DOI: 10.3969/j. issn. 1673-5188. 2016. S0. 002
http://www.cnki.net/kcms/detail/34.1294.TN.20160607.1841.002.html, published online June 7, 2016

Verification of Substring Searches on the Untrusted Cloud

Faizal Riaz-ud-Din and Robin Doss
(School of Information Technology, Deakin University, Burwood, VIC 3125, Australia)

Abstract

Ensuring the correctness of answers to substring queries has not been a concern for consumers working within the traditional confines of their own organisational infrastructure. This is due to the fact that organisations generally trust their handling of their own data hosted on their own servers and networks. With cloud computing however, where both data and processing are delegated to unknown servers, guarantees of the correctness of queries need to be available. The verification of the results of substring searches has not been given much focus to date within the wider scope of data and query verification. We present a verification scheme for existential substring searches on text files, which is the first of its kind to satisfy the desired properties of authenticity, completeness, and freshness. The scheme is based on suffix arrays, Merkle hash trees and cryptographic hashes to provide strong guarantees of correctness for the consumer, even in fully untrusted environments. We provide a description of our scheme, along with the results of experiments conducted on a fully-working prototype.

Keywords

substring search; query verification; cloud

1 Introduction

The paradigm shift from traditional, locally-hosted databases and infrastructure to their deployment on the cloud has provided a robust solution for those seeking to minimising costs whilst at the same time greatly enhancing flexibility. However, in spite of these benefits there are a number of areas of concern over the control of the data that gets outsourced, as well as the question of trust that arises when one hands over data and processing to a cloud service provider (CSP).

An elegant solution to this trust problem would be reducing requirements of trust in the relationship between the user and the CSP, and instead verifying the computation performed by the CSP and the authenticity of the data received by the user. This approach, known as query and data verification, attempts to provide data processing guarantees to consumers that cannot be falsified. Such schemes require the server to return a proof of correctness (known as a verification object or VO) with the results, which is used by the client to verify the correctness of the results.

This paper focuses on a substring query verification scheme for string matching queries against file-based data hosted on untrusted cloud servers. Substring queries match arbitrary substrings to larger strings. Research in the area of substring query verification has been scarce, with keyword search verification being more prevalent.

1.1 Our contributions

Our contributions in this paper may be summarised as follows:

- To the best of our knowledge, we provide the first existential substring query verification scheme that satisfies the properties of completeness, authenticity, and freshness.
- We show that our scheme detects both false positive and false negative query results, as opposed to the closest comparable substring matching verification scheme [1] that provides detection of false positives, but fails to provide proof for the detection of false negatives.
- Our scheme provides smaller VO sizes than the closest comparable substring matching verification scheme for large matches.

1.2 Motivation

There are commonly three types of substring searches that are executed on strings: existential, counting and enumeration queries. Typically, a string x is sought in a string S. Existential queries test whether x exists in S, counting queries that return the number of occurrences of x in S, and enumeration queries list all of the positions in S where x occurs. We focus on providing verification for existential queries in this paper, which we may also refer to simply as substring searches.

Proper substring searches provide greater flexibility and control with what is being searched for than keyword searches (where the search focuses on whole words rather than partial words). As such, one may be able to search for partial words, a combination of words where the text being matched begins or ends in the middle of a word, or in blocks of text that are composed entirely of characters without separators. Suffix trees, suffix arrays, finite automaton, and other techniques are used for realising proper substring searches. We focus on providing a verification scheme for these types of substring searches.

In the cloud environment, without a query verification mechanism in place, the only guarantee clients have of receiving correct answers to queries against cloud-hosted data is the trust between them and the CSP. However this may not always be sufficient, and definitely does not provide an absolute guarantee of correct query results.

1.3 Applications

Providing verification that a query has been correctly executed and that the received response with respect to the submitted query is correct is of tantamount importance for applications running on the cloud. With respect to existential query verification the following represents a small subset of applications that would benefit from our scheme:

- Querying large sets of biological data for specific occurrences of smaller DNA or RNA sequences as is required in sequence alignment algorithms. Our scheme may be used as a building block upon which to construct sequence alignment algorithms that provide proofs of correctness for alignment queries executed on remote cloud servers.
- Querying databases for partial matches of registration numbers, which may consist of alphanumeric and special characters. Our scheme can provide a basis upon which pattern matching queries may be verified for correctness when executed against databases stored on the cloud.
- Proving guarantees of correctness for queries issued against sensitive data such as medical records that may be stored on remote servers.
- Verification of answers to queries against text data created using agglutinative languages [2] where distinguishable words are not well-defined. Although languages such as English, where words are well-defined, may benefit from inverted indexes based on terms, and therefore from verification schemes based on inverted indexes, agglutinative languages may not fully benefit from such indexes, and our scheme provides a more robust method for providing search result verification against searches on such languages.

2 Related Work

Although the research in the areas of file verification at the block-level and byte level is plentiful [3]–[6], there has been relatively little work done in the area of existential substring query verification against single or multiple files.

The same can be said for substring query verification in the area of databases. Research into providing verification for queries based on numeric-based predicates [7]–[10] is plentiful with a number of papers published recently. A number of schemes based on Merkle hash trees (see Subsection 3.3) [7], [11]–[14], signature-chaining [15]–[17], and other approaches [18]–[22] have been presented in the literature largely with respect to numeric-based query verification.

However, substring queries look at a portion of the value in a given tuple attribute. Since any combination of the characters making up the value could be searched for, it is harder to find efficient verification schemes.

The work presented in [23], [24] provides substrings search verification. However, the study is based on inverted indexes, which are limited to providing keyword-level verification. As such, although they provide verification for substring searches within documents, they do so at a higher granularity than what is achieved by proper substring searches.

The scheme most closely related to ours is presented by Martel et al. in [1]. They propose a model called a Search Directed Acyclic Graph (DAG), which is used to provide methods to compute VO for a number of different data structures. One of these data structures is the suffix tree that is used to provide verification of proper substring searches. Their verification scheme uses hashing and techniques similar to that of Merkle Hash Trees to achieve verification of substring searches. However, although they provide proofs for detecting false positive queries, they do not do so for false negative queries. Additionally, their scheme, although efficient for small substring searches, would incur large bandwidth costs for longer substring searches. We address these two concerns and propose our method that also provides authenticity, completeness, and freshness.

3 Preliminaries

In this section we briefly describe the cryptographic primitives we use in our proposed scheme.

3.1 Secure Hash Function

A secure hash function, $h(x) \rightarrow d$, takes as input an arbitrary-length string, x, and produces a fixed-length hash digest, d. The one-way property of the secure hash function guarantees that given only a hash digest, d, it is infeasible to produce the original input string, x. The collision-free property of the secure hash function guarantees that given two distinct strings, x and y, their respective hash digests, d and e, will never be the same i.e. $h(x) \neq h(y)$. Commonly used secure hash schemes are MD5 [25], SHA1 [26] and SHA2 [27].

3.2 Digital Signature

A digital signature scheme consists of key generation, sign-

ing, and verification algorithms. The key generation algorithm produces a pair of related keys, known as public (pk) and private, or secret (sk) keys. The signing algorithm $sign(sk, m) \rightarrow m_{sk}$ takes as input the private key (sk) and a message (m) to produce a digital signature (m_{sk}). The verification algorithm $verify(pk, m_{sk}, m_r) \rightarrow (Y|N)$ takes as input the signature m_{sk}, the public key pk, and the received message m_r, and returns a Y or N to either affirm correctness of the received message m_r with respect to the original message m or to deny it. The most commonly used digital signature scheme is RSA [28].

3.3 Merkle Hash Tree

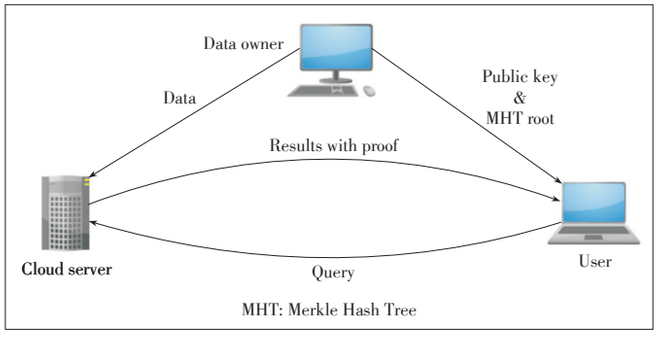
A Merkle Hash Tree (MHT) [29] is a binary tree that has as its leaves secure hash digests of data items. Each node in the tree is formed by concatenating the hashes of its child nodes, and then hashing the concatenated hashes. The root hash is signed with the owner's private key, and then can be validated using the owner's public key. The MHT allows verification of the order of the data items, their individual values, and a range of values. Verification involves retrieving the specified value, and then finding all nodes that are siblings of all nodes from the specified leaf to the root. The given data value is then hashed and combined with the other node hashes to regenerate the root, which is then checked against the original root to confirm or reject the value. MHT may also be implemented as B+ trees to improve efficiency.

4 Proposed Scheme

In this section, we describe our proposed scheme for the verification of substring queries.

4.1 System Model

A typical system model for our proposed substring query verification scheme is illustrated in **Fig. 1**. The data owner holds data that is queried by users. However, due to resource constraints or other reasons, the data owner wishes to outsource the data and query processing to the cloud for easier management. The cloud server therefore becomes a proxy for hosting data and query processing on behalf of the data owner.



▲**Figure 1. System model for the proposed substring query verification scheme.**

The data owner pre-processes the data, the final step of which includes generating an MHT root. The owner stores the root of the MHT and publishes the string data to the server, and publishes the public key within the context of a public key infrastructure.

A user submits substring queries to the server, which in response executes the query to obtain the query result. The query result is then sent back to the user together with a VO. The user then processes the query result and VO with a verification algorithm to either accept or reject the result.

4.2 Notation

Table 1 shows the definitions of pre-defined operations that are used in the rest of the paper. **Table 2** provides a list of the notations used in this paper to facilitate the descriptions of our proposed scheme.

4.3 Verification Properties

Verification schemes must provide solutions for at least three desirable properties authenticity, completeness, and freshness. These properties have been described in the literature previously [7], [16], [18], [30], but for the purpose of our scheme, we provide definitions of these properties below in the context of existential queries.

Consider a result R of an existential query Q that has been executed fully and correctly, in an uncompromised environment, on a string S that comprises substrings $\{S_1 \dots S_n\}$. Over time, S undergoes updates, causing its state to change from S^0, signifying the initial state at time 0, through to the current state at the current time, signified by S^c. The qualifying substrings in S that satisfy the predicates of Q are denoted $\{S_Q^1 \dots S_m^1\}$. We denote the existential function as F. We call

▼**Table 1. Pre-defined operations and their descriptions**

Operation	Definition
$BuildMHT(v_1,\cdots,v_n) \rightarrow mht$	Builds an MHT based on data values sorted in a specified order from v_1 to v_n. Outputs a new MHT, mht
$BuildSA(S) \rightarrow sa$	Builds a suffix array from string S. Outputs a new suffix array, sa.
$BuildVO(V,N) \rightarrow VO$	Builds a VO from a set of values (V), and a set of MHT nodes (N). Outputs a new VO
$IsSubstringOf(x,S,sa) \rightarrow <\{Y,N\}, i, \hat{i}_l, \hat{i}_r>$	Checks whether string x is a substring of string S using S's suffix array, sa. Outputs Y if x is a substring of S, otherwise outputs N. If x is a substring of S, sets i to the first index in sa where x prefixes $S_{suff(sa_i)}$ Otherwise, if x is not a substring of S, $\hat{i}_l$ and $\hat{i}_r$ are set to the neighbouring indexes in sa between which x would have been found, had it existed in sa.
$GetSinglePathSiblings(v,mht) \rightarrow N$	Traverses the given mht from the leaf v to the root, adding all siblings along that path to the set N
$GetRangePathSiblings(v_l,v_r,mht) \rightarrow N$	Traverses the given mht from each of the leaves v_l and v_r to the root, adding all siblings along the paths that are not ancestors of either leaf to the set N

▼Table 2. Notations used in this paper

Notation	Definition	Notation	Definition
DO	Data owner	x	Substring being sought
U	Data user	R	Root of an MHT
CS	Cloud server	R_{DO}	Root signed with DO's secret key
S	String	R_{DO}	DO's secret key
$\|S\|$	Size of string S	R_{DO}	DO's public key
S_i	Character at position i in S	L_i	Leaf I of an MHT
$S_{i\cdots j}$	Substring of S from S_i to S_j	VO	Verification object
$S_{suff(i)}$	Suffix of S starting at i	Q	A query
sa_S	Suffix array of S	QR	A query result
$sa(i)$	Element i in sa		

R the correct, uncompromised, unaltered result of Q. In the case where Q is issued remotely to D, which resides in a possibly untrusted environment, we consider the authenticity, completeness and freshness of the final result R' received by a remote user as follows.

Definition 4.1: Authenticity in our scheme means that R' is a result of executing F only on the substrings of the uncompromised string (i.e. that was created by the data owner). Specifically, authenticity of an existential query result R' is satisfied when

$$R' = F\left(\left\{S'_1 \cdots S'_k\right\}\right) \wedge S'_i \in \left\{S'_1 \cdots S'_k\right\} : S'_i \in (S^0 \cup \cdots \cup S^C).$$

Definition 4.2: Completeness in our scheme means that R' is a result of executing F on the same number of substrings as that executed by a correctly executing Q on an uncompromised string. Specifically, completeness of an existential query result R' is satisfied when

$$R' = F\left(\left\{S'_1 \cdots S'_k\right\}\right) \wedge \left|\left\{S'_1 \cdots S'_k\right\}\right| = \left|\left\{S^Q_1 \cdots S^Q_n\right\}\right|.$$

Definition4.3: Freshness in our scheme means that R' is a result of executing F on the most recently updated version of the uncompromised string. Specifically, freshness of an existential query result R' is satisfied when $R' = F\left(\left\{S'_1 \cdots S'_k\right\}\right) \wedge \left|\left\{S'_1 \cdots S'_k\right\}\right| \in \left\{S'_1 \cdots S'_n\right\} : S'_i \in S^C$.

4.4 Suffix Arrays

Let S be a string composed of characters from a set Σ of fixed sized, finite ordered alphabets. The length of S is denoted by n. \$ specifies a special end-of-string marker, which is smaller than all alphabets in Σ, but which does not occur in S. $S[i]$ denotes the index of the ith character in S.

The suffix array sa of the string S is an array of length $n+1$, where the elements in the array are unique indexes in $S \mid \$$, and are ordered lexicographically based on the suffixes of S, where each element points to a different suffix as indicated by its indexing value.

Table 3 shows an example of the suffix array for the string *'aardvark'*. The end-of-string marker (\$) is appended to the string before the suffix array is constructed, and has the smallest value out of all the alphabets in the string, resulting in its being sorted to the first element. The remaining suffixes' indexes are placed in lexicographical order in the suffix array.

4.5 Assumptions

We assume the following for the correctness of our scheme:

1) DO's public key has been obtained by U, possibly through a secure channel or reliable public-key infrastructure.
2) DO's secret key has not been compromised.
3) The public key encryption scheme used is secure under appropriately specified parameters.
4) It is infeasible to find collisions in the secure hash function that is used as a basis for the hash chaining and MHT generation procedures.

4.6 Hash Chains with Sequential Indexing

We introduce the concept of hash chains with sequential indexing (HCSI) as a building block for our scheme. The HCSI is essentially a hash chain with each link on the hash chain being tagged with a sequential identifier.

Definition 4.4: The HCSI is defined recursively as follows:

$$HCSI(S_i) = \begin{cases} h(i \,//\, S_i) & if\ i = |S| \\ h(i \,//\, S_i \,//\, HCSI(S_{i+1})) & if\ i < |S| \\ null & if\ i > |S| \end{cases}$$

The HCSI allows the specification of four variables that are useful for our scheme $\alpha, \beta, S_\beta, \gamma$, where α and β are indexes in S, S_β is the character at index β in S, and γ is a hash digest.

Definition 4.5: $\bar{x}$ is a prefix of $S_{suff(sa(i))}$ that is minimally unmatched to x, s.t. $\bar{x}_{0\cdots k-1} = x_{0\cdots k-1} \wedge \bar{x}_k \neq x_k \wedge k \in \{0 \cdots |x|\}$.

Definition 4.6: α refers to the position at the head of the HCSI, and is defined in the context of the suffix that it is associated with, as follows: $\alpha(S_{suff(i)}) = i$. In other words, α is the position in the HCSI that corresponds to the first character in $S_{suff(i)}$, and has the same value as $sa(i)$.

▼Table 3. Suffix array for the string *'aardvark'*

SA index	SA value	Resulting suffix
0	9	\$
1	1	aardvark \$
2	2	ardvark \$
3	6	ark \$
4	4	dvark \$
5	8	k \$
6	3	rdvark \$
7	7	rk \$
8	5	vark\$

Definition 4.7: β refers to the position of the last matching character in $S_{suff(i)}$ when x is a prefix of $S_{suff(i)}$, otherwise it refers to the first non-matching character in $S_{suff(i)}$ if x is not a prefix of $S_{suff(i)}$. In the case where x prefixes $S_{suff(i)}$, $\beta = i + |x| - 1$, otherwise if x does not prefix $S_{suff(i)}$, $\beta = i + |\bar{x}| - 1$.

Definition 4.8: S_β is the character in S at position β. If x is a prefix of $S_{suff(i)}$, $S_\beta = x_{|x|}$, otherwise $S_\beta = \bar{x}_{\bar{x}}$

Definition 4.9: γ refers to the first hash digest in the HCSI occurring after β, and is defined as $\gamma(\beta) = HCSI(S_{\beta+1})$.

By utilising the HCSI variables as given above, we are able to minimise the hash operations performed by the user from $|S_{suff(i)}| + 1$ hashes to at most $|x| + 1$ hashes, where $|x| < |S_{suff(i)}|$.

The sequential identifiers allow us to reduce the number of hashing operations to be performed on the user's end, and also reduces the communication cost by sending to the user only those hashes in the chain that the user needs to perform verification. **Algorithm 1** shows how the reconstruction may be realised.

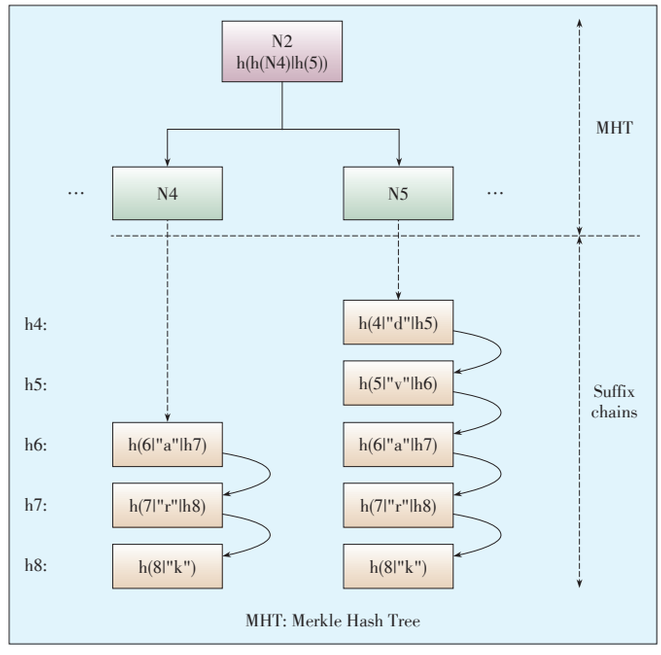
Algorithm 1: HCSI Reconstruction

Input: $x, \alpha, \beta, S_\beta, \gamma$
if $!(S_\beta)$ **then**
  /* Matching reconstruction */
    $\beta' \leftarrow \alpha + |x| - 1$; $S'_\beta \leftarrow x_{|x|}$; $x' \leftarrow x$;
else
  if $(\beta \ \alpha + 1 \leqslant |x|)\ (x_{\beta-\alpha+1} \neq S_\beta)$ **then**
    /* Non-matching reconstruction */
    $\beta' \leftarrow \beta$; $S'_\beta \leftarrow S_\beta$; $x' \leftarrow x_{1\cdots\beta-\alpha} // S_\beta$;
  else
    /* Invalid β and/or S_β */
    reject;
  end
end
$hcsi \leftarrow h(\beta', S'_\beta, \gamma)$;
for $i \leftarrow \beta' - 1$ **down-to** α **do**
  $hcsi \leftarrow h(i, x'_{i-\alpha+1}, hcsi)$;
end

4.7 Scheme Outline

Our intuition is to leverage MHTs to act as verification structures for suffix arrays. In essence, we build an MHT on top of a suffix array (**Fig. 2**), and then allow query results to be passed to the user along with VOs as proofs for the result. The user then verifies the result using the VO.

We define five phases for the implementation of our scheme: Setup Phase, Query Phase, Query Response Phase, Verification Phase, and Update Phase.



▲**Figure 2. An example of the sequenced hash chain of the suffixes '*ark*' and '*dvark*', and their corresponding leaves in a partially illustrated MHT.**

1) Phase 1: Setup

Data owners initiate the scheme by firstly generating a suffix array from the string or text file that their wishes to make available for querying. They then build HCSI digests over the suffixes in the string. The HCSI digests for each suffix are then ordered according to the suffix array indexes and an MHT is constructed over them. The data owners then sign the MHT root with their private keys and upload the string to the server. They also transmit their public keys to the users. They may optionally discard the string, the suffix array, and the MHT. **Algorithm 2** illustrates this phase.

Algorithm 2: Setup

Input: S, sk_{DO}
$sa \leftarrow BuildSA(S)$;
foreach $sfx \leftarrow \left(S_{suff(sa_s(1))} \cdots S_{suff(sa_s(|S|))}\right)$ **do**
  $v_i \leftarrow HCSI(sfx)$;
end
$mht \leftarrow BuildMHT(v1, ..., vn)$;
$R_{DO} \leftarrow \text{sign}(sk_{DO}, R)$;
Upload(S, R_{DO});
/ The following is optional */*
Delete(S, sa, mht);

Fig. 3 shows an example of the MHT constructed from the HCSI digests for the string '*aardvark*'. We will make use of Fig. 3 in running examples with the descriptions of the upcom-

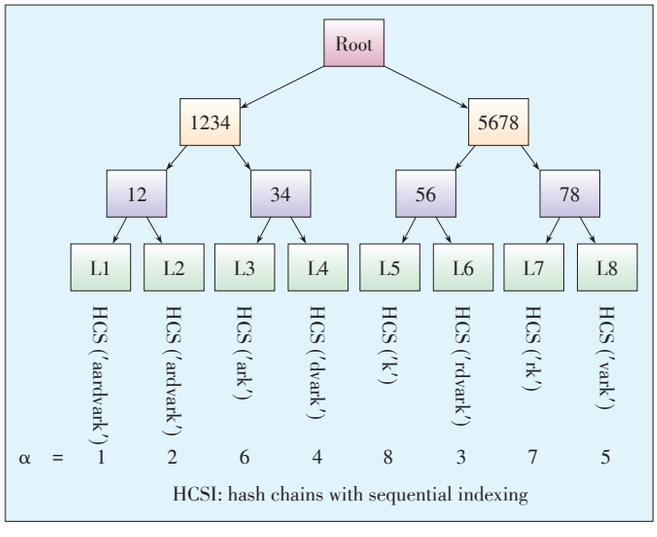
ing phases to provide some intuition as to how the scheme works.

2) Phase 2: Query

A user U submits to CS a query Q to check if substring x exists in string S. In our running example, the user submits a two different queries (to illustrate both positive and negative verification): '*%dva%*' and '*%are%*'.

3) Phase 3: Query Response

The cloud server receives a query from the user, to check for the existence of a substring in the stored string. The server constructs the HCSI digests on the suffixes of the stored string, and then proceeds to construct an MHT on the HCSI digests. This is identical to the data owner's processing in Phase 1. The server then searches for the substring in the suffix array. The result of the search returns one position from the suffix array if the substring was found, otherwise it returns two positions. If the substring was found, then the position returned is that of the matching suffix i.e. the substring is a prefix of the suffix at that position. If the substring was not found, then the two positions returned will be immediate neighbours. The first is the position of the suffix that is lexicographically smaller than the substring being sought, and the second is the position of the suffix that is lexicographically bigger. For each suffix returned, β is then calculated as shown in Definition 4.7. If the substring was found, the result is Y, otherwise it is N. The result is sent back to the client with the following verification data for the position(s) returned from the suffix array search: the position itself (i), the value of the suffix array at that position (α), the HCSI digest at position $\beta + 1$ (γ), and the verification path of the leaf for the corresponding position to the root of the MHT. In the case of a non-matching result, the first non-matching character position in the string (β), and the character at position β (S_β) is also sent back to the user. This phase is shown in **Algorithm 3**.



▲**Figure 3. An example of the MHT constructed from the HCSI digests for the string 'aardvark'.**

Algorithm 3: Query Response

Input: $Q(x)$
$sa \leftarrow BuildSA(S)$;
foreach $sfx \leftarrow \left(S_{suff(sa_S(1))} \cdots S_{suff(sa_S(|S|))}\right)$ **do**
$v_i \leftarrow HCSI(sfx)$;
end
$mht \leftarrow BuildMHT(v_1, \cdots, v_n)$;
$r \leftarrow IsSubstringOf(x, S, sa)$;
if $r.Y$ **then** /* substring was found */
$\alpha \leftarrow sa(i)$; $\gamma \leftarrow HCSI(S_{\alpha+|x|})$; $V \leftarrow \{i, \alpha, \gamma\}$;
$N \leftarrow GetSinglePathSiblings(v_i, mht)$;
else/* substring was not found */
$\alpha_l \leftarrow sa(\hat{i}_l)$; $\beta_l \leftarrow \alpha_l + |\bar{x}_l| - 1$; $\gamma_l \leftarrow HCSI(S_{\alpha_l+|\bar{x}_l|})$;
$\alpha_r \leftarrow sa(\hat{i}_r)$; $\beta_r \leftarrow \alpha_r + |\bar{x}_r| - 1$; $\gamma_r \leftarrow HCSI(S_{\alpha_r+|\bar{x}_r|})$;
$V \leftarrow \{\hat{i}_l, \alpha_l, \beta_l, S_{\beta_l}, \gamma_l, \hat{i}_r, \alpha_r, \beta_r, S_{\beta_r}, \gamma_r\}$;
$N \leftarrow GetRangePathSiblings(v_l, v_r, mht)$;
end
$VO \leftarrow BuildVO(V, N)$;
$Respond(r, VO)$;

To generate the VO for the query '*%dva%*' in our running example, the server searches for the prefix '*dva*' in the suffix array, and finds the corresponding match in L4 (Fig. 3). The server sets $\alpha = 4$, $\gamma =$ HCSI('*rk*'), and chooses the verification path for L4, corresponding to leaf L3, and internal nodes 1, 2, 5, 6, 7, and 8. α, γ. The verification path is then sent along with the response of the query (Y) to the user. To process the query '*%are%*', the server searches for the prefix '*are*' in the suffix array. The prefix is not found, so the neighbouring suffixes that are lexicographically less than and greater than '*are*' are selected, corresponding to leaves L2 and L3. The server sets $\alpha = 2$, $\beta = 4$, $S_\beta =$ 'd', and $\gamma =$ HCSI('*vark*') for L2, and $\alpha = 6$, $\beta = 8$, $S_\beta =$ '*k*', and $\gamma =$ *null* for L3. The server then chooses the verification path for L2 and L3, which is leaves L1 and L4, and internal node 5, 6, 7, and 8. The HCSI variables and verification path is finally sent to the user.

4) Phase 4: Verification

Upon receiving the query response from the server, the user ensures that a proof has been provided, otherwise he rejects the response. The user retrieves the latest root from the data owner, and verifies the root using the owner's public key. If the query response from the server was Y, he reconstructs the HCSI for the leaf at position i in the MHT using the reconstruction algorithm (Algorithm 1). He then uses the reconstructed leaf in conjunction with the verification path (sent by the server), to generate the MHT root. He compares the generated root with the root from the data owner. If the two roots match, he accepts the results as being correct, otherwise the result is rejected. If the query response was N, the user firstly checks that the

positions of the two suffixes returned by the server are neighbouring i.e. the left suffix position is one less than the right suffix position. This is to ensure that the server has not returned two suffixes which have suffixes in-between them, one or more of which may be suffixes that are matches for the substring. If the two positions are not neighbouring, the result is rejected. The user then constructs $\bar{x}$ for each position, by using the first $\beta-\alpha$ characters from the query substring, and appending S_β to it. Doing this allows him to reconstruct the partial suffixes for each position to the character that is the first non-matching character when compared to the query substring. He confirms that each is lexicographically smaller and larger than the query substring. If this is not the case, the result is rejected. This allows the user to ensure that the server has not simply returned two arbitrary neighbouring positions that do not in fact lexicographically border the query substring, thereby not allowing the user to confirm that the position at which the substring may be found but in fact doesn't exist. He then generates the HCSI digests for each suffix position using i, α, β, S_β, and γ (Algorithm 1). He uses the reconstructed leaf for each suffix position in conjunction with the verification path (sent by the server), to generate the MHT root. He then compares the generated root with the root from the data owner. If the two roots match, he accepts the results as being correct, otherwise the result is rejected. The verification process is outlined in **Algorithm 4**.

Algorithm 4: Query verification

Input: QR, VO
Retrieve R_{DO} from DO;
if *verify*(pk_{DO}, R_{DO}) == *false* **then** /* R_{DO} verification failed */
    reject;
end
if QR.*Y* **then** /* substring found */
    $h \leftarrow$ *ReconstructHCSI*(VO);
    $R \leftarrow$ *GenerateMHTRoot*(VO,h);
else/* substring not found */
    $h_l \leftarrow$ *ReconstructHCSI*(VO);
    $h_r \leftarrow$ *ReconstructHCSI*(VO);
    $\bar{x}_l \leftarrow x_{0\cdots\beta_l-\alpha_l} // S_{\beta_l}$; $\bar{x}_r \leftarrow x_{0\cdots\beta_r-\alpha_r} // S_{\beta_r}$;
    if $(\bar{x}_l \geq x) \mid (\bar{x}_r \leq x)$ **then**
        reject;
    end
    $R \leftarrow$ *GenerateMHTRoot*(VO, h_l,h_r);
end
if $R \neq R_{DO}^{-1}$ **then**
    reject;
end

In our running example, U receives the response Y to the query *'%dva%'*. To verify the correctness of the response, he processes the VO, also from CS, as follows: he regenerates L4 using the query *'dva'*, α and γ, by calculating h(α ‖ *'d'* ‖ h(α + 1 ‖ *'v'* ‖ h(α + 2 ‖ *'a'* ‖ γ))) = HCSI(*'dvark'*). He then regenerates the MHT root using the verification path provided by CS, and the checks against the signed root are provided by DO to ensure the generated root is correct. To verify the correctness of the second query, U runs through a similar process, but in this case, he additionally uses S_β to regenerate leaves L2 and L3. This is because the query literal *'are'* only partially matches the suffixes corresponding to leaves L2 and L4, and S_β for each suffix allows U to correctly construct each individual leaf using both part of the query and S_β. After reconstructing the leaves, he uses the verification path to reconstruct the root and checks against DO as a final step.

5) Phase 5: Update

In order to facilitate updates, the data owner simply executes the setup phase with a newer version of string S. This would generate a new MHT root that would then be used by the user to verify queries on the new string S.

5 Asymptotic Performance Analysis

We provide a brief outline of the space and time complexities achievable for both our scheme.

Table 4 shows a comparison of the complexities for the DAG scheme proposed in [1] and our scheme. The DO pre-processing phase in our scheme incurs a quarter of the storage cost of that incurred by the DAG scheme. This is not surprising as the underlying suffix arrays used in our scheme has a similar advantage over suffix trees in general. Although this advantage is in the constant factor, it in fact is a considerable advantage, and can mean the difference between a practically feasible or non-feasible solution. A similar advantage is incurred in the server query response phase, due to the fact that the server goes through a similar proof construction phase initially as that performed by the DO pre-processing phase. The VO size is small for the DAG scheme, but it increases to n as m approaches n. However, with our scheme, the size is always constantly relative to log n regardless of the size of m. This means

▼**Table 4. Space and time complexities for our scheme**

Measure	Reference [1]	Our scheme
Detect false +ve	Y	Y
Detect false -ve	N	Y
Technique used	DAG, compacted suffix tree and Hashing	MHT and suffix arrays
DO preproc.	$O(20n) + O(2n-1)H$	$O(5n) + O(2n-1)H$
Server qry resp.	$O(20n) + O(3m) + O(2n-1)H$	$O(5n + (m+2)\log n) + O(2n-1)H$
User verification	$O(m+k)H$	$O(\log n + m)H$
VO size	$O(m+k)$	$O(3 + 2\log n)$

In string searching theory, research papers provide asymptotic constants due to their impact on practical algorithms, and thus we also include them to allow greater precision for others when comparing our results to other work.
MHT: Merkle Hash Tree DAG: Directed Acyclic Graph

that the size of the VO under large m is smaller in our scheme by a log factor.

6 Empirical Evaluation

In this section, we evaluate the experiments conducted on a prototype of our proposed scheme in this section.

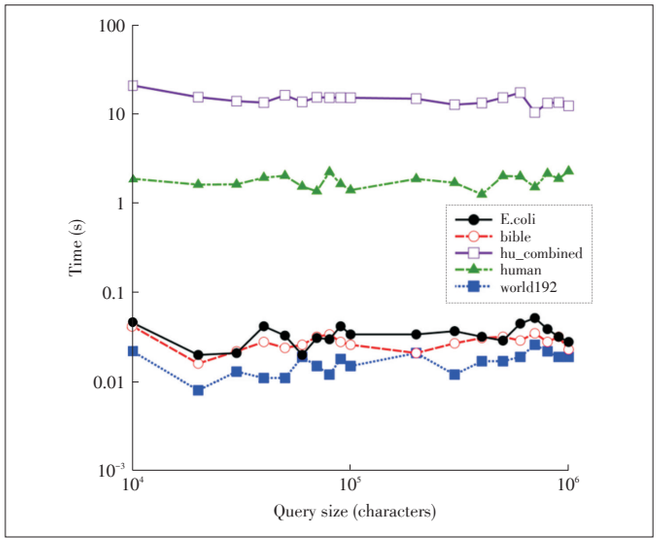
6.1 Experiment Setup

1) Client configuration: The client module was hosted and executed on a Toshiba Satellite Ultrabook U920t running Linux Ubuntu 14.04 LTS 64-bit, with 3.8 GiB RAM, 247.8 GB SSD and Intel® Core™i5-3337U CPU @ 1.80GHz x 4 processor.
2) Server configuration: The server module was run on a VMware 30 vCPU 64GB RAM CentOS 6 Linux virtual machine, which was hosted on a cluster of 19 physical servers.
3) Experiment parameters: RSA was used as the owner's signature mechanism, and the secret and public keys were generated with 2048-bits as the security parameter. The same construction was used to generate the server's secret and public keys. SHA256 was used to generate hashes for the MHT, with the digest truncated to 160 bits.
4) Prototype implementation: The prototype was implemented in C++ on both the client and server machines. Coding was initially performed on a Windows 8.1 machine with Visual Studio 2010, and was then ported to the client running on Ubuntu 14.04 with CodeBlocks 13.12 and GNU C++ 4.8.2. The suffix array construction algorithm was sourced from *libdivsufsort* that has been shown to be very efficient compared to other implementations [31]. Cryptographic functions for hashing and signatures were sourced from OpenSSL 1.0.1g. The owner-generated SA and MHT were made persistent and stored to disk (rather than temporarily creating and destroying them in memory) and 'uploaded' to the server along with the data file to facilitate query processing. From an experimental point-of-view, this facilitated ease-of-use with respect to avoiding running the same process again on the server. In practice, the server would probably re-generate both the SA and MHT independently, however this is not a requirement for the scheme to work securely. Either option (i.e. uploading the SA and MHT to the server, or independently re-generating them at the server) may be taken in practice. Consequently, the entire MHT is not loaded into memory (due to its size) by the server when processing queries. Rather, the appropriate nodes in the MHT are loaded as and when needed by the server during the VO generation phase. This serves two purposes: 1) to avoid using up large amounts of memory that could otherwise be used by other processes on the server, and 2) to reduce the overhead in loading the entire MHT into memory when queries are being processed. VOs are essentially realised as text files with an XML-like structure, without the end tags. This allows the client to recognise and parse the data in the VO in a straightforward manner.
5) Datasets: The datasets comprise of a total of five files. Three of the files were taken from the Large Canterbury Corpus [32], and two were sourced from the NCBI [33]:
   - E.coli: Complete genome of the E. Coli bacterium, size 4,638,690 b
   - bible: The King James version of the bible, size 4,047,392 b
   - world192: The CIA world fact book, size 2,473,400 b
   - human: Chromosome 10 from human genome data, size 128,985,118 b
   - hu_combined: A concatenation of chromosomes 1, 3, 5, 6, 9, and 11 from human genome data to form an approximately 1 GB file, size 1,000,003,018 b.
6) Query workload: Contiguous fragments of size 10,000 to 100,000 characters in 10,000 character increments, and from 100,000 to 1,000,000 characters in 100,000 character increments were taken from each dataset at randomly selected positions. This produced 19 query strings ranging in sizes from 10,000 characters to 1,000,000 characters for each dataset. The 19 query strings were then submitted to the server in ascending size order from the smallest query (10,000 characters) to the largest query (1,000,000 characters). The queries were processed by the server synchronously, with the query result, VO generation, and query verification for each query being performed prior to submission of the subsequent query. This series of 19 queries per dataset was repeated for a total of 30 runs per dataset to get 30 results for each individual query.

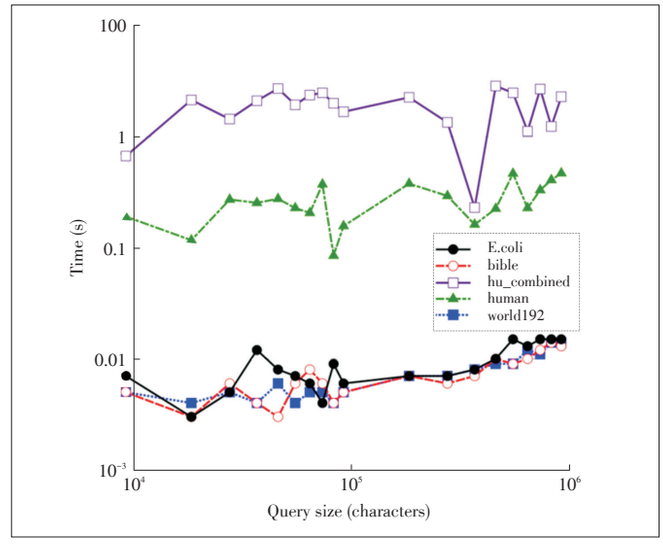
The query execution time, VO generation time, VO size and verification times were measured for each query and recorded. Averages of each of these recordings were taken for each query/dataset combinations to produce the final results as shown in the upcoming graphs.

6.2 Query Execution Times

Our research focuses on the verification of substring queries, and not the querying itself. However, we have included the query execution times as part of the results to provide a more holistic view of the implementation of the scheme. Due to the fact that the data file is not loaded into memory prior to query execution, the execution times are affected by the hits and misses due to caching. For this reason, we find that the resulting graph, in **Fig. 4**, produces slightly varying times. The size of the data file determines the difference in finding queries between files of different sizes, and so we note that the data file queries to bible, world192, and E.coli perform better on the whole than the queries to human and hu_combined. In particular, queries to the hu_combined data file takes more than 10 seconds to execute due to its comparatively larger size (1 GB) than the other data files. We also note that regardless of the query size, the query execution times remain relatively similar for queries executed on individual data files. This is ex-



▲Figure 4. Query execution times for the SA-MHT prototype.



▲Figure 5. VO generation time for the SA-MHT prototype.

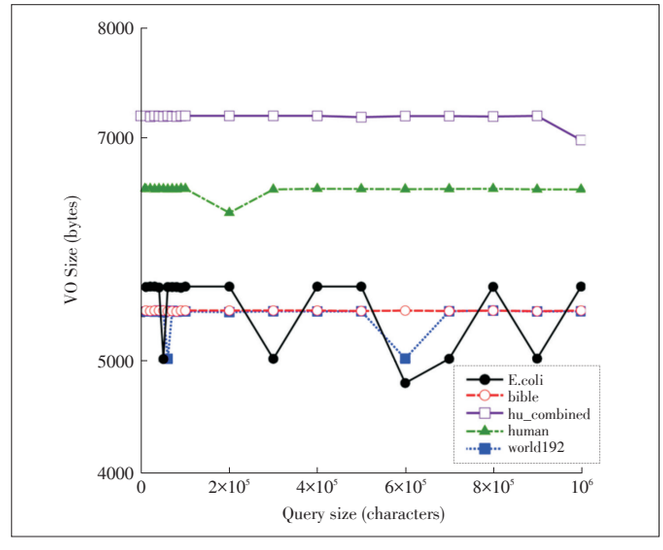
plained by the fact that a binary search is performed for each query, and occupies log $n + m$ time for each query, resulting in fairly similar times regardless of the query size.

6.3 VO Generation Time

The VO generation time is the time the SA-MHT prototype takes for the server to generate VOs for any given query. This time is incurred in addition to the query execution time and from the server's perspective. It is the cost of participating in the verification scheme per query. The results, shown in **Fig. 5**, indicate a range from 3 ms for the shorter queries to 15 ms for longer queries for the bible, world192, and E.coli data files. The human data file queries show a range VO generation times from 85 ms to 469 ms, whilst the hucombined data file shows a range from 230 ms to 2.864 s. The generation time of all VOs tends to move towards the respective upper bounds of their individual ranges as the query sizes increase. The outliers at 10, 000 characters for the human data file, and 10,300 characters for the hu_combined data file could be due to cache hits as well. The VO generation phase reads the verification path nodes of the MHT from disk, node-by-node, and as such is also affected by the cache.

6.4 VO Sizes

The VO sizes seem to be bounded to a fairly constant range for each of the data files, as shown in **Fig. 6**. The VO sizes for the bible, world192, and E.coli data files seem to share a similar range of values between 4.8 KB to 5.6 KB, and this is due to the possible number of nodes in the verification path for each suffix, which is bounded by $O(\log n)$. It is worth noting that the log n bound is reflected by the jump from the lower three data files, which have almost the same log n bounds, to the human data file, which has a higher log n bound (ranging between 6.3 KB and 6.5 KB), and then another jump to the

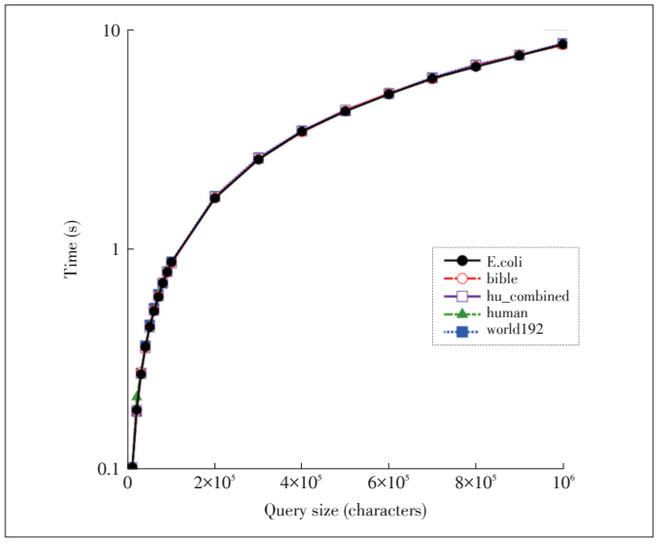


▲Figure 6. VO sizes for the SA-MHT prototype.

hu_combined data file VO sizes which has an even higher log n bound (ranging between 7 KB and 7.2 KB). The deviations from the otherwise constant values are due to fewer verification path nodes being generated for MHT nodes that happen to lie at the end of a level without a sibling (i.e. it is the end node of a level that has an odd number of nodes). In such a case, the node is not included in the verification path, and is simply promoted to the previous levels until a sibling is found. This results in fewer verification path nodes for that verification path compared to the verification path of nodes that have siblings.

6.5 Query Verification Time

Fig. 7 shows the query verification time incurred by the client. The initial observation is that regardless of data file size, the verification time for different query sizes are virtually the



▲Figure 7. Query verification time for the SA-MHT prototype.

same. This is a reflection of the relatively constant sizes of the VO and the fact that the difference amongst the sizes of the VOs for different data files is less than 2 KB (Fig. 6). The query size is the determining factor and this can be seen through the rise of the curve as the query sizes increase. This is a result of the number of hashes performed by the client, the maximum of which is the size of the query per suffix being verified.

6.6 Discussion on Experiment Results

The experiments on the prototype have shown promising results on the whole. The additional time spent by the server in generating the VO is largely a fraction of the query execution time, and in practice would be unnoticeable by the client. Additionally, the size of the VO is also fairly constant and does not appear to be affected much by the size of the query. The data file size causes the VO to increase, but only by a couple of kilobytes for a data file increase from 4 MB to 1 GB. Finally, the client-side verification incurs less than a second of processing time for query sizes of up to 100,000 characters, which is a large query for most applications. Larger query sizes incur more times, and are a function of the size of the query, but may still be considered usable in practice.

7 Conclusions and Future Work

We have presented an existential substring query verification scheme that meets the properties of authenticity, completeness and freshness. The scheme allows consumers to query for the existence of arbitrary substrings that are not restricted to keyword searches only, and provides verification objects with the results as proofs of correctness. Our scheme is based on suffix arrays, and provides improvements in the space and processing time in comparison to the only other comparable scheme proposed in [1]. Our scheme also provides consistently smaller VOs for large substring matches compared to the scheme proposed in [1]. The experiment results on a fully functioning prototype are promising for the applicability of our scheme to appropriate applications on the cloud.

References

[1] C. Martel, G. Nuckolls, P. Devanbu, et al., "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, Jan. 2004. doi: 10.1007/s00453-003-1076-8.
[2] Wikipedia. (2015 March). *Agglutination* [Online]. Available: http://en.wikipedia.org/w/index.php?title=Agglutination&oldid=648155093
[3] G. Ateniese, R. Burns, R. Curtmola, et al., "Provable data possession at untrusted stores," in *Proc. 14th ACM Conference on Computer and Communications Security*, Alexandria, USA, 2007, pp. 598–609. doi: 10.1145/1315245.1315318.
[4] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proc. 4th International Conference on Security and Privacy in Communication Networks*, Istanbul, Turkey, Article 9, 2008, pp. 9:1–9:10. doi: 10.1145/1460877.1460889.
[5] A. Juels and B. S. Kaliski, "PORS: proofs of retrievability for large files," in *Proc. 14th ACM Conference on Computer and Communications Security*, Alexandria, USA, 2007, pp. 584–597. doi: 10.1145/1315245.1315317.
[6] T. S. J. Schwarz and E. L. Miller, "Store, forget, and check: using algebraic signatures to check remotely administered storage," in *26th IEEE International Conference on Distributed Computing Systems*, Lisboa, Portugal, 2006, pp. 12–12. doi: 10.1109/ICDCS.2006.80.
[7] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proc. 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, USA, 2006, pp. 121–132. doi: 10.1145/1142473.1142488.
[8] E. Mykletun, M. Narasimha, and G. Tsudik, "Providing authentication and integrity in outsourced databases using merkle hash trees," UCI-SCONCE Technical Report, 2003.
[9] M. Narasimha and G. Tsudik, "DSAC: integrity for outsourced databases with signature aggregation and chaining," in *Proc. 14th ACM International Conference on Information and Knowledge Management*, Bremen, Germany, 2005, pp. 235–236. doi: 10.1145/1099554.1099604.
[10] H. Pang and K.-L. Tan, "Verifying completeness of relational query answers from online servers," *ACM Transactions on Information and System Security*, vol. 11, no. 2, Article 5, Mar. 2008. doi: 10.1145/1330332.1330337.
[11] Q. Zheng, S. Xu, and G. Ateniese, "Efficient query integrity for outsourced dynamic databases," in *Proc. 2012 ACM Workshop on Cloud Computing Security Workshop*, Raleigh, USA, 2012, pp. 71–82. doi: 10.1145/2381913.2381927.
[12] K. Mouratidis, D. Sacharidis, and H. Pang, "Partially materialized digest scheme: an efficient verification method for outsourced databases," *The VLDB Journal*, vol. 18, no. 1, pp. 363–381, Jan. 2009. doi: 10.1007/s00778-008-0108-z.
[13] S. Singh and S. Prabhakar, "Ensuring correctness over untrusted private database," in *Proc. 11th International Conference on Extending Database Technology: Advances in Database Technology*, Nantes, France, 2008, pp. 476–486. doi: 10.1145/1353343.1353402.
[14] M. T. Goodrich, R. Tamassia, and N. Triandopoulos, "Super-efficient verification of dynamic outsourced databases," in *Proc. The Cryptopgraphers' Track at the RSA conference on Topics in Cryptology (CT-RSA'08)*, San Francisco, USA, Apr. 2008, pp. 407–424. doi: 10.1007/978-3-540-79263-5_26.
[15] M. Noferesti, M. A. Hadavi, and R. Jalili, "A signature-based approach of correctness assurance in data outsourcing scenarios," *Information Systems Security*, vol. 7093, S. Jajodia and C. Mazumdar, Eds. Germany: Springer Berlin Heidelberg, 2011, pp. 374–378. doi: 10.1007/978-3-642-25560-1_26.
[16] T. K. Dang, "Ensuring correctness, completeness, and freshness for outsourced tree-indexed data," *Information Resources Management Journal*, vol. 21, no. 1, pp. 59–76, Jan. 2008. doi: 10.4018/irmj.2008010104.
[17] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *Proc. 11th International Conference on Database Systems for Advanced Applications (DASFAA'06)*, Singapore, Apr. 2006, pp. 420–436. doi: 10.1007/1173383630.
[18] M. Xie, H. Wang, J. Yin, and X. Meng, "Providing freshness guarantees for out-

sourced databases," in *Proc. 11th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '08)*, New York, USA, 2008, pp. 323–332. doi: 10.1145/1353343.1353384.
[19] R. Jain and S. Prabhakar, "Trustworthy data from untrusted databases," in *IEEE 29th International Conference on Data Engineering (ICDE)*, Brisbane, Australia, Apr. 2013, pp. 529–540. doi: 10.1109/ICDE.2013.6544853.
[20] Y. Zhou and C. Wang, "A query verification method for making outsourced databases trustworthy," in *IEEE Ninth International Conference on Services Computing (SCC)*, Honolulu, USA, Jun. 2012, pp. 298– 305. doi: 10.1109/SCC.2012.63.
[21] G. Nuckolls, "Verified query results from hybrid authentication trees," in *Data and Applications Security XIX*, vol. 3654, S. Jajodia and D. Wijesekera, Eds. Springer Berlin Heidelberg, 2005, pp. 84–98. doi: 10.1007/11535706_7.
[22] B. Palazzi, M. Pizzonia, and S. Pucacco, "Query Racing: Fast Completeness Certification of Query Results," in *Data and Applications Security and Privacy XXIV*, vol. 6166, S. Foresti and S. Jajodia, Eds. Germany: Springer Berlin Heidelberg, 2010, pp. 177–192. doi: 10.1007/978-3-642-13739-6_12.
[23] H. Pang and K. Mouratidis. "Authenticating the query results of text search engines," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 126 - 137, Aug. 2008. doi: 10.14778/1453856.1453875.
[24] M. T. Goodrich, C. Papamanthou, D. Nguyen, et al., "Efficient verification of web-content searching through authenticated web crawlers," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 920-931, Jun. 2012. doi: 10.14778/2336664.2336666
[25] R. Rivest, "The MD5 message-digest algorithm," IETF RFC1321, 1992.
[26] *Secure Hash Standard, Federal Information Processing Standard (FIPS)*, FIPS 180-2, Aug. 2002.
[27] *Secure Hash Standard, Federal Information Processing Standard (FIPS)*, FIPS 180-4, Mar. 2012.
[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. doi: 10.1145/359340.359342.
[29] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*, Oakland, USA, Apr. 1980, pp. 122–134. doi: 10.1109/SP.1980.10006.
[30] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *DARPA Information Survivability Conference & Exposition II*, Anaheim, USA, Jun. 2001. pp. 68–82. doi: 10.1109/DISCEX.2001.932160.
[31] Y. Mori. (2014). *The Benchmark Results of Implementations of Various, Latest Suffix Array Construction Algorithms* [online]. Available: https://code.google.com/p/libdivsufsort/wiki/SACABenchmarks
[32] T. Bell. (2014). *The Large Canterbury Corpus* [online]. Available: http://corpus.canterbury.ac.nz/descriptions/#large
[33] National Center for Biotechnology Information. (2014). *Homo-Sapien Genome* [online]. Available: ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/

Manuscript received: 2016-04-09

Biographies

Faizal Riaz-ud-Din (faizal.din@ieee.org) has a background in database and software development. He has been working in academia as well as in industry for a number of years and is currently pursuing a doctorate part-time. His interests lie in the area of query verification in databases and text-based data on the cloud.

Robin Doss (robin.doss@deakin.edu.au) received the BEng from the University of Madras, India, in 1999, and the MEng and PhD degrees from the Royal Melbourne Institute of Technology (RMIT), Australia, in 2000 and 2004, respectively. He has held professional appointments with Ericsson Australia, RMIT University, and IBM Research, Switzerland. He joined Deakin University, Australia, in 2003, and is currently a senior lecturer in computing. Since 2003, he has published more than 50 papers in refereed international journals, international conference proceedings and technical reports for industry and government. His current research interests are in the broad areas of communication systems, protocol design, wireless networks, security and privacy. He is a member of the IEEE.