, g^{x^q})$, output a pair $(c, g^{\frac{1}{x+c}})$. An adversary $\mathcal{A}$ has advantage ϵ in solving the q-SDH problem in $\mathbb{G}$ if

$$\Pr[\mathcal{A}(g, g^x, g^{x^2}, \cdots, g^{x^q}) = (c, g^{\frac{1}{x+c}})] \geq \epsilon.$$

We say that the q-SDH assumption holds for group $\mathbb{G}$ if no probabilistic polynomial-time (PPT) adversary has non-negligible advantage in solving the q-SDH problem in $\mathbb{G}$.

2.5 Dynamic Accumulator

A dynamic accumulator securely represents a set of elements in $\mathbb{Z}_p$ in a compact manner. A dynamic accumulator has the following properties.

- *Efficiency.* The calculation of an accumulator and the evaluation of membership relations or subset relations are efficient.
- *Commutativity.* The accumulation order of the set members does not affect the result.
- *Collision-Resistance.* No PPT adversary can output a set A of polynomial size, an element a' which is not in the set A, and a corresponding witness wit which will pass the verification algorithm for proving that a' is in A.

A dynamic accumulator scheme $\mathcal{ACC}$ is a collection of five polynomial-time algorithms $\mathcal{ACC}$ = (Setup, Accu, Update, CompuWit, AccVerify). Here, we review Nguyen's pairing-based construction [26, 4], which is collision-resistant under the q-SDH assumption.

$\mathsf{Setup}(1^\lambda) \to (\mathsf{param}, \mathsf{aux})$.

1. Choose a type-I bilinear group $(\mathbb{G}, \mathbb{G}_T, e, p, G)$, where the q-SDH assumption holds for $\mathbb{G}$, and G is a random generator of $\mathbb{G}$.
2. Randomly choose $s \xleftarrow{\$} \mathbb{Z}_p$ and compute $\zeta = (G, G^s, G^{s^2}, \cdots, G^{s^q})$.
3. Output the public parameters $\mathsf{param} = (\mathbb{G}, \mathbb{G}_T, e, p, G, \zeta)$ and the auxiliary information $\mathsf{aux} = s$.

$\mathsf{Accu}(\mathsf{param}, \mathsf{aux}, A) \to \mathsf{acc}_A$. On input a member set $A = \{a_i\}_{i=1}^m$, compute the accumulator $\mathsf{acc}_A = G^{\prod_{i=1}^m (s+a_i)}$.

$\mathsf{Update}(\mathsf{param}, \mathsf{acc}, a^*) \to \mathsf{acc}^*$. On input an accumulator acc and a member a^*,

- $\mathsf{Add}(\mathsf{param}, \mathsf{acc}, a^*) \to \mathsf{acc}^*$: add new member a^*, $\mathsf{acc}^* = \mathsf{acc}^{s+a^*}$.
- $\mathsf{Delete}(\mathsf{param}, \mathsf{acc}, a^*) \to \mathsf{acc}^*$: delete member a^*, compute $\mathsf{acc}^* = \mathsf{acc}^{\frac{1}{s+a^*}}$.

$\mathsf{CompuWit}(\mathsf{param}, \mathsf{acc}_A, B \subseteq A) \to \mathsf{W}_{B \subseteq A}$. On input an accumulator acc_A and a subset B, generate a membership witness $\mathsf{W}_{B \subseteq A} = G^{\prod_{i \in A \setminus B} (s+a_i)}$ for proving the relation of $B \subseteq A$.

$\mathsf{AccVerify}(\mathsf{param}, \mathsf{acc}_A, B, \mathsf{W}) \to 0/1$. Verify the membership relation of sets A and B by evaluating the equation $e(\mathsf{W}, G^{\prod_{i \in B} (s+a_i)}) = e(\mathsf{acc}_A, G)$. This relation can be proved without revealing B via a zero-knowledge proof, which is a standard technique in the cryptography literature.

This instantiation of dynamic accumulator supports the verification of multiset union relationship described in the algorithm $\mathsf{AccVerify}_\uplus()$ below.

$\mathsf{AccVerify}_\uplus(\mathsf{param}, \mathsf{acc}_A, \mathsf{acc}_B, \mathsf{acc}_C) \to 1/0$. On input of three accumulators for the sets A, B, C respectively, verify the multiset union relation $A = B \uplus C$ by evaluating the equation $e(\mathsf{acc}_A, G) = e(\mathsf{acc}_B, \mathsf{acc}_C)$. If the equation holds, output 1. Otherwise, output 0.

Finally, given only $\zeta = (G, G^s, G^{s^2}, \cdots, G^{s^q})$ and without the auxiliary information s, acc_A for any set A of size at most q can be computed in time $O(n \log n)$ with polynomial interpolation using Fast Fourier Transform (FFT) techniques. This feature is described in the algorithm $\mathsf{Accu}_\zeta()$ below.

$\mathsf{Accu}_\zeta(\mathsf{param}, A) \to 1/0$. On input a member set $A = \{a_i\}_{i=1}^m$, compute the accumulator without the auxiliary input s using the public parameter ζ. Parse ζ as $(G_0 = G, G_1 = G^s, \cdots, G_q = G^{s^q})$ and compute $\mathsf{acc}_A = \prod_{i=0}^q G_i^{b_i}$, where $b_i = \sum_{B \subset A, |B| = q-i} \prod_{a \in B} a$.

2.6 Digital Signature Scheme

A digital signature demonstrates the authenticity of a message. A digital signature scheme $\mathcal{SIG}$ is a collection of four polynomial-time algorithms $\mathcal{SIG}$ = (Setup, KeyGen, Sign, Verify). $\mathsf{Setup}(1^\lambda) \to \mathsf{param}$ takes in a security parameter λ and generates the system parameter param which defines the message space $\mathcal{M}$. $\mathsf{KeyGen}(\mathsf{param}) \to (\mathsf{pk}, \mathsf{sk})$ takes in the system parameter and generates a verification-signing key pair $(\mathsf{pk}, \mathsf{sk})$. $\mathsf{Sign}(\mathsf{pk}, \mathsf{sk}, m) \to \sigma$ generates a signature σ on the input message m using the input signing key sk. $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 1/0$ is a verification algorithm that outputs 1 for acceptance or 0 for rejection depending on the authenticity of the input message-signature pair (m, σ).

3. SYSTEM MODEL

Authenticated multi-pattern matching involves three entities, the data owner, the cloud server, and the client. The data owner owns a set of patterns or keywords. The pattern set is usually too large to be hosted locally for a long time. So the data owner authenticates the patterns and stores them on a cloud server along with the authentication information. The cloud server then provides pattern matching service for the client and proves that the answer it returns is authentic. Each time a client sends a query in the form of a string to the cloud server, the server will

traverse the queried string and find the patterns contained in the queried string. The server will return these matched patterns and the number of occurrences of each pattern to the client, along with a proof for the answer. The client can verify the proof upon receiving the answer. If the proof is rejected, the client may act accordingly, say, filing complaints against the cloud server.

Our construction is based on an AC automaton built from a trie over the outsourced keywords. The authenticated multi-pattern matching scheme consists of four algorithms $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Query}, \mathsf{Verify})$.

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. The data owner runs this PPT algorithm. This algorithm takes in a security parameter 1^λ, and outputs the public-private key pair. The data owner publishes the public key.
- $\mathsf{Setup}(\mathsf{pk}, \mathsf{sk}, \mathcal{P}) \to \mathcal{T}$. The data owner runs this algorithm to construct a trie storing the pattern set $\mathcal{P}$, build an AC automaton on the trie, and authenticate the trie. At the end of this algorithm, the data owner sends the AC automaton $\mathcal{T}$, including the authentication information (e.g., signatures), to the cloud server.
- $\mathsf{Query}(\mathcal{T}, q) \to (R, \pi)$. The cloud server runs this algorithm. This algorithm takes in the AC automaton $\mathcal{T}$ (including its authentication data) stored on the cloud server and a query from a client, and outputs the multi-pattern matching result with a proof for the correctness and authenticity of the result. At the end of this algorithm, the cloud server sends the result and the proof to the querying client.
- $\mathsf{Verify}(q, R, \pi, pk) \to 1/0$. A querying client runs this algorithm to verify the correctness and authenticity of the result returned from the cloud server. This algorithm takes in the query, the result, and the proof as input, and outputs a bit indicating whether the result is correct and authentic. If the result is correct and authentic, the client accepts the result; otherwise, the client rejects the result.

DEFINITION 3.1 (CORRECTNESS). *Authenticated multi-pattern matching scheme is said to be correct, if for all* pk, sk *output by* $\mathsf{KeyGen}(1^\lambda)$*, all pattern sets* $\mathcal{P}$*, all* $\mathcal{T}$ *output by* $\mathsf{Setup}(\mathsf{pk}, \mathsf{sk}, \mathcal{P})$*, and all query* q*, we have* $\mathsf{Verify}(q, R, \pi, \mathsf{pk}) \to 1$ *where* (R, π) *is output by* $\mathsf{Query}(\mathcal{T}, q)$.

DEFINITION 3.2 (SOUNDNESS). *An authenticated multi-pattern matching scheme is said to be sound, if for all* pk, sk *output by* $\mathsf{KeyGen}(1^\lambda)$*, all pattern sets* $\mathcal{P}$*, all* $\mathcal{T}$ *output by* $\mathsf{Setup}(\mathsf{pk}, \mathsf{sk}, \mathcal{P})$*, any PPT adversary* Adv*, and any query* q*, we have the following inequality holds, where* R *is the correct answer of* q *on* $\mathcal{P}$*,*

$$\Pr\left[\begin{array}{l}\{R^*, \pi\} \leftarrow \mathsf{Adv}(1^\lambda, \mathsf{pk}, \mathcal{T}, q);\\ 1 \leftarrow \mathsf{Verify}(q, R^*, \pi, pk);\\ R^* \neq R\end{array}\right] \leq \mathsf{negl}(k)\,.$$

4. PROPOSED SCHEMES

Alphabet Size. There are two different settings for our problem of authenticated multi-pattern matching. In one setting, the alphabet size $|\mathbf{\Sigma}|$ is a small constant. $|\mathbf{\Sigma}|$ is a relatively small constant for most alphabetic scripts. For example, English has 26 distinct characters, and Greek has 24 distinct characters. In this setting, we do not need to consider the effect of $|\Sigma|$ on the efficiency of the authenticated multi-pattern matching system.

In the other setting, $|\mathbf{\Sigma}|$ is relatively large or even infinite. Most pictographic scripts consist of a large number of symbols. Take Sinitic languages as an example, although the number of commonly used characters is about 7,000, the total number of distinct characters is about 85,000 which is huge. The size of a single node becomes huge because it contains $O(|\Sigma|)$ pointers, and only a small portion of them are not null while a large amount of space is wasted. If it is not treated separately, the space efficiency degrades significantly. Hence, certain measures have to be taken to mitigate this impact of the size $|\Sigma|$ on the efficiency.

The major differences in the implementation of the two settings reside in the setup phase to be discussed. The comparison of the two different implementations will be given in the analysis part.

Building Block. Our proposed schemes use a digital signature scheme $\mathcal{SIG}$ and a dynamic accumulator scheme $\mathcal{ACC}$ as building blocks.

4.1 Small Constant Alphabet Size

We first describe the authenticated multi-pattern matching scheme for alphabet size $|\Sigma|$ being a small constant.

$\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. The data owner runs this algorithm to generate the public-private key pairs for $\mathcal{SIG}$ and $\mathcal{ACC}$. The data owner invokes $\mathcal{SIG}.\mathsf{Setup}()$, $\mathcal{SIG}.\mathsf{KeyGen}()$, and $\mathcal{ACC}.\mathsf{Setup}()$ to setup the signature scheme and the accumulator scheme, and obtains $(\mathsf{pk} = (\mathcal{SIG}.\mathsf{pk}, \mathcal{ACC}.\mathsf{param})$, $\mathsf{sk} = (\mathcal{SIG}.\mathsf{sk}, \mathcal{ACC}.\mathsf{aux}))$. The data owner publishes pk.

Algorithm 1 $\mathsf{Setup}(\mathsf{pk}, \mathsf{sk})$

```
1: procedure Setup(pk, sk)
2:     setup AC-automaton T
3:     root_T.acc := G
4:     let Q be a queue
5:     Q.enqueue(root_T)
6:     while Q is not empty do
7:         v := Q.dequeue()
8:         (ID, {w_c}_{c∈Σ∪{fail,suffixP}}, acc, σ) ← v
9:         if p_v is a pattern then
10:            acc* := v.w_suffixP.acc
11:            v.acc := ACC.Add(param, acc*, p_v)
12:        else
13:            v.acc := v.w_suffixP.acc
14:        m := ID||w_c1|| ··· ||w_c|Σ||||w||acc
15:        v.σ := SIG.Sign(pk, sk, m)
16:        for all child node w of v do
17:            Q.enqueue(w)
18:    Output: T
```

$\mathsf{Setup}(\mathsf{pk}, \mathsf{sk}) \to \mathcal{T}$. Algorithm 1 shows the pseudocode of $\mathsf{Setup}()$. The data owner runs this algorithm to setup the AC automaton and provide authentication for the data to be stored on the trie. The data owner first constructs the trie and builds the AC automaton $\mathcal{T}$ on the trie as described in Section 2.2.

For authentication, this algorithm initializes the root node $root_{\mathcal{T}}$ and set $root_{\mathcal{T}}.\mathsf{acc} = G$, where G is the accumulator for empty set from $\mathcal{ACC}.\mathsf{param}$. Let p_v be the word denoted by the path $(root_{\mathcal{T}}, v)$. Each node v on $\mathcal{T}$ stores the tuple $(\mathsf{ID}, \{w_c\}_{c\in\Sigma\cup\{fail, suffixP\}}, \mathsf{acc})$, where ID is the identity of node v representing the insertion timestamp, w_c denotes the child node of v for character c represented by the edge between them, w_{fail} denotes the node that node v's *fail* link connects to, $w_{suffixP}$ denotes the node that node v's *suffixP* link connects to, and acc denotes the accumulator of node v accumulating all the patterns which are the suffixes of the word denoted by the path $(root_{\mathcal{T}}, v)$ (including the word p_v if it is a pattern). In the case that w_c does not exist for some c, which means there is no child edge of v denoting the character c, the data owner sets w_c to NULL.

The algorithm firstly sets up the trie $\mathcal{T}$ and sets up the AC automaton on $\mathcal{T}$, then the data owner traverses $\mathcal{T}$ in a breadth-first-search (BFS) manner. For each node the data owner arrives at, the data owner first finds the node $w_{suffixP}$ which is connected to node v by *suffixP* link, and gets its accumulator $w_{suffixP}.\mathsf{acc}$. If p_v is a pattern, the data owner computes node v's accumulator acc by adding p_v to the accumulator $w_{suffixP}.\mathsf{acc}$; if not, the data owner sets the accumulator of v to $v.w_{suffixP}.\mathsf{acc}$. Then, the data owner generates a signature $\sigma := \mathcal{SIG}.\mathsf{Sign}(\mathsf{pk}, \mathsf{sk}, \mathsf{ID}||w_{c1}||\cdots||w_{c|\Sigma|}||w_{fail}||\mathsf{acc})$ on the information of node v, and updates the node v as (ID, $\{w_c\}_{c\in\Sigma\cup\{fail, suffixP\}}$, acc, σ). After the traversal, the data owner sends $\mathcal{T}$, including the signatures and the accumulators for all the nodes, and the basic AC automaton itself, to the cloud server.

Algorithm 2 $\mathsf{Query}(\mathcal{T}, q)$

```
1: procedure Query(T, q)
2:     let R, B be arrays
3:     let S be a set
4:     let A be a queue storing <accumulator, pattern list>
5:     v := root_T
6:     for all i := 1 to q.length do
7:         while (v.w_{q[i]} == NULL) & (v ≠ root_T) do
8:             S.insert(v)
9:             v := v.w_fail
10:        if (v.w_{q[i]} == NULL) & (v == root_T) then
11:            continue
12:        v := v.w_{q[i]}
13:        S.insert(v)
14:        SP := tracePatterns(v)
15:        if SP ≠ φ then
16:            R.append(SP)
17:            A.enqueue(v.acc, SP)
18:    while A contains more than 1 member do
19:        a_1 := A.dequeue()
20:        a_2 := A.dequeue()
21:        acc := ACC.Accu(param, a_1.SP ⊎ a_2.SP)
22:        A.enqueue(acc, a_1.SP ⊎ a_2.SP)
23:        B.append(acc)
24:    π := (S, B, root_T)
25:    Output: (R, π)
```

$\mathsf{Query}(\mathcal{T}, q) \to (R, \pi)$. Algorithm 2 shows the pseudocode of Query(). When a client wants to find the patterns contained in a string q, the client sends q to the cloud server as a query. Upon receiving a query q, the cloud server runs this algorithm to generate the multi-pattern matching result and a proof for the result.

Let R be a container storing the counters for each pattern hit, and S be a container storing the nodes on $\mathcal{T}$ which are hit during the traversal. The cloud server starts from $root_{\mathcal{T}}$ and traverses $\mathcal{T}$ according to the character sequence in q. When the cloud server arrives at the node v at the $(i-1)$-th step, and needs to continue the traversal according to the character $q[i]$, the i-th character in the sequence q, the cloud server will face one of the following situations.

1. (Case 1 — $q[i]$ on the edge :) Node v has a child edge marked with the character $q[i]$. In this case, the cloud server first adds v to the container S, and then goes to node v's child node $w_{q[i]}$. On the node $w_{q[i]}$, the cloud server performs a tracePatterns() operation. This operation finds the pattern set SP which contains all the suffixes of the word denoted by the path ($root_{\mathcal{T}}$, $w_{q[i]}$). This operation is realized by tracing the *suffixP* link of the nodes recursively starting at $w_{q[i]}$ (refer to Section 2.2). If the word $p_{w_{q[i]}}$ denoted by the path $(root_{\mathcal{T}}, w_{q[i]})$ is also a pattern, it is a member in SP. The cloud server increases the counter for each of the patterns in SP stored in R by 1, adds node $w_{q[i]}$ to the container S, and takes the accumulator acc of node $w_{q[i]}$ for later use. After the cloud server finishes these operations, it continues the traversal with the character $q[i+1]$ from the node $w_{q[i]}$.

2. (Case 2 — No edge with $q[i]$:) There is no existing child edge of node v denoting the character $q[i]$. In this case, the cloud server needs to continue the traversal via the *fail* link of node v. If node v is not $root_{\mathcal{T}}$, the cloud server needs to add node v to the container S indicating that node v is on the path of traversal for query q. This information is important for the verification. Then the cloud server goes to the node w_{fail} to continue the traversal with the character $q[i]$. However, if node v is $root_{\mathcal{T}}$, the cloud server will face a dead lock since the *fail* link of $root_{\mathcal{T}}$ points to $root_{\mathcal{T}}$ itself. According to the mechanism of AC automaton, if we cannot find a child edge denoting a character $q[i]$, then we can safely say that there exists no pattern which is or contains any suffix of the string $q[1]q[2]\cdots q[i]$ (refer to Section 2.2). Hence, in this case, the cloud server can just skip the character $q[i]$ and continue the traversal with the character $q[i+1]$.

After finishing the traversal (reaching the last character of the sequence q), the cloud server constructs an accumulation tree from the accumulators of the nodes hit (which are stored in A) as shown in Figure 2. On the accumulation tree, the leaf nodes store the accumulators of the nodes hit (which are temporarily stored in A). Each parent node stores the accumulator of the multiset union of the child nodes' member sets (the duplicated members are not eliminated). The root stores the accumulator of all the patterns hit during the traversal. The cloud server stores all the nodes except the leaves of the accumulation tree in a container B. R is the result of the query, and $\pi = (S, B, root_{\mathcal{T}})$ is the proof for the authenticity of the result. Note that all the members in the container S are only stored once. The cloud server sends (R, π) to the client at the end of this operation.

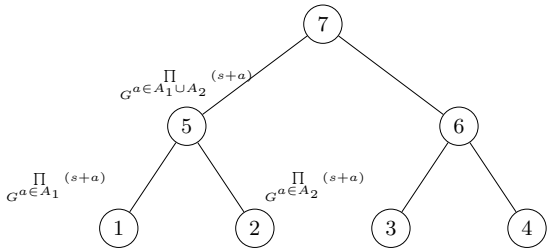


Figure 2: Accumulation Tree

$\mathsf{Verify}(q, R, \pi) \to 1/0$. Algorithm 3 shows the pseudocode of $\mathsf{Verify}()$. The client receives the result R and the proof π after a query. The client parses the proof π as $(S, B, root_{\mathcal{T}})$, and proceeds with the verification.

1. The client verifies the signatures for all the nodes stored in S.

2. The client runs $\mathsf{VerifyPath}()$ to verify the path that the cloud server walked during $\mathsf{Query}()$. This can be easily done as the client holds the query q and all the nodes hit on the path.

3. The client runs $\mathsf{VerifyAccumulator}()$ to verify the correctness of the accumulation tree. The client takes the accumulators of the nodes stored in S as leaf nodes, and verifies all the intermediate nodes received from the cloud server using the algorithm $\mathsf{Verify}_{\uplus}(\mathsf{param}, \mathsf{acc}_p, \mathsf{acc}_{c_1}, \mathsf{acc}_{c_2})$ in Section 2.5, where acc_p is the accumulator of the parent node, acc_{c_1} and acc_{c_2} are the accumulators of the two child nodes. The verification is performed level by level in a bottom-up manner.

4. The client runs $\mathsf{VerifyAccumulationTreeRoot}()$ to verify the root of the accumulation tree. The client constructs an accumulator for the multiset of patterns stored in the container R by $\mathsf{acc}_R \leftarrow \mathsf{Accu}(\mathsf{param}, R)$, and evaluates the equation $\mathsf{acc}_R == \mathsf{acc}_{root}$.

If all the signatures are authentic, the path is correct, the accumulation tree can be correctly verified, the accumulator stored on the root of the accumulation tree accumulates the multiset of patterns stored in the result R, then the result R is correct; otherwise, the client rejects the result.

The mismatch cases are not specially authenticated since they are just the cases that the result is an empty set. In those cases, every accumulator on nodes accumulates an empty set, and every root accumulator also accumulates an empty set. All steps need not be specially treated for empty sets.

4.2 Large Alphabet Size

In the case that the alphabet size, $|\Sigma|$, is relatively large (or even unlimited), the size of the child node set $\{w_c\}_{c\in\Sigma}$ is large for a large $|\Sigma|$. This fact makes the path verification resource consuming. We need to take special measures to provide authentication for the multi-pattern matching queries.

For our scheme described in Section 4.1, the cloud server sends the complete node information (including all the child edges and null edges) of each node hit during the traversal to the client as a part of the proof. When the alphabet size is large or even unlimited, the number of possible child edges is also large or unlimited, though the number of effective edges is small. This makes the proof inefficient to generate.

Algorithm 3 $\mathsf{Verify}(q, R, \pi)$

```
1: procedure Verify(q, R, π)
2:     (S, B, root_T) ← π
3:     for v in S do
4:         b_0 := b_0 & SIG.Verify(pk, v.m, v.σ)
5:     b_1 := VerifyPath(q, S)
6:     b_2 := VerifyAccumulator(S, B)
7:     b_3 := VerifyAccumulationTreeRoot(R, B)
8:     if b_0 & b_1 & b_2 & b_3 is true then
9:         Output: Accept
10:    else
11:        Output: Reject
```

To solve this problem, the data owner generates signatures for each pair of adjacent child edges of each node, instead of directly signing on each node. Each signature now not only authenticates the two child edges it signs on, but also proves that there exist no other child edges for the characters in between the two characters corresponding to the two signed child edges.

During each query, the cloud server picks only the necessary edge information to generate the proof, and leaves out the unnecessary edge information. The cloud server picks the walked edge of each node and sends the signature for that specific edge to the client as the proof. In this way, the client only needs to verify the child edge it needs to walk, instead of verifying the node. $\mathsf{KeyGen}()$ and $\mathsf{Verify}()$ remains the same as the scheme in Section 4.1. We describe $\mathsf{Setup}()$ and $\mathsf{Query}()$ in this setting below.

Algorithm 4 $\mathsf{Setup}(\mathsf{pk}, \mathsf{sk})$

```
1: procedure Setup(pk, sk)
2:     setup AC-automaton T
3:     root_T.acc := G
4:     let Q be a queue
5:     Q.enqueue(root_T)
6:     while Q is not empty do
7:         v := Q.dequeue()
8:         (ID, c_v, {w_c}_{c∈Σ_v∪{fail,suffixP}}, acc, {σ_c}) ← v
9:         if p_v is a pattern then
10:            acc* := v.w_suffixP.acc
11:            v.acc := ACC.Add(param, acc*, p_v)
12:        else
13:            v.acc := v.w_suffixP.acc
14:        for (v.w_{c_i}, v.w_{c_j} ≠ NULL) & (∄k ∈ (i, j) s.t.
       v.w_{c_k} ≠ NULL) do
15:            m := ID||w_{c_i}||w_{c_j}||w_fail||acc
16:            v.σ_{c_i} := SIG.Sign(pk, sk, m)
17:        for all child node w of v do
18:            Q.enqueue(w)
19:    Output: T
```

$\mathsf{Setup}()$ is modified as shown by the pseudocode in Algorithm 4. The alphabet Σ is sorted beforehand. Each node v only stores the existing child nodes with two additional virtual nodes $w_{-\infty}$ and $w_{+\infty}$. That is to say, for all child nodes w_c stored on node v, the character c is denoted by an existing child edge. Node v's subset $\{c\} \subset \Sigma$ is sorted

in the same order of the alphabet Σ. There is one more item c_v stored on node v which is used to identify the character denoted by the edge between node v and its parent node.

Setup() constructs the trie and initiates the AC automaton. The data owner then traverses $\mathcal{T}$ in a BFS manner. During the traversal, the data owner sets up the accumulators for each node in the same way as it does in the case of small $|\Sigma|$. When it comes to the signature generation, the data owner behaves differently as discussed. For each node v, the data owner creates one signature for each pair of adjacent characters c_i and c_j, which certifies the fact that node v has two child edges denoting characters c_i and c_j respectively and node v does not have any child edge denoting any characters between c_i and c_j.

An example is shown in Figure 3. The node v shown in Figure 3 has four child edges denoting the characters 'a', 'c', 'r', and 's' respectively, along with two virtual edges $-\infty$ and $+\infty$. Node v has five pairs of adjacent child edge pairs, ($-\infty$, 'a'), ('a', 'c'), ('c', 'r'), ('r', 's'), and ('s', $+\infty$). The data owner constructs the following five pieces of information $m_{-\infty} = \mathsf{ID}||w_{-\infty}||w_a||w_{fail}||\mathsf{acc}$, $m_a = \mathsf{ID}||w_a||w_c||w_{fail}||\mathsf{acc}$, $m_c = \mathsf{ID}||w_c\ ||w_r||w_{fail}||\mathsf{acc}$, $m_r = \mathsf{ID}\ ||w_r||w_s||w_{fail}||\mathsf{acc}$, and $m_s = \mathsf{ID}||w_s||w_{+\infty}||w_{fail}||\mathsf{acc}$, and then generates five signatures $\sigma_{-\infty}$, σ_a, σ_c, σ_r, and σ_s on the five pieces of information respectively with the signature scheme $\mathcal{SIG}$.

After finishing the traversal, the data owner sends $\mathcal{T}$, along with the signatures and the accumulators of all the nodes, to the cloud server.

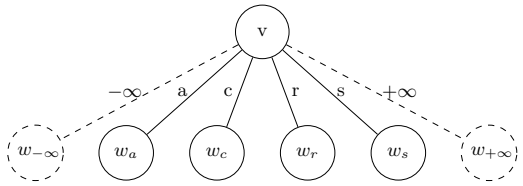


Figure 3: Signing on the Edges

Query() is modified as shown by the pseudocode in Algorithm 5. The cloud server traverses $\mathcal{T}$ in the same way as it does in the case of small $|\Sigma|$. For each node hit, the cloud server only sends the authentication of the walked child edges to the client. Suppose the cloud server arrives node v at the $(i-1)$-th step, it needs to generate the proof showing that node v is hit during the traversal and goes to the next node according to $q[i]$. As the case of small $|\Sigma|$, the cloud server will face one of the following conditions. The traversal behavior is the same as in the case of small $|\Sigma|$. Here, we only describe the different part.

1. Node v has a child edge denoting the character $q[i]$: The cloud server finds (j,k) so that $q[i] == c_j$, and $v.w_{c_j}, v.w_{c_k}$ are adjacent child nodes of v. The cloud server then sends $(w_{c_j}, w_{c_k}, w_{fail}, \sigma_{c_j})$ to the client for path verification.

2. There is no existing child edge of node v denoting the character $q[i]$: The cloud server finds (j,k) so that $q[i]$ falls between c_j and c_k in Σ, and sends $(w_{c_j}, w_{c_k}, w_{fail}, \sigma_{c_j})$ to the client for path verification.

To illustrate, let us consider an example shown in Figure 3 again. When the cloud server arrives at node v during the traversal at the $(i-1)$-th step, it takes the next step according to the character $q[i]$. If $q[i] \in$ {'a', 'c', 'r', 's'}, the cloud server can just give out the corresponding signature as the proof. Take $q[i] ==$ 'c' as an example, the cloud server just picks $v.\sigma_c$ to prove that w_c is truly the node hit in the next step. On the other hand, if $q[i] \notin$ {'a', 'c', 'r', 's'}, the cloud server needs to go to w_{fail} in the next step. In this case, the cloud server needs to prove two statements that node v does not have a child edge corresponding to the character $q[i]$ and that w_{fail} is the node connected by v's *fail* link. Take $q[i] ==$ 'h' as an example, 'h' falls between 'c' and 'r', so the cloud server picks the signature $v.\sigma_c$ as the proof for the two statements. $v.\sigma_c$ indicates that there is no character between 'c' and 'r' which corresponds to one child edge of v. Meanwhile, $v.\sigma_c$ also authenticates w_{fail}. Thus, both statements are proved by this single signature. The cloud server just adds part of node v's information, $\mathsf{ID}||w_c||w_r||w_{fail}||\mathsf{acc}||\sigma_c$, to the container S. The rest of the operations remain the same as stated in the case of small $|\Sigma|$.

Algorithm 5 Query($\mathcal{T}, q$)

```
1: procedure Query(T, q)
2:     let R, B be arrays
3:     let S be a set
4:     let A be a queue storing <accumulator, pattern list>
5:     v := root_T
6:     for all i := 1 to q.length do
7:         while (v.w_{q[i]} == NULL) & (v ≠ root_T) do
8:             find i, j s.t. (v.w_{c_i}, v.w_{c_j} ≠ NULL) & (q[i] is between c_i and c_j)
9:             v' := (v.ID, {v.w_{c_i}, v.w_{c_j}, v.w_{fail}}, v.acc, v.σ_{c_i})
10:            S.insert(v')
11:            v := v.w_{fail}
12:        if (v.w_{q[i]} == NULL) & (v == root_T) then
13:            continue
14:        A.enqueue(v.acc)
15:        find j, k s.t. (q[i] == c_j) & (c_k is the first character after q[i] in the Σ that v.w_{c_k} ≠ NULL)
16:        v' := (v.ID, {v.w_{c_j}, v.w_{c_k}, v.w_{fail}}, v.acc, σ_{c_j})
17:        S.insert(v')
18:        SP := tracePatterns(v)
19:        R.append(SP)
20:        A.append(v.acc, SP)
21:    while A contains more than 1 members do
22:        a_1 := A.dequeue()
23:        a_2 := A.dequeue()
24:        acc := ACC.Accu(param, a_1.SP ⊎ a_2.SP)
25:        A.enqueue(acc, a_1.SP ⊎ a_2.SP)
26:        B.append(acc)
27:    π := (S, B, root_T)
28:    Output: (R, π)
```

4.3 Complexity Analysis

We discuss the complexity of our schemes in this section, and further discuss the feasibility to parallelize the schemes.

Setup. The setup consists of two main parts: the AC-automaton setup and the later authentication information setup. With no doubts, the AC automaton setup complexity is $O(N\bar{l})$, which is the total length of the patterns. The authentication information setup part for the small alphabet case accessed every node on the trie once in a BFS way.

The complexity of operations for every node is $O(1)$, and the number of nodes does not exceed $N\bar{l}$, so the total setup complexity is $O(N\bar{l})$.

For the large alphabet size case, during the nodes traversal, each adjacent edge pair is also accessed once. The number of edges is the number of nodes minus one (the root has no parent edge and each other node has one and only parent edge exclusively). So, the total complexity is also $O(N\bar{l})$.

For each node, there is only one accumulator update conducted, so for either case, overall only $O(N)$ accumulator updates are conducted.

Space. For the small alphabet case, each node is associated with an accumulator and a signature, which are all constant size. So the space usage is $O(N\bar{l})$. For the large alphabet, each node is associated with an accumulator and each edge is associated with a signature. So, the total space used in this case is also $O(N\bar{l})$.

Query. The original query time complexity of the AC automaton is $O(n+m)$. The extra complexity introduced by our schemes is resulted from the accumulation tree computation. The number of the nodes at the leaf level is bounded by n, so the height of the accumulation tree is bounded by $O(\log n)$, as it is a binary tree. For different levels, the numbers of nodes are different, but the total number of elements for each accumulator at each level is the same, thus m. Calculating an accumulator without secret key costs $O(k \log k)$ time where k is the number of the elements it accumulates. So the calculation time for each level is bounded by $O(m \log m)$. As the result, the total complexity for the accumulation tree is $O(m \log m \log n)$. The total complexity for the query algorithm is thus $O(n + m \log m \log n)$, where $O(n)$ accumulations are conducted.

Verification. The verification complexity comes from four parts. The first is for verifying all the signature of nodes in the small alphabet case, or the signature of edges in the large alphabet case. Apparently, this step costs $O(n)$ complexity, since the number of nodes or edges does not exceed two times the length of queries as guaranteed by AC automaton.

The second step of verifying the path is just traversing the query and recording the accumulators. The number of nodes accessed does not exceed $2n$. The complexity of this step is lower than the AC automaton's searching, since the client already has the final (temporally untrusted) result, and needs not (and actually could not backtrace) to fetch hitting patterns.

The third part of verifying the accumulator tree costs $O(n)$ time, since there are $O(n)$ internal nodes to be verified and verifying each node costs only two pairings. This step is much faster than generating those accumulators, since generating an accumulator needs conducting FFT on the set, while verifying if one accumulator is the union of other two accumulators needs two pairings.

The last step of verifying the root costs $O(m \log m)$ complexity because it is just a round of FFT on the result set. As the result, the overall verification complexity is $O(n + m \log m)$. Among all the operations, $O(n)$ pairings are conducted.

Proof Size. The proof consists of two parts: nodes set S and internal accumulator tree node array B. The size of either one does not exceed $O(n)$. So, the overall length of the proof is $O(n)$.

Parallelism. The scheme can be easily adapted to a parallel computing environment. Firstly, for the setup, both AC automaton and the accumulator construction are done in a BFS order, and constructing a node needs only information from the higher level nodes. So, the nodes at the same level can be constructed simultaneously. Secondly, for the query, the proof construction mainly spends time on the accumulation tree construction, where the accumulators at the same level can be constructed simultaneously without conflict. Besides, for each node, the accumulator construction relies on FFT which is much more efficient in a parallel environment. Thirdly, for the verification, the signature verification can be done in parallel and the accumulation tree can be similarly verified in parallel at each level. The FFT required for verifying the root accumulator can also be parallelized.

4.4 Security

THEOREM 4.1. *The authenticated multi-pattern matching schemes described above are correct and sound, according to Definitions 3.1 and 3.2.*

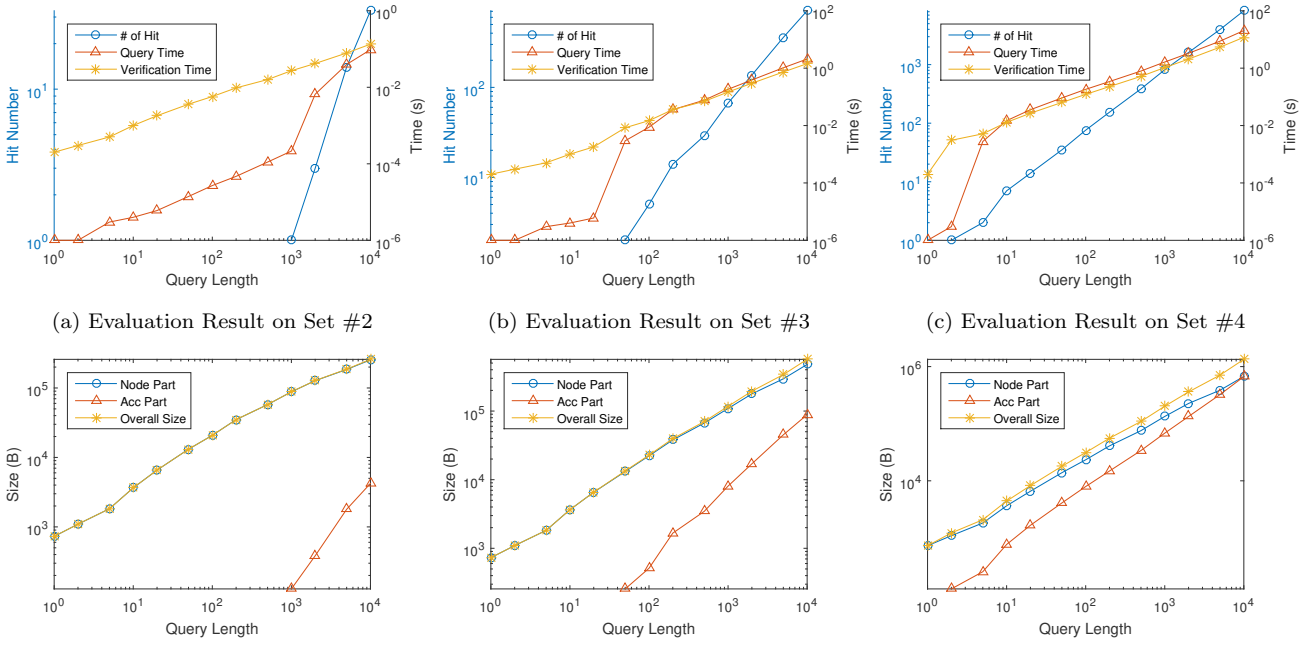
The correctness of the result follows the correctness of the AC automaton. The correctness of proof is elaborated as follow: 1) The signature guaranteed the authenticity of the nodes including the accumulators. 2) With those authenticated nodes, the AC automaton can output a set of accumulators which are also inherently authenticated, because the automaton algorithm is fixed and the input of the automaton is authenticated. 3) The accumulation tree structure guaranteed that the root accumulator is the accumulation of all the authenticated accumulators, which implies that the root accumulator is the accumulation of the result. 4) The ultimate result is verified by the relationship between the root accumulator and the result set.

For the soundness, if an adversarial cloud server can output a wrong result R with a proof π which passes the verification, it implies the ability to either generate forgeries for $\mathcal{SIG}$, or to break the security of $\mathcal{ACC}$. In R, the node set S must contain all the hit nodes of the query q; otherwise, $\mathsf{Verify}()$ trivially rejects the result. Hence, it is reasonable to assume the node set S is correct with respect to q. If $\mathcal{ACC}$ is secure, then the accumulators stored in the nodes and on the accumulation tree accumulate the members from a wrong pattern set corresponding to the result R, which indicates that the accumulators stored in the nodes are not genuine. Hence, the signatures on the nodes are forged by the adversarial cloud server. If $\mathcal{SIG}$ is unforgeable, then the accumulators stored in the nodes are genuine. In this case, it means that genuine accumulators yield a wrong multiset union result, which implies that the adversarial cloud server can break the security of $\mathcal{ACC}$.

5. EVALUATIONS

In this section, we discuss the performance of our proposed scheme. For evaluation, we implemented a keyword searching application that uses the small $|\Sigma|$ scheme described in Section 4. The application reads pattern strings, generates the AC automaton, and sets up the authentication related information thereafter. Then, it reads query strings and answers all the occurrences with proofs. Finally, it checks the answers with the proofs.

The application was all written in C++ with third-party libraries including PBC [3] for pairing, Crypto++ [1] for



(a) Evaluation Result on Set #2 (b) Evaluation Result on Set #3 (c) Evaluation Result on Set #4

(d) Proof Size on Set #2 (e) Proof Size on Set #3 (f) Proof Size on Set #4

Figure 4: Number of hit, query time, verification time, and proof size (the nodes (Node) and the accumulators (Acc)) vs. Query string length (for different pattern word sets)

signature, and GMP [21] for high precision computation. We employ type-I (symmetric) bilinear group. PBC library implements such group over an elliptic curve of 512 bits, providing security of 1024-bit discrete logarithm. It is also easier to instantiate the accumulator by a symmetric bilinear group, or verification may require each node to have two accumulators, one in each of the asymmetric base groups.

The latter parts in this section show the time, storage, and some other performance quantities for each step. All the performance experiments were conducted in a PC with Core i5 3.2GHz CPU and 8GB main memory.

5.1 English Keyword Search

As one of the most important applications for pattern matching, keyword search demonstrates the performance of the scheme well in many practical scenarios. We test our application in a natural language environment to better emulate practical cases.

Setup. To evaluate the scheme, we need a typical and representative pattern string set. After researching on a lot of different sets, we chose **Complete English Words List** provided by SIL [2] as the pattern set (hot word). This list contains over 100,000 real English words. This set is the set #4 in our experiment. To test the performance in different scales, besides using the whole dictionary, we also randomly sampled one hundred, one thousand, and ten thousand words from the list, and tested the time used to set them up respectively. The results are shown in Table 3.

According to the experiment result, we can see that the time used to setup increases nearly linearly with the pattern number, because most of the setup time was used in accumulator updating and it happens once and only once for every pattern. However, the storage used to store the automaton and authentication information does not increasing linearly with the size, because of the compression feature of trie: the common prefixes of all the patterns are stored only once. So, at least for English, the larger the pattern set is, the higher compression ratio it will be. Hence, we believe in a larger scale, the storage performance could even be better because of the compression.

Table 3: Setup performance test result

Number #	Number of Patterns	$\bar{l}$	Time	Storage
1	100	8.43	289 ms	273 KB
2	1,000	8.45	2.89 s	2.32 MB
3	10,000	8.55	29.5 s	17.34 MB
4	109,582	9.53	318.8 s	87.12 MB

Query & Verification Time. We selected some English texts from wiki pages as the query texts and tested the time used to answer the query on different pattern scale and different query lengths (result from pattern set #1 is abandoned since that set is too small to be typical). One may question the necessity to conducting experiment on different pattern scale, since according to the complexity analysis, the query time is independent of the scale. We found that in the natural language environment, the pattern size influences the hit rate, which in turns influences the result set size, and therefore influences the query time indirectly. In other words, the more keywords, the more hit will be obtained and slower the query will be. The query times used for different pattern sets were measured and are shown in Figure 4a, Figure 4b and Figure 4c.

The evaluation results show that the query time is negligible when there is no pattern hit, because the accumulation tree is null for those cases. So, we focus on the matched

cases. We analyzed the cases with similar hit pattern number but different query lengths to see how the query length influences query time, and analyzed the cases with different hit pattern number but the same query length to see the effect of result set size. We found that the query time is mainly influenced by the hit pattern set size, because most of the time was spent on computing the accumulators of the tree. While the query length results in only tiny difference, as the number of hit pattern increases, the query time increases not prominently faster than linear. As a representative case, a query of 1,000 characters on pattern set #3 costs 183ms, where there are 67 hits.

We also tested the verification time used in those cases. The verification time also increases linearly. As a reason, according to our analysis, the dominant part of time lies in the verification of the accumulator tree, or in other words, the pairings. The root accumulator verification in fact costs negligible time, in our test scale. We believe that the verification time may increase faster than linear in some extreme cases where the result set is very large. But, even in the case which has the largest result size, the verification time is dominated by the pairing calculation and the effect from the result size is not obvious enough. As a representative case, verification to a query of 1,000 characters on pattern set #3 costs 147ms.

Proof Size. The proof size increases linearly with query length, as shown in Figure 4d, Figure 4e, and Figure 4f. In theory, the proof size is independent of the pattern set size. However, by comparing the proof sizes for the same query on two different pattern sets, we found that the proof size for a larger set is larger. This is because the server may walk through one node more than once, but the authentication information is stored only once, yet the user can still correctly verify that. It is deduplication in a sense. When the set size is small, a node is more likely to be accessed multiple times in the natural language settings. So, it needs a smaller space to contain less nodes' authentication information. The overall proof size is still acceptable and the proof can be further compressed using generic compression algorithms.

Analysis. The evaluation reflects the performance of the scheme in a natural language keyword matching scenario. The time spent on Setup, Query, and Verify are all reasonable and practical enough for real world usage. Most importantly, the times for both Query and Verify are not directly influenced by the number of patterns.

5.2 Evaluation Beyond Natural Language

Section 5.1 tested the performance of our scheme in a natural language environment. Some extreme situation may never happen in this environment. For example, a query text of length n theoretically may result in a result set which at most has $n \cdot (n+1)/2$ elements, where all the substrings of the query are in the pattern set. Yet, meaningful text can hardly have so many meaningful keywords occurred. Even for the complete word set used in Section 5.1, the highest witnessed hitting rate (m/n) is 0.83. Besides, the hitting rate is influenced by the pattern set size, which masked the effect from pattern size to query time, and we cannot distinguish the time consumption resulted from the larger pattern set or from the larger result set. So, we also devised some experiments on data set beyond natural language to show the effects from those parameters only.

Effects from Hit Pattern Number. To test the effects from hit pattern number m only, we fixed the pattern set size to be 10,000 and the query length to be 100, then we craft the content of the query string to make the result number different (with meaningless string).

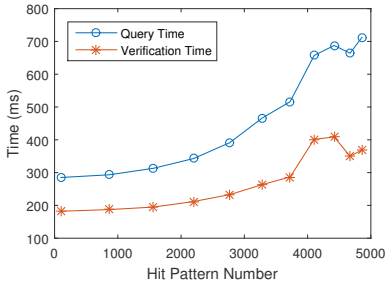


Figure 5: Performance vs. Hit Pattern Number

Figure 5 shows the time used to query and verify in those cases with different number of hit patterns. As we can see from the result, the time consumption increases with the hit pattern number slightly faster than linear, which is compatible with our theoretical analysis. In the case where almost all the substrings are hit, the performance is relatively bad. But we believe it happens only when the data set is special. In real situations, it does not happen often.

Effects from Pattern Set Size. Theoretically, the pattern set size does not influence the query time directly. But in general, a larger pattern set implies a higher hitting possibility. We crafted some meaningless patterns to make the hitting rate independent of the pattern set size, and tested the time performance in different pattern sets.

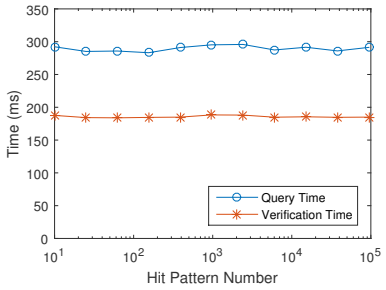


Figure 6: Performance vs. Pattern Number

We fix the query length at 100 and the hit pattern number also at 100. Figure 6 shows the time used for query and verification on random pattern set with different sizes. The results show that the time consumption is nearly irrelevant to the pattern set size, which implies that it is an efficient multi-pattern matching scheme.

6. CONCLUSION

This paper presented the first authenticated multi-pattern matching scheme, which enables a data owner who owns a large amount of patterns to outsource them to an untrusted cloud server which provides pattern matching services, such that any clients can verify the authenticity of the query result without contacting the data owner or getting the whole

set of data. Our schemes are efficient and do not introduce extra storage complexity at the cloud server. The proof attached with the result is of an acceptable size, which does not prominently increase the communication overhead. The verification process is also efficient. At last, we conducted a lot of evaluations to support the efficiency claim.

Acknowledgement

Sherman Chow is supported by the Direct Grant (4055018) of the Chinese University of Hong Kong, Early Career Scheme and the Early Career Award (CUHK 439713) and General Research Funds (CUHK 14201914) of the Research Grants Council, University Grant Committee of Hong Kong.

The work of Kehuan Zhang was partially supported by Direct Grant (4055047) of the Chinese University of Hong Kong, NSFC (61572415), and the General Research Funds (CUHK 24207815) established under the Research Grants Council, University Grant Committee of Hong Kong.

7. REFERENCES

[1] Crypto++ Library. http://www.cryptopp.com. Accessed: 2015-Jul-11.

[2] English Wordlists Source. http://www-01.sil.org/linguistics/wordlists/english. Accessed: 2015-Nov-11.

[3] The PBC (Pairing-Based Cryptography) Library. https://crypto.stanford.edu/pbc. Accessed: 2015-Jul-11.

[4] T. Acar, S. S. M. Chow, and L. Nguyen. Accumulators and U-Prove Revocation. In *Financial Cryptography*, pages 189–196, 2013.

[5] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.

[6] 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*, pages 90–108, 2013.

[7] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security Symposium*, pages 781–796, 2014.

[8] E. Bertino, B. Carminati, E. Ferrari, B. M. Thuraisingham, and A. Gupta. Selective and Authentic Third-Party Distribution of XML Documents. *IEEE Trans. Knowl. Data Eng.*, 16(10):1263–1278, 2004.

[9] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again. In *ACM ITCS*, pages 326–349, 2012.

[10] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive Composition and Bootstrapping for SNARKS and Proof-carrying Data. In *ACM STOC*, pages 111–120, 2013.

[11] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct Non-interactive Arguments via Linear Interactive Proofs. In *Theory of Cryptography Conference (TCC)*, pages 315–333, 2013.

[12] D. Boneh and X. Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *J. Cryptology*, 21(2):149–177, 2008.

[13] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[14] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying Computations with State. In *ACM SOSP*, pages 341–357, 2013.

[15] Y. Chen, S. S. M. Chow, K. Chung, R. W. F. Lai, W. Lin, and H. Zhou. Cryptography for Parallel RAM from Indistinguishability Obfuscation. In *ACM ITCS*, pages 179–190, 2016.

[16] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile Verifiable Computation. In *IEEE Symp. on Security and Privacy*, pages 253–270, 2015.

[17] P. T. Devanbu, M. Gertz, and A. Kwong. Flexible Authentication of XML Documents. *Journal of Computer Security*, 12(6):841–864, 2004.

[18] S. Faust, C. Hazay, and D. Venturi. Outsourced Pattern Matching. In *ICALP*, pages 545–556, 2013.

[19] R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *CRYPTO*, pages 465–482, 2010.

[20] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic Span Programs and Succinct NIZKs without PCPPs. In *EUROCRYPT*, pages 626–645, 2013.

[21] T. Granlund and the GMP Development Team. The GNU Multiple Precision Arithmetic Library. https://gmplib.org, 2006. Accessed: 2015-Jul-11.

[22] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[23] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1):21–41, 2004.

[24] R. C. Merkle. A Certified Digital Signature. In *CRYPTO*, pages 218–238, 1989.

[25] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[26] L. Nguyen. Accumulators from Bilinear Pairings and Applications. In *CT-RSA*, pages 275–292, 2005.

[27] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Practical Authenticated Pattern Matching with Optimal Proof Size. *PVLDB*, 8(7):750–761, 2015.

[28] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *IEEE Symp. on Security and Privacy*, pages 238–252, 2013.

[29] S. T. V. 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*, pages 71–84, 2013.

[30] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making Argument Systems for Outsourced Computation Practical (Sometimes). In *NDSS*, 2012.

[31] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A Hybrid Architecture for Interactive Verifiable Computation. In *IEEE Symp. on Security and Privacy*, pages 223–237, 2013.