Secure Hashing-Based Verifiable Pattern Matching

Fei Chen, Donghong Wang, Ronghua Li, Jianyong Chen, Zhong Ming, Alex X. Liu, Huayi Duan, Cong Wang, and Jing Qin

***Abstract*— Verifiable pattern matching is the problem of finding a given pattern *verifiably* from the outsourced textual data, which is resident in an untrusted remote server. This problem has drawn much attention due to a large number of applications. The state-of-the-art method for this problem suffers from low efficiency. To enable *fast* verifiable pattern matching, we propose a novel scheme in this paper. Our scheme is based on an ordered set accumulator data structure and a newly developed verifiable suffix array structure, which only involves fast cryptographic hash computations. Our scheme also supports fast multiple-occurrence pattern matching. A striking feature of our proposed scheme is that our scheme works even with no secret keys, which ensures public verifiability. We conduct extensive experiments to evaluate the proposed scheme using Java. The results show that our scheme is orders of magnitude faster than the state-of-the-art work. Specifically, our scheme with public verifiability only costs a preprocessing time of 47 s (merely one-time off-line cost during outsourcing), a search time of 30 μs, a verification time of 149 μs, and a proof size of 2760 bytes for a verifiable pattern matching query with pattern length 200 on 10-million long textual data which consists of sequences of two-byte, Unicode characters in Java.**

***Index Terms*— Pattern matching outsourcing, verifiability, accumulator, hashing.**

Manuscript received December 25, 2017; revised March 10, 2018; accepted March 18, 2018. Date of publication April 9, 2018; date of current version May 21, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61502314 and Grant 61672358, in part by the Science and Technology Plan Projects of Shenzhen under Grant JCYJ20160307115030281 and Grant JCYJ20170302145623566, in part by a Grant from the Innovation and Technology Fund of Hong Kong under Project ITS/304/16, and in part by CCF-Venustech funding under Grant CCF-VenustechRP2017001. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Julien Bringer.
(Corresponding author: Jianyong Chen.)

F. Chen, D. Wang, J. Chen, and Z. Ming are with the College of Computer Science and Engineering, Shenzhen University, Shenzhen 518060, China (e-mail: fchen@szu.edu.cn; dhwwang@gmail.com; jychen@szu.edu.cn; mingz@szu.edu.cn).

R. Li is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: lironghuascut@gmail.com).

A. X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA, and also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210008, China (e-mail: alexliu@cse.msu.edu).

H. Duan and C. Wang are with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: hduan2-c@my.cityu.edu.hk; congwang@cityu.edu.hk).

J. Qin is with the Center for Smart Health, School of Nursing, The Hong Kong Polytechnic University, Hong Kong (e-mail: harry.qin@polyu.edu.hk).

This paper has supplementary downloadable material at http://ieeexplore.ieee.org, provided by the authors. The file consists of additional performance evaluation results of the proposed scheme. The material is 139 KB in size.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2825141

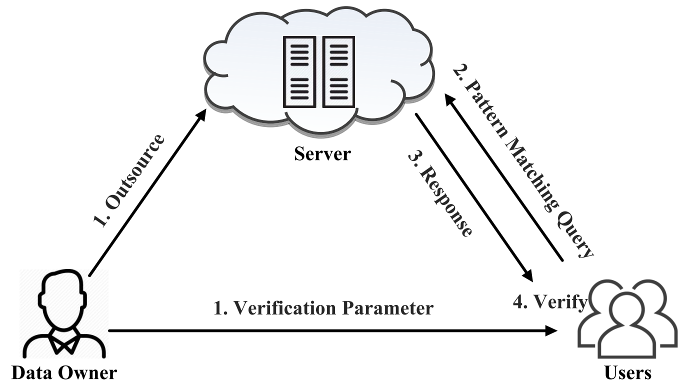


Fig. 1. Verifiable pattern matching model.

I. Introduction

A. Motivation and Problem Statement

PATTERN matching has found numerous applications in deep packet inspection, intrusion detection, spam filtering, text search, bioinformatics etc. With the ongoing paradigm shift of cloud computing, researchers are studying traditional pattern matching outsourcing to a remote server [1]–[7], which is the core of cloud-based applications involving pattern matching, e.g. cloud firewall, cloud middlebox services, etc. While pattern matching is well studied, outsourcing pattern matching query to a remote server poses new challenges, because the remote server may not be trusted for concerns on its management issues, software and hardware failures, potential hacking incentives etc. Among the challenges, a fundamental one is how to verify the correctness of outsourced pattern matching results from the remote server, which is referred to as *verifiable pattern matching*.

In this paper, we study verifiable pattern matching in a typical service model shown in Fig. 1, where three types of entities are involved, i.e. *data owner*, *data user*, and *server*. The data owner outsources/publishes its data, which can be regarded as a long sequence of characters from a given alphabet, to a remote server (e.g. cloud). Afterwards, pattern matching queries can be issued by data users to the server on demand; the query result could be one or more matches of the queried pattern in the outsourced data, (collectively) called the case of *match*, or no match at all, called the case of *mismatch*.

The data owner and the data user trust each other, but not the server. The untrusted server, incentivized by the economic benefit from reduced computation burden for example, may only do partial processing and return a mismatch result for a genuinely matched query, and vice versa. Under a verifiable setting, the data owner will send augmenting authentication information, in addition to the outsourced data itself, to the

1556-6013 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

server during system setup; later, the server must return corresponding *proof* along with the result in response to subsequent query. The user can in turn validate the query result with the proof.

B. Limitation of Prior Art

The main limitation of current verifiable pattern matching solutions [5], [8]–[10] is that they require large computation, communication overhead. The state-of-the-art solution for verifiable pattern matching appeared at [5]. The proposed scheme features *asymptotically* optimal proof size; the search time is also independent on the outsourced text size. However, it relies on heavy public key cryptographic computations over big integers. Besides the state-of-the-art work, other solutions also exist. The general solution in [8] requires considerable communication cost because the nodes in the underlying directed acyclic graph have variant out-degrees. Other works [9], [10] are mainly of theoretic interest; they formulate the pattern matching problem as polynomial/integer substraction, thus the pattern matching takes time linear in the text size, which is pretty slow in practice for million-scale and larger texts.

C. Proposed Approach

To enable *fast* verifiable pattern matching, we propose a novel scheme in this work. Our scheme only relies on collision resistant hash functions and general-purpose secure authentication schemes. This makes our scheme orders of magnitude faster than the state-of-the-art work [5]. Our scheme even works with no secret key, if required.

From a high level, our scheme works as follows. We first enable fast pattern matching using a traditional approach. Specifically, given a string with length n, we sort all its n suffix strings in ascending order. To search a pattern string p, we use the binary search method; we compare p with the sorted suffix strings. If the pattern p is a match, p can be matched by a suffix string. Algebraically, there exists two suffix strings α, β such that $\beta = p||\alpha$ where $||$ denotes string concatenation. Similarly, if the pattern is a mismatch, the binary search stops with two neighbouring sorted suffix strings such that the pattern lies between them. Algebraically, there exists two neighboured suffix strings α, β such that $\alpha < p < \beta$. Therefore, with the sorted suffix strings, the pattern matching problem is transformed into an algebraical problem.

Next, we design fast algorithms to verify the two algebraic operations, i.e. string concatenation $\beta = p||\alpha$ and string comparison $\alpha < p < \beta$. To verify an equality $\beta = p||\alpha$ for pattern match, we design a fast authenticated data structure based on a collision-resistant cryptographic hash function to allow a user checking the correctness of the equation. The rationale of our data structure is that it compresses each suffix string into an accumulated value that depends on all the characters of the suffix string sequentially, which enables equality verification.

To verify an inequality $\alpha < p < \beta$ for pattern mismatch, we base the verification on an equality verification. We first find the longest common prefix string of α and p; then we extract a prefix of α containing the longest common prefix string and one more subsequent character, denoted as γ. To check the correctness of $\alpha < p$, we only need to show γ is contained in the outsourced string using an equality verification. Similarly, we can prove $p < \beta$. To finally show pattern p cannot be matched, we further prove that the two suffix strings α, β are adjacent in the sorted suffix strings. In our scheme, we use authenticated position information to achieve this.

Using the verifiable equality and inequality checks, we finally enable fast verifiable pattern matching. We also optimize our scheme from three perspectives. We improve the search time and the size of the proof returned from the server to the optimal. We improve our scheme to support verifiability. We also propose a more efficient scheme for a multi-occurrence pattern match. It has fewer hash computations than the direct approach that invokes the basic scheme multiple times for each pattern match.

D. Technical Challenges and Solutions

To enable fast verifiable pattern matching, two main technical challenges exist. The first challenge is how to design a verifiable data structure to enable *fast* pattern matching for both match and mismatch cases. The second is how to enable fast verification of a multiple-occurrence pattern matching.

For the first challenge, traditional approaches employ a general accumulator to validate *string concatenation* and *string comparison* relations. However, general accumulators are based on public key cryptography and work over big integer operations inefficiently. We aim at fast solutions. For pattern matching, we observe that the strings are ordered and that general accumulators are heavy tools. We thus design a light-weight, efficient, collision-resistant-hash-only hash-chain-like accumulator for strings, which we call ordered set accumulator. This data structure supports extremely fast string concatenation and string comparison verifications.

For the second challenge, a straight-forward approach is to perform a verification for each matched pattern. While this approach works, the communication cost is considerably high that low-end devices may not be able to afford. Thus a faster solution is better. We find that if a pattern can be matched multiple times, their corresponding suffixes must be adjacent to each other in the sorted suffix strings. By recording the length of the longest common prefixes of each adjacent suffix strings, multiple-occurrence matching verification can be made faster. The multi-occurrence match verification problem is then reduced to verify the first match, the last match, and whether the lengths of the longest common prefixes in between are larger than the length of the pattern. The first and last match verification is similar to the basic scheme; for the length verifications of the longest common prefixes, we build again an ordered set accumulator to enable fast verifications.

E. Summary of Experimental Results

We prototyped and open-sourced our code in Java. We employed public textual data in the evaluation, which consists of sequences of characters; each character is stored as a two-bye, Unicode character in Java. Experimental results show that our scheme with public verifiability only costs

preprocessing time 0.46s (only one-time cost during data outsourcing), search time 22μs, verification time 143μs, and proof size 2032 bytes for a verifiable pattern matching query with pattern length 200 on a text data with length 100,000 on a commodity PC. When the text size scales to 10 million, our scheme only costs preprocessing time 47s, search time 30μs, verification time 149μs, and proof size 2760 bytes for a verifiable pattern matching query with pattern length 200. The experimental results confirm that our scheme is highly efficient and scalable.

II. Related Work

Researchers have proposed several schemes to address verifiable pattern matching. We categorize these works into two groups: practical solutions and theoretical solutions. For the former, the proposed solutions are much faster and thus more suitable for practical applications. For the latter, the proposed solutions are often with theoretical, scientific interests, although having performance drawbacks. We discuss these two groups in more details.

A. Practical Solutions

The first practical solution is the work by Martel *et al.* [8]. They proposed a general scheme for verifiable query outsourcing on a remote server. This scheme was built on a core verifiable data structure - directed acyclic graph, which is so general that many practical data structures (e.g. array, list, tree) can be modeled as it is. Its general nature also results in large communication cost, which is normally a bottleneck of current cloud-based applications. The state-of-the-art solution is by Papadopoulos *et al.* [5]. This scheme has an optimal constant proof size. However, verification and preprocessing time are considerably large due to the big integer operations of the public key cryptography. Zhou *et al.* [11] also proposed a scheme to authenticate multi-pattern matching on fixed text. The search time is linear on the text length, which poses challenges for million-scale text pattern matching.

B. Theoretical Solutions

Researchers also proposed several theoretical solutions, which are also interesting although the efficiency is relatively lower. Catalano *et al.* [9] modeled the pattern matching problem as a vector identity testing problem, based on which the authors proposed verifiable pattern matching scheme using homomorphic MACs for polynomials. This scheme scans all the outsourced string; thus, the search time is linear. Zhou *et al.* [10] modeled pattern matching as polynomial identity tests. This work also employs sequential comparison to solve pattern matching. The performances of both schemes decline dramatically when the data scale up.

Some work has been done on *privacy-preserving* pattern matching. Wang *et al.* [12] transformed a pattern to a vector and then modeled pattern matching as vector similarity computations. The privacy is protected by encrypting the pattern-to-vector transformation. Liang *et al.* [13] modeled pattern matching using finite automata and protected the privacy using functional encryptions. Compared with our work here, such prior work concerns privacy under the semi-honest model, but not verifiability. It is worth noting that verifiability and privacy are two different, yet complementary, aspects of pattern matching outsourcing; both aspects are important [5], [9], [12]–[14].

Closely related to verifiable pattern matching, researchers also studied verifiable spatial databases and relational databases. Hu *et al.* [15] proposed a scheme for verifiable nearest neighbors query for a spatial database based on Voronoi diagram. Chen *et al.* [16] also studied the problem using R-tree and the power diagram for location based services. For relational databases, Riaz-ud-Din *et al.* [17] studied verifiable string matching for relational databases with text attributes. The scheme first employs a suffix array for searching (but not for authenticating), and then uses a Merkle hash tree to authenticate each matched rows, which incurs a linear communication cost in terms of each matched row. Zhang *et al.* [18] proposed an interesting scheme for verifiable SQL queries over outsourced cloud relational databases. This scheme supports expressive SQL queries. Chen *et al.* [19], [20] further proposed two schemes for efficient database updates based on vector commitments and MACs. Compared with verifiable pattern matching studied in this work, spatial database and relational database are inherently different: database is more structured than a long, consecutive text; thus, solutions for verifiable databases are not known to be effective for pattern matching over such text.

III. Proposed Basic Scheme

A. Threat Model, Assumptions, and Formalization

We model the server as untrusted. The server may cheat to save computation and communication costs; the cheating may be caused by malicious employees, or hardware/software faults. We assume the data owner and the data user are mutually trusted. This assumption sounds because practical application mainly considers the server as the untrusted entity. This work *mainly pursues fast verifiability*, similar to the state-of-the-art scheme [5]. This is because verifiability can guarantee faithful service quality; it is sufficient for typical applications, e.g. data publishing to third-party servers. We later also show how to extend the proposed scheme to further protect privacy.

A verifiable pattern matching scheme proceeds in three phases: preprocess phase, search phase, and verification phase respectively.

- Preprocess$(T, K) \rightarrow T'$. The data owner first initializes the verifiable pattern matching scheme. The data owner may (or may not) generate some secret key K with regard to a security level λ. Then the data owner preprocesses the data T by embedding authentication information to enable future pattern matching verifications. After preprocessing, data owner outsources the processed data T' to the server. The data owner also shares the secure key K(if any) with the data user.
- Search$(T', q) \rightarrow (\chi, \Gamma)$. After receiving a pattern matching query q from a data user, the server searches for the pattern in the outsourced data T' and then returns

the search result χ to the user. Additionally, the server also returns a proof Γ to prove that the returned result is correct.

- VERIFY$(q, K, \chi, \Gamma) \rightarrow \delta \in \{0, 1\}$. On receiving the pattern matching result (χ, Γ) from the server, the data user checks whether it is correct using the query q and the secret key K(if any). If it is correct, the data user accepts the result and outputs 1; otherwise rejects it and outputs 0.

1) Security Model: We formalize the cheating phenomenon first. Define a tuple (q^*, χ^*, Γ^*) as a *forgery* if either the following case holds: 1) q^* can be matched by the outsourced data, but the server using (χ^*, Γ^*) successfully proves that it cannot be matched, or it is matched in a wrong position; 2) q^* cannot be matched by the outsourced data, but the server successfully proves that it is matched by the outsourced data. Then, a server is cheating if the server can find such a forgery. Thus we have the following standard security definition.

Definition 1: Let Pr[Cheat] *be*

$$\Pr\left[\begin{array}{c} T' \leftarrow \mathsf{Preprocess}(T, K) \\ (\chi_i, \Gamma_i) \leftarrow \mathsf{Search}(T', q_i) \\ \delta_i \leftarrow \mathsf{Verify}(q_i, \chi_i, \Gamma_i; K) \\ i = 1, \cdots, \mathrm{poly}(\lambda) \end{array} : \begin{array}{c} \mathcal{A}(T', q_i, \delta_i) \\ \textit{finds a forgery} \\ (q^*, \chi^*, \Gamma^*) \end{array}\right] \quad (1)$$

which denotes the probability that an untrusted server $\mathcal{A}$ can find a forgery $(q*, \chi^*, \Gamma^*)$ after the server having learned about the verifiable pattern matching scheme by running polynomial times of the scheme. If Pr[Cheat] *is negligible, we say such a verifiable pattern matching scheme is secure.*

B. Notation and Suffix Array Based Pattern Matching

Let S denote a string of length n and $S = S[1]S[2] \cdots S[n]$ with a special ending symbol '\$'. Let $S[i, j]$ denote the substring of S which starts from position i to end position j inclusive, i.e. $S[i, j] = S[i]S[i+1] \cdots S[j], 1 \le i \le j \le n$. The suffix of the string S is $S[i, n], 1 \le i \le n$. For convenience, we denote $SF_i = S[i, n]$. Pattern matching is then to find a pattern string in a given string; its alphabet is the set of all possible characters in the string. In this work, we study substring patterns (possibly with wildcard characters '*' and '?').

Suffix array is a standard data structure [21]–[23], which stores all suffixes of a string in a sorted manner. Denote a suffix array of string S as SA. It is a numeric array; SA$[i]$ denotes the start position of the i-th suffix string, i.e. $SF_{\mathrm{SA}[i]}$ has rank i with regard to the ascending order of all suffix strings. Algorithm 1 details how to employ binary search in suffix array to find a pattern.

We explicitly list the matching rules for both match and mismatch as follows:

- Match Case: Once we found a suffix $SF_{\mathrm{SA}[i]}$ whose prefix matches the pattern P, there must exist another suffix $SF_{\mathrm{SA}[i]+m}$ such that $SF_{\mathrm{SA}[i]} = P||SF_{\mathrm{SA}[i]+m}$.
- Mismatch Case: When arriving at the end of the search, the search index range must be $(i, i+1)(1 \le i < n)$ or (n, n). There must exist the situation with $SF_{\mathrm{SA}[i]} < P < SF_{\mathrm{SA}[i+1]}$ or $SF_{\mathrm{SA}[n]} < P$.

Algorithm 1 Suffix Array Based Pattern Matching

Input: S, P: a string S with length n and a special ending symbol '\$', and a pattern P with length m
Output: MATCH, MISMATCH

```
1: Let (L, R) = (1, n).
2: while L ≤ R do
3:   M = (L+R)/2
4:   if P is a prefix of SF_SA[M] then
5:     Pattern is matched.
6:     return MATCH.
7:   else if P < SF_SA[M] then
8:     R = M − 1
9:   else
10:    L = M + 1
11:  end if
12: end while
13: return MISMATCH.
```

We later employ the above two cases to design our verifiable pattern matching scheme.

C. Understanding Verifiable Pattern Matching

To address verifiability, we need to understand the problem itself and its unique challenges. We handle verifiable pattern matching in two steps: first, we address verifiable substring pattern match; then, we generalize our solution for wildcard pattern matching.

Two challenges exist for verifiable substring pattern matching. The most prominent one is how to verify an arbitrary substring search in a long continuous text on a remote server. Specifically, we need to solve the following difficulties:

- the substring may appear in the text at arbitrary positions;
- the substring could be any-length long;
- the substring may not be matched by the text;
- the substring may be matched for multiple times.

After a substring can be verified, another challenge is how to, from the design perspective, make the verification as efficient and simple as possible. Generally, a scheme builds on public key cryptography is not efficient compared with that based on private key cryptography. A simple verifiable pattern matching scheme also should embed minimal information during the interactions between a user and a remote server.

D. Technical Preparation: Ordered Set Accumulator

Our scheme design employs a new mechanism which we call *ordered set accumulator* and describe in this subsection. This mechanism can solve the first challenge and the partial of the second challenge of verifiable substring pattern matching as discussed in the last subsection.

Specially, an ordered set accumulator takes an ordered set $\{a_1, a_2, \cdots, a_n\}$ as an input where $a_i \in \{0, 1\}^*$, then outputs a fixed-length aggregation value for each element recursively as follows:

$$ha_i = \mathsf{H}(a_i || ha_{i+1}) \quad (2)$$

where $1 \leq i \leq n-1$, $ha_n = \mathsf{H}(a_n)$, $\mathsf{H}(\cdot)$ is a cryptographic hash function, and '$||$' denotes string concatenation. We call the set $\{ha_i, 1 \leq i \leq n\}$ as ordered set accumulator; we also call each ha_i as an accumulator for convenience.

To verify a subset containing some consecutive elements $\{a_i, \cdots, a_j\}$ of the set, a remote server returns accumulators ha_i, ha_{j+1}. A user then checks whether

$$ha_i = \mathsf{H}(a_i||\mathsf{H}(a_{i+1}||\mathsf{H}(\cdots\mathsf{H}(a_j||ha_{j+1})\cdots))) \qquad (3)$$

holds. If verification passes, the consecutive subset is then authenticated. We employ this mechanism and the suffix array to enable verifiable pattern matching in the following subsections.

E. Our Approach

With all preparation work done, we now present our approach. Our high-level main idea to solve the verifiable substring pattern matching problem is summarized as *index construction* and *multiple-level authentication*, which we specify as follows.

1) Index Construction: We employ suffix array to index the outsourced text. The suffix array not only enables fast substring pattern matching, but also makes pattern matching easier to verify. Specifically, a suffix array sorts all the suffixes of the text to be outsourced in an ascending order. With suffix array, pattern matching is then transformed into a string concatenation problem for a pattern match case, or a string comparison problem for a pattern mismatch case. String concatenation and string comparison are essentially algebraic problems which are easier to verify the correctness.

We illustrate this idea using an example. Suppose the text to be outsourced is "suffix$" where '$' is a special character not contained in any string and denotes the end of a string. All its suffixes are "$", "x$", "ix$", "fix$", "ffix$", "uffix$", "suffix$". A suffix array sorts all the suffixes into an ascending order, i.e. "$", "ffix$","fix$", "ix$", "suffix$", "uffix$", "x$" where '$' is assumed to be smaller than any character. To find a pattern $p =$ "ff" for a match case, we can find two entries $\alpha =$ "ix$" and $\beta =$ "ffix$" in the suffix array such that $\beta = p||\alpha$ where $||$ denotes string concatenation. For a mismatch case, to find a pattern $p =$ "fg", we can find two *consecutive* entries $\alpha =$ "ffix$" and $\beta =$ "fix$" in the suffix array such that $\alpha < p < \beta$. For both cases, we only need to handle *algebraic* operations.

2) Multiple-Level Authentication: We employ the ordered set accumulator to authenticate all suffixes; the authentication further enables *extremely fast* string concatenation and string comparison verification. Specifically, we first build an ordered set accumulator for each suffix of the outsourced text. Later, for a match case verification, it is relatively easy to verify a string concatenation operation of two suffixes by using Eq. (3). For a mismatch case verification, we first find two consecutive suffixes, one of which is smaller than the pattern while the other of which is larger; we then verify whether the smaller- and larger- relation holds using the ordered set accumulator. Additionally, we authenticate the correctness of the accumulators to prevent forgery of such accumulators by the server. During the authentication of the accumulators, we also embed semantics (e.g. position information) to help later verifications.

Algorithm 2 Preprocess

Input: T: the text to be outsourced

Output: $\{\mathrm{SA}, \mathsf{H}, \mathrm{HA}(T), \mathrm{Auth}\}$: preprocessed text and authentication information

1: The data owner constructs a suffix array SA for T and chooses a collision-resistant hash function H.

2: Let $x_j = (j||-||T[j])$ represent the string consisting of an index, a connection character '-' not in any string, and the j-th character $T[j]$. Calculate the ordered set accumulator $\mathrm{HA}(T) = \{ha_i | 1 \leq i \leq n\}$ for the set $\{x_j, 1 \leq j \leq n\}$ where $ha_i = \mathsf{H}(x_i||\mathsf{H}(x_{i+1}||\mathsf{H}(\cdots\mathsf{H}(x_{n-1}||\mathsf{H}(x_n)))))$.

3: Let $t_i = \langle i, ha_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle, 1 \leq i \leq n$, which consists of the suffix array index and the accumulator value of a corresponding suffix. Authenticate each tuple using an authentication scheme, obtaining Auth_i for t_i. Denote the set of $\{\mathrm{Auth}_i\}$ as Auth.

4: **return** $\{\mathrm{SA}, \mathsf{H}, \mathrm{HA}(T), \mathrm{Auth}\}$

F. Detailed Scheme Design

We now show every detail of our scheme: Preprocess, Search, and Verify. Let T be the text padded with the special ending symbol '$', which is to be outsourced. Denote a suffix array as SA and a collision-resistant hash function as H. Denote the ordered set accumulator as an array $\mathrm{HA}(T)$ and its i-th element's authentication information as Auth_i; denote also Auth as the set of all elements $\{\mathrm{Auth}_i\}$. For authenticating $\mathrm{HA}(T)$, our scheme uses a general authentication scheme, i.e. either a MAC scheme or a Merkle hash tree scheme works. For example, readers may consider Auth_i as the authentication of the i-th accumulator in $\mathrm{HA}(T)$ using the classical HMAC scheme. Denote the length of the longest common prefix of two strings α, β as $\mathsf{lcp}(\alpha, \beta)$. Denote the queried pattern as P, its length as m, the returned search result as χ, and the proof as Γ.

Preprocess. The data owner runs Algorithm 2 to preprocess the text and then outsourced the processed text to the server. The preprocessing mainly contains two steps: indexing as in line 1 and authenticating as in lines 2 and 3. We first authenticate each suffix of the text in line 2 in an ordered set accumulator. The ordered set accumulator prevents any modification of the outsourced text. We note that we employ an enhanced semantics here: we embed the index information in each character. The embedded index helps in future verifications of pattern matching result.

We then further authenticate the ordered set accumulator to prevent the server modifying these accumulators in line 3. This is because we need authenticated accumulators to verify a pattern match/mismatch result.

At the end of preprocessing, the data owner outsources $\{T, \mathrm{SA}, \mathrm{HA}(T), \mathrm{Auth}\}$ to the server.

Search and Verify. We now discuss server search and data user verification simultaneously. After receiving a pattern

matching query from a data user, the server straight-forwardly employs the suffix array and the outsourced text to search the pattern. In order to make pattern matching result verifiable, the server needs to return a proof information together with a pattern matching result. We proceed in two parts: match and mismatch.

MATCH: For a match case, the server just proves (or the data user only verifies) that there are two suffixes α, β such that $\alpha = P||\beta$. The returned pattern matching result is in Eq. (4). It returns two entries of the ordered set accumulator to the data user, i.e. $ha_{\mathrm{SA}[i]}$ and $ha_{\mathrm{SA}[i]+m}$ where $SA[i]$ is the matched position of the queried pattern and m is the pattern length. Note that $ha_{\mathrm{SA}[i]}$ (as in the tuple $t_i = \langle i, ha_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle$) is implicitly contained in the authenticated Auth_i. For a match case, it holds that $ha_{\mathrm{SA}[i]} = P||ha_{\mathrm{SA}[i]+m}$. The data user, holding these three strings, just verifies whether this relation holds using Eq. (7).

Note that we only require $ha_{\mathrm{SA}[i]}$ be authenticated by Auth_i, but not $ha_{\mathrm{SA}[i]+m}$. This saves communication cost; but it is also secure. If somehow a malicious server forges $ha_{\mathrm{SA}[i]+m}$, Equation (7) cannot be satisfied; otherwise, this results in a collision for the chosen hash function. Please refer to Theorem 2 for security argument.

MISMATCH: For the mismatch case, the searched pattern is either lying between two consecutive suffixes, or greater the largest suffix. We refer the former as the *normal case* and the latter as the *boundary case*. The server just proves (or the data user only verifies) the two cases.

For the normal case, we have $SF_{\mathrm{SA}[i]} < P < SF_{\mathrm{SA}[i+1]}$ for some i. Correspondingly, the returned result is as in Eq. (5). It contains two parts: one is to prove $SF_{\mathrm{SA}[i]} < P$ as in the first two lines of Γ; the other is to prove $P < SF_{\mathrm{SA}[i+1]}$ as in the third and fourth lines of Γ.

Now we detail the idea to prove $SF_{\mathrm{SA}[i]} < P$. Let $SF_{\mathrm{SA}[i]} = \alpha||\gamma_1||beta_1$ and $P = \alpha||\gamma_2||beta_2$, where α is the longest common prefix of $SF_{\mathrm{SA}[i]}$ and P with length m_1, $\gamma_1||beta_1$ and $\gamma_2||beta_2$ are the remaining strings of $SF_{\mathrm{SA}[i]}$ and P, and γ_1 and γ_2 are single characters. The first and second lines of Γ prove that $\alpha||\gamma_1 = \mathrm{SA}[i]||T[\mathrm{SA}[i]+m_1]$ is a substring of the outsourced text; note that this is similar to a match case proof. Now the data user just needs to verify $\gamma_1 < P[1+m_1]$ to confirm the relation $SF_{\mathrm{SA}[i]} < P$. Using the same idea, the third and fourth lines of Γ show that $P < SF_{\mathrm{SA}[i+1]}$.

For the boundary case, it suffices to prove $SF_{\mathrm{SA}[n]} < P$. This is just a special situation of the normal case. Similarly, it is proved using Eq. (6).

1) Wildcards Pattern Matching: To extend the proposed scheme to support wildcard (i.e. * or ?) pattern matching, the idea is similar to [5]. That is to segment a wildcard pattern matching query into substring matching using the wildcards as delimiters to get a set of substrings. The server returns each substring query to the user in the verifiable manner. Finally, the user verifies the positions of the returned results, in addition to the verifiable substring pattern matching search. It is worth noting that both our scheme and the state-of-the-art work cannot support general regular expression pattern matching efficiently; we leave this as a future work.

Algorithm 3 Search

Input: $P, T, \mathrm{SA}, \mathrm{HA}(T), \mathrm{Auth}$: searched pattern and the outsourced processed text

Output: $\{\chi, \Gamma\}$: pattern matching result and correctness proof

1: The server uses text T and SA to search the pattern P.
2: **if** it is a *match* **then**
3: Let the matched position be $\mathrm{SA}[i]$. The server finds out Auth_i as an integrity proof of the tuple $t_i = \langle i, ha_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle$.
4: The server also finds out the accumulator value $ha_{\mathrm{SA}[i]+m}$ from $\mathrm{HA}(T)$.
5: The server returns

$$\begin{aligned} \chi &= \text{`match'}, \\ \Gamma &= \{i, \mathrm{SA}[i], ha_{\mathrm{SA}[i]+m}, \mathrm{Auth}_i\}. \end{aligned} \tag{4}$$

6: **else**
7: It is a *mismatch*. Two cases exist.
8: NORMAL CASE: $SF_{\mathrm{SA}[i]} < P < SF_{\mathrm{SA}[i+1]}$ for some i
9: Compute $m_1 = \mathsf{lcp}(SF_{\mathrm{SA}[i]}, P)$. Let the (m_1+1)-th character of $SF_{\mathrm{SA}[i]}$ be $T[\mathrm{SA}[i]+m_1]$. The server finds out the accumulator value $ha_{\mathrm{SA}[i]+m_1+1}$ from $\mathrm{HA}(T)$ and an integrity proof Auth_i for $t_i = \langle i, ha_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle$.
10: The server returns

$$\begin{aligned} \chi &= \text{`mismatch'}, \\ \Gamma &= \{\{i, \mathrm{SA}[i], m_1, T[\mathrm{SA}[i]+m_1], \\ &\quad ha_{\mathrm{SA}[i]+m_1+1}, \mathrm{Auth}_i\}, \\ &\quad \{i+1, \mathrm{SA}[i+1], m_2, T[\mathrm{SA}[i+1]+m_2], \\ &\quad ha_{\mathrm{SA}[i+1]+m_2+1}, \mathrm{Auth}_{i+1}\}\}. \end{aligned} \tag{5}$$

11: BOUNDARY CASE: $P > SF_{\mathrm{SA}[n]}$
12: Similarly, the server returns

$$\begin{aligned} \chi &= \text{`mismatch'}, \\ \Gamma &= \{\{n, \mathrm{SA}[n], m_1, T[\mathrm{SA}[n]+m_1], \\ &\quad ha_{\mathrm{SA}[n]+m_1+1}, \mathrm{Auth}_n\}\}. \end{aligned} \tag{6}$$

13: **end if**

G. An Example

We further illustrate our scheme using a detailed example. Let the text be "suffix". The owner appends the special ending character '\$' to the text to obtain $T =$ "suffix\$", and then the owner builds a suffix array for the text. We have

$$\mathrm{SA} = [7, 3, 4, 5, 1, 2, 6].$$

Later, the owner computes the ordered set accumulator value ha_i for each suffix SF_i of T; for instance

$$\begin{aligned} ha_7 &= \mathsf{H}(7-\$) \\ ha_6 &= \mathsf{H}((6-x)||\mathsf{H}(7-\$)) \\ ha_5 &= \mathsf{H}((5-i)||\mathsf{H}((6-x)||\mathsf{H}(7-\$))). \end{aligned}$$

We then get the ordered set accumulator

$$\mathrm{HA}(T) = \{ha_1, ha_2, ha_3, ha_4, ha_5, ha_6, ha_7\}.$$

Algorithm 4 Verify

Input: P, K, χ, Γ: searched pattern, secret key (for the employed authentication scheme if any), and returned pattern matching result

Output: ACCEPT, REJECT

1: MATCH CASE: Let $x'_j = (\mathrm{SA}[i] + j - 1|| - ||P[j]), 1 \leq j \leq m$.

2: The data user verifies: 1) whether $\langle i, ha'_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle$ is intact using Auth_i; 2) whether $SF_{\mathrm{SA}[i]} = P||SF_{\mathrm{SA}[i]+m}$ is correct using

$$ha'_{\mathrm{SA}[i]} = \mathsf{H}(x'_1||\mathsf{H}(x'_2||\mathsf{H}(\cdots \mathsf{H}(x'_m||ha_{\mathrm{SA}[i]+m})\cdots))). \tag{7}$$

3: If all verifications are correct, return ACCEPT, otherwise REJECT.

4: MISMATCH NORMAL CASE: First, verify whether the two indices are consecutive. Then verify $SF_{\mathrm{SA}[i]} < P$ according to the followings: 1) whether $T[\mathrm{SA}[i] + m_1] < P[m_1 + 1]$; 2) whether $\langle i, ha'_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle$ is intact using Auth_i; and 3) whether

$$ha'_{\mathrm{SA}[i]} = \mathsf{H}(x'_1||\mathsf{H}(\cdots \mathsf{H}(x'_{m_1}||\mathsf{H}(x_{\mathrm{SA}[i]+m_1}||ha_{\mathrm{SA}[i]+m_1+1}))\cdots)).$$

5: Second, similar to $SF_{\mathrm{SA}[i]} < P$, verify $P < SF_{\mathrm{SA}[i+1]}$ using three verifications.

6: If all verifications are correct, return ACCEPT, otherwise REJECT.

7: MISMATCH BOUNDARY CASE: It is similar to verify $SF_{\mathrm{SA}[i]} < P$ with $i = n$ specifically.

The owner continues to authenticate the tuples

$$t_i = \langle i, ha_{\mathrm{SA}[i]}, \mathrm{SA}[i]\rangle, \quad 1 \leq i \leq n$$

using HMAC with a secret key, obtaining the authenticated $\mathrm{Auth} = \{\mathrm{Auth}_i\}$. We note that Auth_i contains both the tuple t_i and its authentication. After the preprocessing, the owner sends the text T, suffix array SA, the ordered set accumulator $\mathrm{HA}(T)$, and Auth to the server; the data owner also shares the hash function H and the secret key of HMAC with the data users.

The server receives the preprocessed text from the data owner and then is ready for pattern matching query from data users. We discuss both a match and mismatch example, respectively.

A user Alice sends a pattern matching query $P_A =$ "ff" to the server, which is a matched pattern. The server uses the suffix array based query algorithm to search the result, getting the tuple $\langle i, \mathrm{SA}[i]\rangle = \langle 2, 3\rangle$, the accumulator value $ha_{\mathrm{SA}[i]+m} = ha_5$ from the accumulator set $\mathrm{HA}(T)$, and the authenticated $\mathrm{Auth}_i = \mathrm{Auth}_3$ for the accumulator value ha_3. Finally, the server sends back the result

$$\chi_A = \text{'match'},$$
$$\Gamma_A = \{2, 3, ha_5, \mathrm{Auth}_3\}$$

to Alice.

Alice receives her 'match' result and the corresponding proof. To verify correctness, Alice first computes the accumulator value $ha'_3 = \mathsf{H}((3 - f)||\mathsf{H}((4 - f)||ha_5))$, and then uses the authenticated Auth_3 to verify the correctness of ha'_3. If the verification process succeeds, the returned result from the server is correct and Alice accepts it. Otherwise, Alice rejects the result.

Another user Bob sends the pattern $P_B =$ "fg" for query, which is a mismatched pattern. The server runs the query algorithm to search the result, obtaining two tuples

$$\langle j, \mathrm{SA}[j]\rangle = \langle 2, 3\rangle$$
$$\langle j + 1, \mathrm{SA}[j + 1]\rangle = \langle 3, 4\rangle$$

for proving that

$$SF_3 < P_B < SF_4.$$

The server computes the longest common prefix for SF_3 and P_B, SF_4 and P_B, which is

$$m_1 = \mathsf{lcp}(SF_3, P_B) = 1$$
$$m_2 = \mathsf{lcp}(SF_4, P_B) = 1.$$

The server gets two characters

$$T[\mathrm{SA}[j] + m_1] = T[3 + 1] = \text{'f'}$$
$$T[\mathrm{SA}[j + 1] + m_2] = T[4 + 1] = \text{'i'}$$

from text T. The server gets the accumulator values $ha_{\mathrm{SA}[j]+m_1+1} = ha_5$, $ha_{\mathrm{SA}[j+1]+m_2+1} = ha_6$ from the ordered set accumulator $\mathrm{HA}(T)$, and the authenticated $\mathrm{Auth}_j = \mathrm{Auth}_3$ for accumulator value ha_3 and the authenticated $\mathrm{Auth}_{j+1} = \mathrm{Auth}_4$ for accumulator value ha_4. Finally, the server sends the result

$$\chi_B = \text{'mismatch'},$$
$$\Gamma_B = \{\{2, 3, 1, \text{'f'}, ha_5, \mathrm{Auth}_3\}, \{3, 4, 1, \text{'i'}, ha_6, \mathrm{Auth}_4\}\}$$

to Bob.

Bob also receives his 'mismatch' result and proof. Bob first checks the two indices 2 and 3 are continuous, and whether 'f' $< P_B[2]$ and 'i' $> P_B[2]$. Bob then computes the accumulator values, $ha'_3 = \mathsf{H}((3 - f)||\mathsf{H}((4 - f)||ha_5))$ and $ha'_4 = \mathsf{H}((4 - f)||\mathsf{H}((5 - i)||ha_6))$. Bob further uses the authenticated Auth_3 to verify the correctness of ha'_3 and the authenticated Auth_4 to verify the correctness of ha'_4. If all the verification succeeds, the returned result is correct and Bob accepts it. Otherwise, Bob rejects the returned result.

IV. Optimized Scheme

A. Optimal Search Time, Optimal Proof Size, and Public Verifiability

We first note that our scheme can be directly improved to achieve optimal $O(m)$ search time for a pattern with length m. Compared with our current scheme with search complexity $O(m \log n)$, it is a significant speedup.

Our idea is to assist the suffix array with two new data structures. Traditionally, a suffix array combined with a longest common prefix (LCP) array and a child table, which are two other arrays, can further improve pattern matching to $O(m)$ time [22]. Note that our basic scheme as in last section employs a suffix array as a *black box*; the two new arrays can also be embedded in a black-box manner. In addition, the basic scheme only depends on the embedded authentication information of the ordered set accumulators and the *detailed match/mismatch position* of a pattern. The two new arrays, i.e. the LCP array and the child table, do not influence the match/mismatch position of a query; thus, they can be directly incorporated into our scheme, achieving the optimal $O(m)$ search time.

We note that the proof size is $O(1)$ if the employed authentication scheme is a message authentication code scheme (e.g. HMAC). Thus, the proof size is optimal. However, the MAC scheme requires a secret key shared by the data owner and the data user. Thus, our scheme with a MAC is not *public verifiable*.

We may replace a MAC scheme with a Merkle hash tree scheme which requires no secret key, but just a public hash tree root value. In this case, any one with the root value can verify the correctness of the server's result, achieving public verifiability. A drawback is that the communication cost becomes $O(\log n)$.

We may also replace a MAC with any secure digital signature scheme. The communication cost is still $O(1)$; the resulted verifiable pattern matching scheme can also be verified by anyone publicly. But it requires a computational tradeoff; this is because digital signature is much less efficient than a hash function.

B. Fast Verification for Multiple-Occurrence Matches

Our scheme in Section III already supports single pattern matching. A pattern may also be matched by multiple substrings in a text, which we call multiple-occurrence match. This subsection proposes a scheme to enable fast verification for such case.

To verify multiple matches, a straight-forward approach is to apply single match verification multiple times. Suppose the cost for one match verification is t_{unit}. For a k-occurrence match, the complexity is $k \cdot t_{\text{unit}}$. Now we show this can be done much faster by *again* employing the longest common prefix (LCP) array.

Specifically, the LCP array is a numeric array with length n similar to the suffix array. It stores the length of the longest common prefix between adjacent sorted suffixes [22], [24], [25]. Denote an LCP array as LCP and the longest common prefix of two strings α and β as $\mathsf{lcp}(\alpha, \beta)$.

Algorithm 5 Fast Multi-Occurrence Match

1: Preprocessing. The data owner additionally constructs a LCP array LCP. Similar to the authentication of all suffixes, the data owner derives another semantics-embedded set $\{y_i = \langle i, \text{SA}[i], \text{LCP}[i]\rangle, 1 \le i \le n\}$ and computes its ordered set accumulator HALCP $= \{halcp_i, 1 \le i \le n\}$ where

$$halcp_i = \mathsf{H}(y_i || \mathsf{H}(\cdots \mathsf{H}(y_{n-1} || \mathsf{H}(y_n)))).$$

The data owner also authenticates each $halcp_i$ using the authentication scheme that guarantees the correctness of all the ha_i's in the basic scheme.

2: Search. For a multi-occurrence match, the server finds all the matches. To return a proof, the server gets the proof Γ_{first} for the first match, the proof Γ_{last} for the last match using our basic scheme.

3: In addition, the server returns all entries $\{i = \text{first}, \cdots, j = \text{last}\}$ of $\{y_i = \langle i, \text{SA}[i], \text{LCP}[i]\rangle\}$ between the first and last match, together with the authenticated accumulator value $halcp_{\text{first}}, halcp_{\text{last}}$ of the first and last entries.

4: Verify. The data user just verifies the first and last match using our basic scheme. The user further verifies: 1) whether the returned first and last LCP entries are smaller than the pattern length and other entries are greater or equal to pattern length, 2) whether the indices of LCP entries are between first and last, 3) whether the first and last entries pass the authentication verification, and 4) whether

$$halcp_{\text{first}} = \mathsf{H}(y_i || \mathsf{H}(\cdots \mathsf{H}(y_{j-1} || halcp_{\text{last}})))$$

is true. If all verifications pass, the data user accepts the result; otherwise, reject.

Then, $\text{LCP}[i] = \mathsf{lcp}(SF_{\text{SA}[i-1]}, SF_{\text{SA}[i]}), 1 < i \le n$; the first entry of LCP is not defined. For the example string "suffix$" in Section III-G, its LCP array is

$$\text{LCP} = [\perp, 0, 1, 0, 0, 0, 0].$$

We now explain our idea for fast multi-occurrence match verification as follows. The spirit is to transform string comparison verifications into integer comparison verifications. First, we observe that the suffixes corresponding to the substrings of multiple pattern matches are stored consecutively in the suffix array. To verify multiple matches, we only need to check the first match and the last match in the suffix array, and then check whether the longest common prefixes of two consecutive suffixes is greater or equal to the length of the queried pattern, using the LCP array. To ensure all matches are included, just check the matched-first and -last entries in the LCP array, which should be smaller than the pattern length. Second, the LCP array also needs to be *fast* authenticated. We can again use the ordered set accumulator, which further speedups the authenticated comparison of pattern length and the longest common prefixes of two consecutive suffixes.

Algorithm 5 details our optimized scheme for multi-occurrence match. We explain that we put the index in the set $\{y_i = \langle i, \text{SA}[i], \text{LCP}[i]\rangle, 1 \le i \le n\}$ because we want

to prevent the server from cheating, and that we put $\mathsf{SA}[i]$ in because the server needs to tell the data user where the pattern is matched in the outsourced text. Compared with the basic scheme, the optimized scheme only requires two single match verifications, plus three more light-weight authentication verifications. Thus, the optimized scheme features better performance.

C. Protecting Privacy

The proposed scheme well satisfies applications where verifiability is the only concern. When privacy is further required, we now outline how to preserve text privacy and search privacy.

Our observation is as follows: The privacy leakage comes from the outsourced text and the queried pattern. The verifiability part of the proposed scheme does not leak privacy; it is merely hash computations on the ordered set accumulators. Thus the main task is to protect the privacy of the outsourced text and the queried pattern, with the challenge that the cloud still has a way to search the queried pattern.

To solve this challenge, our main idea is twofold: 1) embed privacy preserving search in Algorithm 1 and 2) allow the user to interact with the cloud. Specifically, we first encrypt the text and search query using homomorphic encryption that supports homomorphic addition/substraction (e.g. consider the Paillier encryption scheme). To enable the binary search of the queried pattern, the user sends an encrypted pattern to the cloud; the cloud first compares the the queried pattern with the middle suffix string $SF_{\frac{n}{2}}$ using homomorphic evaluations of substraction and then sends the encrypted substraction result back to the user. By decrypting the result, the user knows whether to search up or down next and asks the cloud to perform recursive searches. In essence, this is an encrypted version of Algorithm 1. Once the matched/mismathed position is found, the cloud proceeds the same way as in our proposed scheme.

It is worthy noting that we trade-off performance of fast verifiability for privacy protection. Logarithmic rounds of interaction is required. This is expected because privacy does need additional processing. In future, we plan to find better approaches to guarantee verifiability and privacy simultaneously in fast manner.

V. Discussions

A. Security

Our scheme mainly builds its security on collision-resistance of cryptographic hash functions given a secure authentication scheme. In practice, various cryptographic hash functions exist, e.g. SHA256; we may just choose one to be plugged in our scheme. We prove the security in the following theorem.

Theorem 2: Given a secure authentication scheme (e.g. MAC, Merkle hash tree, digital signatures) which guarantees the correctness of the order set accumulator, our scheme is secure with respect to Definition 1 if the cryptographic hash function we employed is collision resistant.

Proof: We first show the proof for the single-match case. Suppose our scheme is not secure, then the server can find a forgery tuple (q^*, χ^*, Γ^*) according to our security definition. We employ the forgery to find a collision for the hash function. We distinguish the forgery into two cases.

1) Match Forgery Case: That is the queried pattern q^* cannot be matched by the outsourced text, but the server shows that it is matched using the proof (χ^*, Γ^*) where $\chi^* =$ 'match' and $\Gamma^* = \{i, \mathsf{SA}[i], ha_{\mathsf{SA}[i]+m}, \mathrm{Auth}_i\}$. The server may choose to cheat either on $i, \mathsf{SA}[i], \mathrm{Auth}_i$ or $ha_{\mathsf{SA}[i]+m}$.

If the cloud cheats on the former, the authentication scheme is not secure, which contradicts with our assumption. If the former is correct, we proceed to next. In this case $ha_{\mathsf{SA}[i]}$ is correct and we have

$$ha_{\mathsf{SA}[i]} = \mathsf{H}(x_1||\mathsf{H}(x_2||\mathsf{H}(\cdots\mathsf{H}(x_m||ha_{\mathsf{SA}[i]+m}))))$$

where $x_j = (\mathsf{SA}[i] + j - 1|| - ||p[j])$. Note that when outsourcing, we have

$$ha_{\mathsf{SA}[i]} = \mathsf{H}(y_1||\mathsf{H}(y_2||\mathsf{H}(\cdots\mathsf{H}(y_m||ha_{\mathsf{SA}[i]+m}))))$$

where $y_j = (\mathsf{SA}[i]+j-1||-||T[\mathsf{SA}[i]+j-1])$. These two are different inputs with the same cryptographic hash value. Thus, we find a hash collision.

2) Mismatch Forgery Case: That is the queried pattern q^* can be matched by the outsourced text, but the server shows that it is not matched during the search using the proof (χ^*, Γ^*) where

$$\begin{aligned}\chi^* &= \text{'mismatch'},\\ \Gamma^* &= \{\{i, \mathsf{SA}[i], m_1, T[\mathsf{SA}[i]+m_1],\\ &\quad ha_{\mathsf{SA}[i]+m_1+1}, \mathrm{Auth}_i\},\\ &\quad \{i+1, \mathsf{SA}[i+1], m_2, T[\mathsf{SA}[i+1]+m_2],\\ &\quad ha_{\mathsf{SA}[i+1]+m_2+1}, \mathrm{Auth}_{i+1}\}\}.\end{aligned}$$

Again the same analysis with the 'Match Forgery Case' applies here to find a hash collision. Specifically, the authenticated Auth_i ensures the first two lines of Γ^* pass the verification. Then the server needs to cheat on the last two lines of Γ^*. Note that Auth_{i+1} also guarantees the correctness of $\{i+1, \mathsf{SA}[i+1]\}$. Thus the server needs to cheat on $m_2, T[\mathsf{SA}[i+1]+m_2], ha_{\mathsf{SA}[i+1]+m_2+1}$ which are different from the outsourced text. Note that when outsourcing, ha_i is computed using the ordered set accumulator on the original text. The two above are different inputs into the hash function but have the same output ha_i, from which we have found a hash collision.

We now turn to multi-occurrence matches. Similarly, the first match and the last match cannot be forged; otherwise, there is a hash collision using the same argument above. For verifying the indices of the returned LCP entries, they are authenticated by the ordered set accumulator. If they can be forged, they together with original LCP entries comprise a hash collision.

Because the probabilities of breaking a secure authentication scheme and finding a hash collision are both negligible, the probability of breaking our scheme $\Pr[\mathsf{Cheat}]$ is also negligible. Combing the above arguments, the proof is completed. ■

TABLE I

THEORETICAL PERFORMANCE COMPARISON

	MAC scheme	**MHT scheme**	[5]
Storage	$O(n)$	$O(n)$	$O(n)$
Preprocessing	$O(n)$	$O(n)$	$O(n)$
Search	$O(m)$	$O(m + \log n)$	$O(m)$
Verification	$O(m)$	$O(m + \log n)$	$O(m \log m)$
Communication	$O(1)$	$O(\log n)$	$O(1)$

B. Theoretical Performance

We now analyze the theoretical performance of our verifiable pattern matching scheme and compare the theoretical performance with state-of-the-art work [5]. Table I lists a short summary where our scheme is instantiated using MAC and Merkle hash tree respectively. The theoretical comparison is further validated during experimental evaluations in Section VI.

1) Storage Cost: The storage size of the preprocessed data outsourced to server is $O(n)$. First, the data structures including the suffix array and the longest common prefix array are linear in the length of text. Second, the ordered set accumulator values are stored according to the suffixes of text and longest common prefix array, which also have size $O(n)$. Last, the authentication scheme (e.g. MAC, hash tree, etc.) is built for authenticating the ordered set accumulator values; thus it has size $O(n)$. In total, the storage cost of preprocessed data is $O(n)$. Comparing with [5], which also has $O(n)$ storage cost, the coefficients in the big O is smaller for our scheme. The main reason is that suffix array is more space efficient, and that our scheme employs the ordered set accumulator which is smaller than general accumulator based on bilinear groups over big integers in [5].

2) Computation Cost: For preprocessing in the data owner, building the suffix array, the LCP array, and the ordered set accumulators takes $O(n)$ time. Computing a MAC or building a Merkle hash tree also takes time $O(n)$. Thus, the total cost for preprocessing is $O(n)$. Although the preprocessing time in [5] is also $O(n)$, it uses heavy algebraic computations in a bilinear group; it costs much higher computation than our scheme which only involves cryptographic hash computation, as validated in later experimental evaluation.

The search time includes query time and proof generation time. The SEARCH algorithm in our optimized scheme costs $O(m)$ time to find out the occurrence of the queried pattern in text. It further takes $O(1)$ time to obtain the authentication information for the ordered set accumulator. In total, the search time in our optimized scheme is $O(m)$. It is similar to the $O(m)$ complexity of [5].

For the verification, the computation that accumulates $O(m)$ element for a pattern needs $O(m)$ hashing operation in our scheme. The verification of accumulator value using an authentication scheme can be done in $O(1)$ time in our optimized scheme. Thus the verification cost achieves $O(m)$ in our scheme. The verification time in [5] is $O(m \log m)$ and the verification involves many group element exponentiation computations. Thus, the cost of our scheme is much smaller than [5]. Note that verification is an on-line operation; thus a smaller cost is considerably better.

3) Communication Cost: The proof returned to a user consists of a constant number of ordered set accumulator values and its authentication information. Thus the proof size is $O(1)$ in our scheme with MAC, which is similar to the theoretical $O(1)$ as in [5]. However, the coefficients in the big O differ considerably; our coefficient is much smaller due to the succinct proof. Our later experimental results also confirm that our proof size is better.

C. Limitation

It is worth noting that our current scheme cannot support *efficient* data updates, i.e. adding, deleting, modifying data. This is because the underlying data structure to enable fast pattern matching substantially changes with data updates, which requires substantial trustworthy updates of its authentication on the server. This is a common problem both for our scheme and other schemes, including the state-of-the-art scheme [5]. We leave this as an ongoing work.

VI. PERFORMANCE EVALUATION

A. Methodology

We prototyped our scheme in both its basic and optimized version to empirically evaluate the performance of our scheme. We employ public human genome data set [26] in characters and open-source our code [27]. In our evaluation, we first compare our scheme with the state-of-the-art work; we then further give a more in-depth evaluation of our scheme due to its more practical efficiency and the resulted potentiality to be deployed.

1) Performance Measure: We mainly evaluate the computation, communication, and storage cost that are required by our scheme in three phases, i.e. Preprocess, Search, and Verify, respectively. We report an averaged experimental results to get a stable performance indicator. We also vary the text length and pattern length to understand the detailed performance of our scheme.

2) Implementation Detail: Our prototype is written in Java on a PC with Intel Core i7-4790k 4.00GHz CPU and 24G RAM; each character of the human genome data set is stored as a two-byte, Unicode character in Java. We set the JVM size as 16G. We employ a pubic library 'classmexer.jar' [28] to evaluate the memory size of Java objects, which are useful to evaluate the storage cost of our scheme. In our prototype, we simulate the data owner, the server, and the users in a PC. The data owner and data users share the secret key/root hash value of the Merkle hash tree for authentication. Communications between parties are implemented by passing parameters through functions. We employ SHA-256 as our cryptographic hash function. We instantiated our proposed verifiable pattern matching scheme both with a MAC and a Merkel hash tree scheme separately to authenticate the ordered set accumulators. We refer to the former as the MAC scheme while the later as the MHT scheme.

B. Comparison With the State-of-the-Art Scheme

We first compared our scheme with the state-of-the-art work in [5], which we refer to as the ST scheme. Since we were

TABLE II
STORAGE COST COMPARISON

Text Length(n)	**MAC**(MB)	**MHT**(MB)	**ST**(MB)
20,000	4.360	9.969	37.376
40,000	8.720	19.937	75.435
60,000	13.080	22.696	113.485
80,000	17.439	39.870	151.999
100,000	21.799	42.630	190.460

TABLE III
PREPROCESSING TIME COMPARISON

Text Length(n)	**MAC**(s)	**MHT**(s)	**ST**(s)
20,000	0.236	0.140	1064.288
40,000	0.310	0.245	2145.537
60,000	0.381	0.277	3278.859
80,000	0.476	0.437	4292.508
100,000	0.541	0.464	5362.205

TABLE IV
VERIFICATION TIME COMPARISON FOR SINGLE-MATCH ON 100,000-LONG FIXED TEXT

Pattern Length(n)	**MAC**	**MHT**	**ST**
200	102.747(μs)	143.109(μs)	7.647(s)
400	191.605(μs)	236.432(μs)	14.687(s)
600	282.714(μs)	323.501(μs)	21.740(s)
800	373.745(μs)	419.916(μs)	28.775(s)
1000	448.203(μs)	486.582(μs)	36.065(s)

TABLE V
VERIFICATION TIME COMPARISON FOR MISMATCH ON 100,000 LONG FIXED TEXT

Pattern Length(n)	**MAC**	**MHT**	**ST**
200	107.466(μs)	166.772(μs)	7.840(s)
400	206.498(μs)	251.532(μs)	14.850(s)
600	293.195(μs)	339.208(μs)	21.854(s)
800	398.935(μs)	454.934(μs)	29.157(s)
1000	492.178(μs)	523.698(μs)	37.768(s)

not able to get the source code of [5], we also implemented the scheme of [5] and open-sourced our code [29]. We used the well-accepted jPBC library to compute bilinear computations [30] for the ST scheme; specifically, we adopt type A curve with $rBit = 256, qBit = 1024$.

We observed that the results of our implementation of the state-of-the-art scheme [5] are close to the performance results reported in [5]; that is, our numbers when scaled are similar to the numbers of [5]. We explain more as follows: The original setup time of [5] are 99.6s for 10,000 long text, and 976.5s for 100,000 long text; in our implementation, the times are 1064.288s for 20,000 long text, and 5362.205s for 100,1000 long text. The scale ration is roughly 5. For verification which is the core performance indicator of a verifiable pattern matching scheme, the time cost of [5] is roughly 50ms for 200 long pattern, and 90ms for 400 long pattern; in our implementation, the times are 7.840s (7840ms) and 14.850s (14850ms) respectively. The ratio is also constant, roughly being 120. We also note that Java is generally slower than C++. Even in this case, the verification time in our proposed scheme is in the orders of milliseconds, which is orders-of-magnitude better than [5]. This really comes from theoretical confidence: our scheme only relies on hash functions while [5] heavily relies on paring operations and big integer arithmetic.

1) Storage Cost: The storage cost is incurred by the additional data structures and their authentication. We measure the cost 10 times and report the average performance of 10 tests. Table II shows the storage cost of the ST scheme and our scheme for texts whose length range from 20,000 to 100,000. The experimental results confirm that the storage cost of the ST scheme is larger than that in our scheme. The main reason is that the authenticated data in the ST scheme are a few bilinear group elements, and that the size of bilinear group elements is bigger than the hash values in our scheme.

2) Computation Cost: We now compare computation cost. We report an average measure of 10 tests for preprocessing time and 100 tests for search and verification time.

a) Preprocessing time: Table III lists the comparison. The experimental results are consistent with our theoretical analysis. The preprocessing time in our scheme is much smaller than that in the ST scheme. This is because the preprocessing computation in the ST scheme are mainly big integer operations of bilinear group elements. In contrast, our scheme only requires the cryptographic hash operations which are much simpler and highly more efficient.

b) Verification time: Tables IV and V show the experimental results on patterns with different length. We again confirm that our scheme are orders of magnitude faster. This reason is still the sharp efficiency gains of hash functions over big integer operations on bilinear curves.

We also tested multi-matches. We fix the text with 100,000 long and increase the pattern length from 2 to 10 in order to test the influence of occurrences on verification time. Table VI shows the comparison results. Again, our scheme is significantly faster than the ST scheme, as demonstrated in the table. Experimental results also show that with the increasing of occurrences, the verification time in both schemes increases due to the influence of occurrences.

c) Search time: We now compare the search time. We consider single-match and mismatch situations on fixed text and fixed pattern length as a case study. The search time contains the time to find a pattern in the text, and the time to find its proof information that convinces the correctness of the pattern matching results. Figure 2 displays the experimental results. Our scheme in general costs slightly smaller time. The reason is the ST scheme involves tree traversals while our scheme mainly contains direct array accesses, which is faster than tree traversals.

3) Communication Cost: Figure 3 displays the communication comparison. The cost is also an average result of 100 tests. The proof size in our scheme is much smaller than that in the ST scheme. This reason is that the proof items in our scheme are more succinct than that in ST scheme, and also that the bilinear group elements(part of the proof in the ST scheme) is much bigger than the cryptographic hash value in our scheme. The same observation also applies to multi-occurrence matches as in Table VII.

TABLE VI

VERIFICATION TIME FOR MULTI-MATCH ON FIXED 100,000 LONG TEXT IN OUR SCHEME AND ST SCHEME

Length Pattern	Avg Occur(MAC)	VerifyTime(MAC)(μs)	Avg Occur(MHT)	VerifyTime(MHT)(μs)	Avg Occur(ST)	VerifyTime(ST)(μs)
2	6995	3,072	7540	3,348	7890	757,589
4	584	357	557	407	579	825,523
6	45	41	47	95	50	898,166
8	7	14	5	71	4	959,174
10	2	12	2	69	1	1,039,978

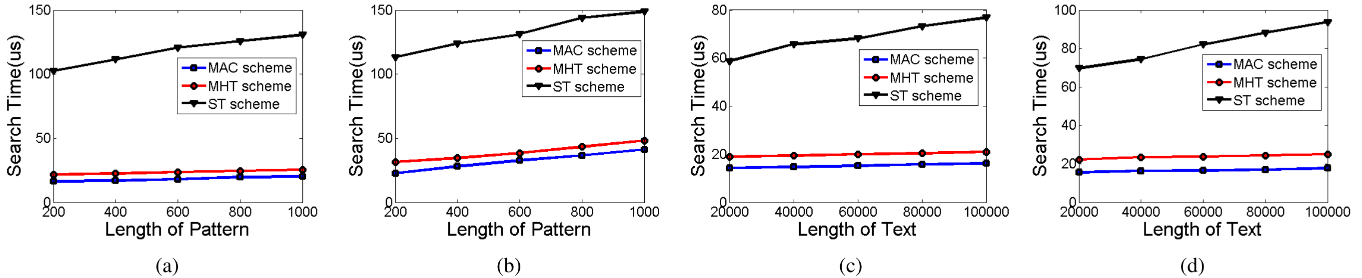


Fig. 2. Search time comparison for single-match and mismatch. (a) Single-match result for 100,000-long fixed text. (b) Mismatch result for 100,000-long fixed text. (c) Single-match result for 10-long fixed pattern. (d) Mismatch result for 10-long fixed pattern.

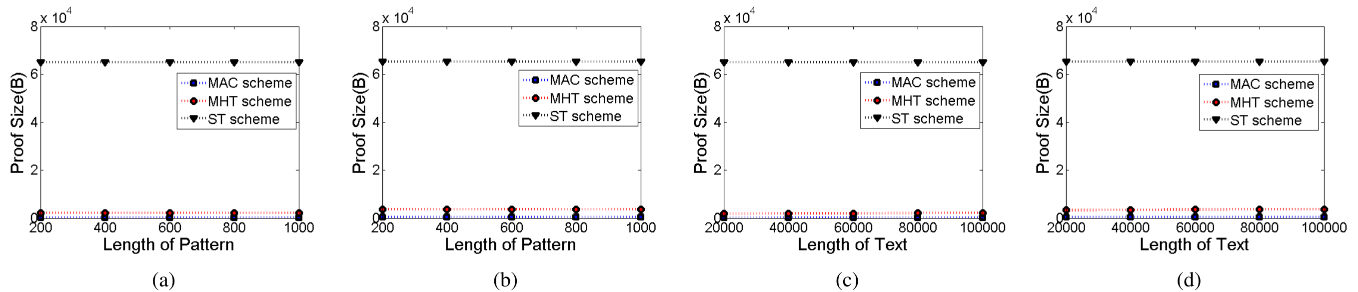


Fig. 3. Proof size comparison for single-match and mismatch. (a) Single-match result for 100,000-long fixed text. (b) Mismatch result for 100,000-long fixed text. (c) Single-match result for 10-long fixed pattern. (d) Mismatch result for 10-long fixed pattern.

TABLE VII

PROOF SIZE FOR MULTI-MATCH ON 100,000 LONG FIXED TEXT IN OUR SCHEME AND ST SCHEME

Length Pattern	Avg Occur(MAC)	ProofSize(MAC)(KB)	Avg Occur(MHT)	ProofSize(MHT)(KB)	Avg Occur(ST)	ProofSize(ST)(KB)
2	6995	196.246	7540	215.095	7890	380.945
4	584	16.750	557	19.574	579	88.503
6	45	1.650	47	5.280	50	67.352
8	7	0.579	5	4.123	4	65.533
10	2	0.439	2	4.034	1	65.387

4) Performance Comparison Summary: Our experimental comparison confirms that our scheme is orders of magnitude faster than the state-of-the-art work, especially when outsourcing the text, verifying the server's returned result. Our scheme also incurs smaller proof size. A remaining question is whether our scheme is scalable to larger data sets, which we investigate next.

C. Scalability Evaluation of Our Scheme

We further increase the length of the text from 2 million to 10 million to test the scalability of our scheme. Experimental results show that our scheme is also scalable; for detailed experimental data, please refer to our online supplemental material and source codes due to page limits.

D. Summary of Performance Evaluation

Experimental results confirm that the computation and communication cost for the MHT scheme and the MAC scheme are small; the storage cost is also acceptable in nowadays cloud computing environments. Our scheme is also scalable. The MHT scheme and MAC scheme have different strengths. The MHT scheme even does not require secret keys while the MAC scheme does need one. Thus MHT supports public verification. However, the performance for proof size of MAC is better. We therefore remark that applications may trade-off between performance and user application scenarios when employing the proposed verifiable pattern matching scheme.

VII. CONCLUSIONS

In this paper, we propose a verifiable pattern matching scheme. Our scheme mainly relies on cryptographic hash functions and thus is highly efficient. Our scheme also uses a modular-and-minimal design methodology such that the structure of our scheme is clear. We also embed minimal information for the parameters in our scheme to enable fast verifiable pattern matching. Our scheme further supports

public verifiability and efficient multiple-occurrence pattern matching. No secret key is needed in our scheme if desired by upper-layer applications. Our experimental results show that our protocol is orders of magnitude faster than the state-of-the-art work.

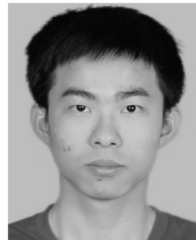
ACKNOWLEDGMENTS

The authors are grateful for the reviewers' insightful comments.

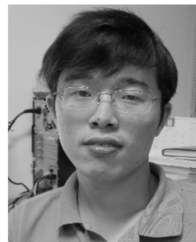
REFERENCES

[1] R. Sion, "Query execution assurance for outsourced databases," in *Proc. VLDB*, 2005, pp. 601–612.
[2] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proc. SIGMOD*, 2006, pp. 121–132.
[3] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally, "Database management as a service: Challenges and opportunities," in *Proc. ICDE*, 2009, pp. 1709–1716.
[4] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM Trans. Storage*, vol. 2, no. 2, pp. 107–138, 2006.
[5] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Practical authenticated pattern matching with optimal proof size," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 750–761, 2015.
[6] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.
[7] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable network function outsourcing: Requirements, challenges, and roadmap," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2013, pp. 25–30.
[8] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, May 2004.
[9] D. Catalano, M. Di Raimondo, and S. Faro, "Verifiable pattern matching on outsourced texts," in *Proc. SCN*, 2016, pp. 333–350.
[10] J. Zhou, Z. Cao, and X. Dong, "PPOPM: More efficient privacy preserving outsourced pattern matching," in *Proc. ESORICS*, 2016, pp. 135–153.
[11] Z. Zhou, T. Zhang, S. S. M. Chow, Y. Zhang, and K. Zhang, "Efficient authenticated multi-pattern matching," in *Proc. ASIACCS*, 2016, pp. 593–604.
[12] D. Wang, X. Jia, C. Wang, K. Yang, S. Fu, and M. Xu, "Generalized pattern matching string search on encrypted data in cloud systems," in *Proc. IEEE INFOCOM*, Apr. 2015, pp. 2101–2109.
[13] K. Liang, X. Huang, F. Guo, and J. K. Liu, "Privacy-preserving and regular language search over encrypted cloud data," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 10, pp. 2365–2376, Oct. 2016.
[14] Z. Fu, F. Huang, K. Ren, J. Weng, and C. Wang, "Privacy-preserving smart semantic search based on conceptual graphs over encrypted outsourced data," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1874–1884, Aug. 2017.
[15] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi, "Spatial query integrity with Voronoi neighbors," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 863–876, Apr. 2013.
[16] Q. Chen, H. Hu, and J. Xu, "Authenticating top-k queries in location-based services with confidentiality," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 49–60, 2013.
[17] F. Riaz-ud Din, R. Doss, and W. Zhou, "String matching query verification on cloud-hosted databases," in *Proc. 17th Int. Conf. Distrib. Comput. Netw.*, 2016, p. 17.
[18] Y. Zhang, J. Katz, and C. Papamanthou, "IntegriDB: Verifiable SQL for outsourced databases," in *Proc. ACM CCS*, 2015, pp. 1480–1491.
[19] X. Chen, J. Li, X. Huang, J. Ma, and W. Lou, "New publicly verifiable databases with efficient updates," *IEEE Trans. Depend. Sec. Comput.*, vol. 12, no. 5, pp. 546–556, Sep. 2015.
[20] X. Chen, J. Li, J. Weng, J. Ma, and W. Lou, "Verifiable computation over large database with incremental updates," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 3184–3195, Oct. 2016.
[21] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.
[22] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.
[23] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard, "Dynamic extended suffix arrays," *J. Discrete Algorithms*, vol. 8, no. 2, pp. 241–257, 2010.
[24] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. Annu. Symp. Combinat. Pattern Matching*, 2001, pp. 181–192.
[25] S. Gog and E. Ohlebusch, "Fast and lightweight LCP-array construction algorithms," in *Proc. Meeting Algorithm Eng. Experim.*, 2011, pp. 25–34.
[26] P. Gutenberg. (2016). *Human Genome Project, Chromosome 4*. [Online]. Available: http://onlinebooks.library.upenn.edu/webbin/gutbook/lookup?num=2204
[27] D. Wang. *Source Code for Secure Hashing Based Verifiable Pattern Match*. Accessed: Dec. 2017. [Online]. Available: https://sites.google.com/site/chenfeiorange/resource/VerifiablePatternMatching.zip
[28] N. Coffey. (2017). *Classmexer Agent*. [Online]. Available: http://www.javamex.com/classmexer/
[29] D. Wang. (2017). *Source Code for Verifiable Pattern Match of VLDB 2015*. [Online]. Available: https://sites.google.com/site/chenfeiorange/resource/VerifiablePatternMatchingST.zip
[30] A. de Caro and V. Iovino, "jPBC: Java pairing based cryptography," in *Proc. 16th IEEE Symp. Comput. Commun.*, Jun. 2011, pp. 850–855.

Fei Chen received the Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong. He joined the College of Computer Science and Engineering, Shenzhen University, China, as a Lecturer, in 2015. His research interests include information and network security, data protection, and privacy.



Donghong Wang is currently pursuing the master's degree in computer science and engineering with Shenzhen University, China. His research interests include information and network security, data protection, and privacy.

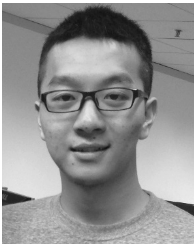


Ronghua Li received the Ph.D. degree from The Chinese University of Hong Kong in 2013. He is currently an Associate Professor with the Beijing Institute of Technology, Beijing, China. His research interests include graph data management and mining, social network analysis, graph computation systems, and graph-based machine learning.

Jianyong Chen received the Ph.D. degree from the City University of Hong Kong, Hong Kong, in 2003. He is currently a Professor with the College of Computer Science and Software Engineering, Shenzhen University. His research interests include artificial intelligence and information security. From 2004 to 2012, he was the Vice Chairman of the International Telecommunication Union-Telecommunication (ITU-T) SG17. He was an Editor of three recommendations developed in ITU-T SG17.

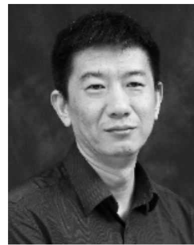
Zhong Ming is currently a Professor with the College of Computer and Software Engineering, Shenzhen University. He led four projects of the National Natural Science Foundation, and two projects of the Natural Science Foundation of Guangdong Province, China. His major research interests include Internet of Things and cloud computing. He is a member of the CCF Council.

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin in 2006. He is currently a Professor with the Department of Computer Science and Engineering, Michigan State University. His research interests focus on networking and security. He received the IEEE and IFIP William C. Carter Award in 2004, the National Science Foundation CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received the best paper awards from ICNP-2012, SRDS-2012, and LISA-2010. He has served as an Editor for the IEEE/ACM TRANSACTIONS ON NETWORKING. He is currently an Associate Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and the IEEE TRANSACTIONS ON MOBILE COMPUTING. He is also an Area Editor of *Computer Communications*.



Huayi Duan received the B.S. degree (Hons.) from the City University of Hong Kong, Hong Kong, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Computer Science. His research interests include network security and cloud security.

Cong Wang received the Ph.D. degree in electrical and computer engineering from the Illinois Institute of Technology, USA. He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His research has been supported by multiple government research fund agencies, including National Natural Science Foundation of China, Hong Kong Research Grants Council, and Hong Kong Innovation and Technology Commission. His current research interests include data and computation outsourcing security in the context of cloud computing, network security in emerging Internet architecture, multimedia security and its applications, and privacy-enhancing technologies in the context of big data and Internet of Things. He is a member of the ACM. He was a co-recipient of CHINACOM 2009, the Best Paper Award of the IEEE MSN 2015, and the Best Student Paper Award of IEEE ICDCS 2017. He received the President's Awards from the City University of Hong Kong in 2016. He has been serving as the TPC co-chair for a number of IEEE conferences/workshops.



Jing Qin received the Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong in 2009. He is currently an Assistant Professor with the Centre for Smart Health, School of Nursing, The Hong Kong Polytechnic University. He has authored over 150 papers in major journals and conferences in these areas. He has participated in over 10 research projects. His research interests include medical image processing, virtual/augmented reality for healthcare and medicine training, deep learning, visualization and human–computer interaction, and health informatics. He and his collaborators were a recipient of the Outstanding Paper Award at the International Simulation and Gaming Association 40th Annual Conference in 2009. He received the Global Scholarship Program for Research Excellence (CNOOC Grants) from CUHK in 2008, the Hong Kong Medical and Health Device Industries Association Student Research Award in 2009, the Best Paper Award in Medical Image Computing in International Conference on Medical Imaging and Augmented Reality 2016, the Champion award at The 3rd Hong Kong Innovation Day and Innovation Awards Competition, and the Medical Image Analysis-MICCAI'17 Best Paper Award.