

Melissa Chase and Emily Shen

# Substring-Searchable Symmetric Encryption

**Abstract:** In this paper, we consider a setting where a client wants to outsource storage of a large amount of private data and then perform substring search queries on the data – given a data string  $s$  and a search string  $p$ , find all occurrences of  $p$  as a substring of  $s$ . First, we formalize an encryption paradigm that we call *queryable encryption*, which generalizes searchable symmetric encryption (SSE) and structured encryption. Then, we construct a queryable encryption scheme for substring queries. Our construction uses suffix trees and achieves asymptotic efficiency comparable to that of unencrypted suffix trees. Encryption of a string of length  $n$  takes  $O(\lambda n)$  time and produces a ciphertext of size  $O(\lambda n)$ , and querying for a substring of length  $m$  that occurs  $k$  times takes  $O(\lambda m + k)$  time and three rounds of communication. Our security definition guarantees correctness of query results and privacy of data and queries against a malicious adversary. Following the line of work started by Curtmola et al. (ACM CCS 2006), in order to construct more efficient schemes we allow the query protocol to leak some limited information that is captured precisely in the definition. We prove security of our substring-searchable encryption scheme against malicious adversaries, where the query protocol leaks limited information about memory access patterns through the suffix tree of the encrypted string.

**Keywords:** searchable encryption, suffix trees, substring search, cloud storage

DOI 10.1515/popets-2015-0014

Received 2015-02-15; revised 2015-05-15; accepted 2015-05-15.

## 1 Introduction

In traditional symmetric-key encryption schemes, a user encrypts a message so that only the owner of the corresponding secret key can decrypt it. Decryption is “all-or-nothing”; that is, with the key one can decrypt the mes-

sage completely, and without the key one learns nothing about the message. However, many settings such as cloud storage call for encryption schemes that support the evaluation of certain classes of queries on the data without decrypting the data. A client may wish to store encrypted data on a cloud server and then be able to issue queries on the data to the server in order to make use of the data without retrieving and decrypting the original ciphertext.

Much work has been done on searchable symmetric encryption (SSE), which considers the setting where the data consists of a set of documents that the client wishes to search for combinations of keywords. However, we are interested in applications where one wants to search not for predetermined keywords but for arbitrary substrings. For example, suppose a medical research lab wants to store subjects’ genomic data using a cloud storage service. Privacy concerns may require that this data be encrypted. At the same time, the researchers need to be able to use the data efficiently. Researchers may be interested in making substring queries on the genomic data to determine whether a particular cancer marker sequence appears in any of the data or to count whether a certain probe sequence is rare enough to be useful. In addition to protecting the privacy of the data from the cloud provider, researchers would like to ensure that the process of performing queries does not reveal information to the cloud about the queries or the original data.

We note that existing SSE techniques do not solve the substring search problem efficiently; applying SSE by considering every substring of the original string as a separate keyword results in  $O(n^2)$  storage for a string of length  $n$ . Our goal is to avoid this storage overhead and achieve  $\tilde{O}(n)$  storage (as one would have in the unencrypted scenario).

### Queryable encryption

In this paper, we first define *queryable encryption*, which generalizes the SSE and structured encryption paradigms. A queryable encryption scheme allows for evaluation of some query functionality  $\mathcal{F}$  that takes as input a message  $M$  and a query  $q$  and outputs an answer. A client encrypts a message  $M$  under a secret key and stores the ciphertext on a server. Then, using the

---

**Melissa Chase:** Microsoft Research, E-mail: melis-sac@microsoft.com

**Emily Shen:** MIT Lincoln Laboratory, E-mail: emily.shen@ll.mit.edu. Work performed while at Microsoft Research.

secret key, the client can issue a query  $q$  by executing an interactive protocol with the server. At the end of this protocol, the client learns the value of  $\mathcal{F}(M, q)$ . For example, for substring search queries, a query  $q$  is a search string, the message  $M$  is a string, and  $\mathcal{F}(M, q)$  returns the set of indices of all occurrences of  $q$  as a substring of  $M$ .

### Substring-searchable encryption

We give a construction for a queryable encryption scheme for substring search queries – given a string  $s$  and a search string  $p$ , return all occurrences of  $p$  as a substring of  $s$ . Our construction has asymptotic efficiency comparable to that of substring search on unencrypted data.

We note that general techniques such as fully homomorphic encryption [9, 10, 25, 26] and functional encryption [7, 36, 42] would not achieve our efficiency goals. Instead, we tailor our scheme to the specific functionality of substring search.

To construct a substring-searchable encryption scheme, we use suffix trees, a data structure used to efficiently perform substring search on unencrypted data. We combine basic symmetric-key primitives to develop a method that allows traversal of select edges in a suffix tree in order to efficiently perform substring search on encrypted data, without revealing significant information about the string or the queries.

A suffix tree for a data string of length  $n$  takes  $O(n \log n)$  space, and searching for a substring of length  $m$  takes  $O(m + k)$  time, where  $k$  is the number of occurrences of the substring. In our substring-searchable encryption scheme, encryption time and ciphertext size are  $O(\lambda n)$ , and querying for a substring takes time and communication complexity  $O(\lambda m + k)$ , where  $\lambda$  is the security parameter. The query protocol takes a constant number of rounds of communication. All operations are based only on symmetric-key primitives.

### Security

We prove security of our scheme against malicious adversaries. For security, we will think of the server as an adversary trying to learn information about the message and the queries. Ideally, we want an adversary that is given a ciphertext and that engages in query protocols for several queries to learn nothing about the message or the queries. However, in order to construct a more efficient scheme, we will allow some limited information about the message and the queries to be revealed

(“leaked”) to the server through the ciphertext and the query protocol. Our security definition specifies explicitly what information is leaked and guarantees that an adversary learns nothing more than the specified leakage. This approach of trading off perfect privacy for efficiency has been adopted previously in the case of structured encryption [13] and in recent work on searchable encryption variants [11].

Our definition is similar to previous definitions for structured encryption and SSE. However, while previous definitions focused on document retrieval and the adversary had to learn the list of documents returned, in our definition there are no documents and all that is required is that the client learn the result of the query. Furthermore, most previous definitions have focused on honest-but-curious adversaries. We define security within a malicious adversary model.

## 2 Related Work

### Searchable encryption and structured encryption

We draw on related work on symmetric searchable encryption (SSE) [16] and its generalization to structured encryption [13]. These works take the approach of considering a specific type of query and identifying a data structure that allows efficient evaluation of those queries in an unencrypted setting. The construction then “translates” the data structure into an encrypted setting, so that the user can encrypt the data structure and send the server a *token* to evaluate a query on the encrypted structure. This translation is designed to preserve the efficiency of the unencrypted data structure.

Since the server is processing the query, the server can determine the memory access pattern of the queries, that is, which parts of memory have been accessed, and when the same memory block is accessed again.<sup>1</sup> The approach to security in SSE and structured encryption is to acknowledge that some information will be leaked because of the memory access pattern, but to clearly specify the leakage, and to guarantee that is the only information that the server can learn.

There have been many recent advances in SSE. Cash et al. [11] propose an efficient construction for searches involving multiple keywords. Several works [12, 30, 31,

<sup>1</sup> Note that this is true even if we use fully homomorphic encryption (e.g., [9, 10, 25, 26]) or functional encryption [7, 36, 42].

45] propose schemes that allow updates to the stored documents, and Kurosawa and Ohtaki [38] propose a UC definition. However, all of these works focus on the problem of retrieving documents based on keywords; there has been very little work that considers encrypting more complex types of data structures.

### Predicate encryption and fully homomorphic encryption

Predicate encryption (a special case of functional encryption [7]) allows the secret key owner to generate tokens for various predicates. One can evaluate a token for a predicate  $f$  on an encryption of  $m$  to determine whether  $f(m)$  is satisfied. State-of-the-art predicate encryption schemes (e.g., [36, 42]) support inner-product queries; that is,  $f$  specifies a vector  $v$ , and  $f(m) = 1$  if  $\langle m, v \rangle = 0$ . Applying an inner product predicate encryption scheme naively to construct a substring-searchable encryption scheme, where the substrings can be of any length, would result in ciphertexts and query time that are  $O(n^n)$ , where  $n$  is the length of the string  $s$ , which is clearly impractical.

Fully homomorphic encryption (FHE), beginning with the breakthrough work of Gentry [25] and further developed in subsequent work, e.g., [9, 10, 27], allows one to evaluate any arbitrary circuit on encrypted data without being able to decrypt. FHE would solve the substring-searchable encryption problem (although it would require  $O(n)$  query time), but existing constructions are extremely impractical.

### Oblivious RAMs

The problem of leaking the memory access pattern is addressed in the work on oblivious RAMs [41], which shows how to implement any query in a way that ensures that the memory access pattern is independent of the query. There has been significant progress in making oblivious RAMs more efficient; however, even the most efficient constructions to date (see, e.g., Stefanov et al. [43]) increase the amortized costs of processing a query by a factor of at least  $\log n$ , where  $n$  is the size of the stored data. In our setting, where we assume that the large size of the dataset may be one of the primary motivations for outsourcing storage, a  $\log n$  overhead may be unacceptable.

### Secure two-party computation of substring search

There have been several works on secure two-party or multiparty computation (e.g., [17, 40]) and specifically on secure substring search and other text processing in the two-party setting (see [3, 22, 24, 29, 34, 39, 46]). This is an interesting line of work; however, our setting is rather different. In our setting, the client has outsourced storage of its encrypted data to a server, and then the client would like to query its data with a search string. The server does not have the data string in the clear; it is encrypted. Thus, even ignoring the extra rounds of communication, we cannot directly apply secure two-party substring search protocols.

### Memory delegation and integrity checking

We consider security against malicious adversaries. One way a malicious adversary may misbehave is by returning something other than what was originally stored on the server. Along these lines, there is related work on memory delegation (e.g., [14]) and memory checking (e.g., [18]), verifiable computation (e.g., [6, 23]), integrity checking (e.g., [44]), and encrypted computation on untrusted programs (e.g., [21]); the theme of these works is retrieving and computing on data stored on an untrusted server. For our purposes, since we focus on the specific functionality of substring-searchable encryption in order to achieve an efficient scheme using simple primitives, we do not need general purpose integrity checking techniques, which can be expensive or rely on more complex assumptions.

## 3 Preliminaries

In this section, we review notation and definitions of the data structures and cryptographic primitives we will use. For formal definitions of the cryptographic primitives, we refer the reader to [33].

### 3.1 Notation

We write  $x \stackrel{R}{\leftarrow} X$  to denote an element  $x$  being sampled uniformly at random from a finite set  $X$ , and  $x \leftarrow A$  to denote the output  $x$  of an algorithm  $A$ .

If  $x$  is a string, then  $|x|$  refers to the length of  $x$ , and  $x_i$  denotes the  $i$ th character of  $x$ . If  $|x| = n$  and  $a$  and  $b$  are integers  $1 \leq a \leq b \leq n$ , then  $x[a..b]$  denotes the substring  $x_a \dots x_b$ . If  $x$  and  $y$  are strings, then  $x||y$

denotes the concatenation of  $x$  and  $y$ . We use  $\epsilon$  to denote the empty string.

If  $S$  is a set, then  $|S|$  refers to the cardinality of  $S$ , and  $\mathcal{P}(S)$  denotes the power set of  $S$  (the set of all subsets of  $S$ ). If  $n$  is a positive integer,  $[n]$  denotes the set  $\{1, \dots, n\}$ .

If  $F : \mathcal{K} \times D \rightarrow R$  is a family of functions, we write  $F_K$  for the function defined by  $F_K(x) = F(K, x)$ . We sometimes write  $\text{Enc}_K(m)$  and  $\text{Dec}_K(c)$  for  $\text{Enc}(K, m)$  and  $\text{Dec}(K, c)$ , respectively.

### 3.2 Data Structures

A dictionary  $D$  is a data structure that contains key/value pairs. For our construction it is sufficient for a dictionary to support insert and lookup operations. The insert operation takes a key/value pair  $(k, v)$  and adds it to the dictionary. The lookup operation takes a key  $k$  and returns the associated value  $v = D[k]$ .

### 3.3 Symmetric-Key Encryption

A symmetric encryption scheme  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  consists of three polynomial-time algorithms.  $\text{Gen}$  is a probabilistic algorithm that takes a security parameter  $\lambda$  and outputs a secret key  $K$ .  $\text{Enc}$  is a probabilistic algorithm that takes a key  $K$  and a message  $M$  and outputs a ciphertext  $CT$ .  $\text{Dec}$  is a deterministic algorithm that takes a key  $K$  and a ciphertext  $CT$  and outputs a message  $M$  or the symbol  $\perp$ . Correctness requires  $\text{Dec}(K, \text{Enc}(K, M)) = M$  with probability 1 for all  $K$  and  $M$ .

We use the following security notions for symmetric encryption.

- *CPA security* requires that ciphertexts reveal no information about plaintexts (other than length) to a PPT adversary that can adaptively query an encryption oracle.
- *Ciphertext integrity* [4, 5, 35] requires that it be infeasible for any PPT adversary given access to an encryption oracle to construct a new ciphertext that decrypts successfully. A symmetric encryption scheme is *authenticated* if it has both CPA security and ciphertext integrity.
- *Key hiding* (also known as *which-key concealing*) [1, 20] requires that it be infeasible for any PPT adversary given access to two encryption oracles to tell whether they encrypt using the same key or different keys.

### 3.4 Pseudorandom Functions and Permutations

A pseudorandom function family (PRF) (respectively, pseudorandom permutation family (PRP)) is a family  $F$  of functions such that it is computationally infeasible for any PPT adversary to distinguish a function chosen randomly from  $F$  from a uniformly random function (resp., permutation).

An almost-universal hash function is a family  $\mathcal{H}$  of hash functions such that for any pair of distinct messages the probability of a hash collision for a hash function chosen randomly from  $\mathcal{H}$  is negligible.

A PRF composed with an almost-universal hash function results in another PRF. That is, one can evaluate a PRF on a long input by first hashing it using an almost-universal hash function to a short input and then applying a PRF.

## 4 Queryable Encryption

We now formalize queryable encryption and present our main definitions.

### 4.1 Functionality

**Definition 4.1.** A *queryable encryption scheme* supporting query functionality  $\mathcal{F} : \mathcal{M} \times \mathcal{Q} \rightarrow \mathcal{R}$  for message space  $\mathcal{M}$ , query space  $\mathcal{Q}$ , and result space  $\mathcal{R}$  consists of three probabilistic polynomial-time algorithms.

$\text{Gen}(1^\lambda) \rightarrow K$ : The key generation algorithm takes a security parameter  $\lambda$  and generates a secret key  $K$ .

$\text{Enc}(K, M) \rightarrow CT$ : The encryption algorithm takes a secret key  $K$  and a message  $M \in \mathcal{M}$ , and outputs a ciphertext  $CT$ .

$\text{Query}(K, q, CT)$ : The interactive query protocol occurs between a client and a server. The client's input is the secret key  $K$  and a query  $q \in \mathcal{Q}$ , and the server's input is a ciphertext  $CT$ . The client's output is a query result  $r \in \mathcal{R}$ ; the server has no output.

For correctness we require the following property. For all  $\lambda \in \mathbb{N}$ ,  $q \in \mathcal{Q}$ ,  $M \in \mathcal{M}$ , let  $K \leftarrow \text{Gen}(1^\lambda)$ ,  $CT \leftarrow \text{Enc}(K, M)$ , and  $r \leftarrow \text{Query}(K, q, CT)$ . Then  $\Pr[r = \mathcal{F}(M, q)] = 1 - \text{negl}(\lambda)$ .

### Substring-searchable symmetric encryption

We define substring-searchable encryption as a special case of queryable encryption.

**Definition 4.2.** A *substring-searchable symmetric encryption scheme* for an alphabet  $\Sigma$  is a queryable encryption scheme for message space  $\mathcal{M} = \Sigma^*$ , query space  $\mathcal{Q} = \Sigma^*$ , result space  $\mathcal{R} = \mathcal{P}(\mathbb{N})$ , and query functionality  $\mathcal{F}$ , where  $\mathcal{F}(s, p)$  is the set of indices of all the occurrences of  $p$  as a substring of  $s$ . That is,  $\mathcal{F}(s, p) = \{i \mid s[i..i + m - 1] = p\}$ , where  $m = |p|$ .

#### Discussion

Note that the definition of a queryable encryption scheme does not include an explicit decryption algorithm. If full decryption is desired, one can include a query in  $\mathcal{F}$  that returns the entire message.

Note also that typically we expect  $M$  to be quite large, while the representation of  $q$  and  $\mathcal{F}(M, q)$  are small, so we would like the query protocol to be efficient relative to the size of  $q$  and  $\mathcal{F}(M, q)$ . Without such efficiency goals, designing a queryable encryption scheme would be trivial: the server could return the entire ciphertext, and the client could decrypt the ciphertext to get  $M$  and compute  $\mathcal{F}(M, q)$  directly.

#### Related definitions

Our queryable encryption definition can be viewed as a generalization of previous definitions of searchable encryption [16] and structured encryption [13]. Queryable encryption allows any general functionality  $\mathcal{F}$ . In contrast, the definition of searchable encryption is tied to the specific functionality of returning documents containing a requested keyword. Structured encryption is a generalization of searchable encryption, but the functionalities are restricted to return pointers to elements of an encrypted data structure.

Since we allow general functionalities, our definition is similar to those of functional encryption. The main technical difference is that our security definition only allows for a single ciphertext. Intuitively, in queryable encryption, encryption is a one-time process: a single (potentially very large) ciphertext is encrypted, and then many queries are performed on that ciphertext. We stress that one “message” encrypted under our scheme refers not to a single word or document but to a body of data upon which one wishes to be able to perform queries; this could be a collection of documents in

the SSE case, or a (set of) very long string(s) of data (e.g. a genome database) as in this work.<sup>2</sup>

Also, in queryable encryption we allow the query protocol to be interactive. In structured encryption, functional encryption, and many searchable encryption schemes the query protocol consists of two algorithms  $TK \leftarrow \text{Token}(K, q)$  and  $A \leftarrow \text{Query}(TK, CT)$ . The client constructs a query token and sends it to the server, and the server uses the token and the ciphertext to compute the answer to the query, which it sends back to the client. We can think of these schemes as having a one-round interactive query protocol. Our more general definition allows for arbitrary interactive protocols, which may enable better efficiency or privacy.

Finally, in contrast to related searchable encryption notions, we do not require the server to actually learn the answer to the query. After the server’s final message, the client may do some additional computation using its secret key to compute the answer. This can allow stronger privacy guarantees against the server.

## 4.2 Malicious $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 Security

We now present our simulation-based security definition against malicious adversaries. Following [13], we call the definition  $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 security, where the name “CQA2” comes from “chosen query attack” because the adversary chooses its queries adaptively. The security definition will be parameterized by two *leakage functions*  $\mathcal{L}_1$  and  $\mathcal{L}_2$ . First,  $\mathcal{L}_1(M)$  denotes the information about the message that is leaked by the ciphertext. Second, for any  $j$ ,  $\mathcal{L}_2(M, q_1, \dots, q_j)$  denotes the information about the message and all queries made so far that is leaked by the  $j$ th query.

We want to ensure that the information specified by  $\mathcal{L}_1$  and  $\mathcal{L}_2$  is the only information that is leaked to the adversary, even if the adversary can choose the message that is encrypted and then adaptively choose the queries for which it executes a query protocol with the client. To capture this, our security definition requires that the view of any adaptive adversary be simulatable given only the information specified by  $\mathcal{L}_1$  and  $\mathcal{L}_2$ .

Our definition differs from many previous definitions in that we allow the adversary to be arbitrarily malicious in the protocol. Since our protocol is interactive, this guarantee is important for privacy as well

<sup>2</sup> See the discussion at the end of Section 5.4 for an extension to allow for searches over a set of strings.

as correctness. We require that the adversary cannot distinguish the honest player’s output from the correct output (or  $\perp$  if the adversary misbehaved in the protocol). This means the honest protocol must always produce the correct output (or  $\perp$ ) even in the face of a malicious adversary; thus this definition captures both privacy and correctness.

**Definition 4.3** (Malicious  $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 security).

Let  $\mathcal{E} = (\text{Gen}, \text{Enc}, \text{Query})$  be a queryable encryption scheme for message space  $\mathcal{M}$ , query space  $\mathcal{Q}$ , result space  $\mathcal{R}$ , and query functionality  $\mathcal{F} : \mathcal{M} \times \mathcal{Q} \rightarrow \mathcal{R}$ . For functions  $\mathcal{L}_1$  and  $\mathcal{L}_2$ , adversary  $\mathcal{A}$ , and simulator  $\mathcal{S}$ , consider the following experiments:

**Real $_{\mathcal{E}, \mathcal{A}}(\lambda)$ :** The challenger begins by running  $\text{Gen}(1^\lambda)$  to generate a secret key  $K$ . The adversary  $\mathcal{A}$  outputs a message  $M$ . The challenger runs  $\text{Enc}(K, M)$  to generate a ciphertext  $CT$ , and sends  $CT$  to  $\mathcal{A}$ . The adversary adaptively makes a polynomial number of queries  $q_1, \dots, q_t$ . For each query  $q_i$ , first  $\mathcal{A}$  interacts with the challenger. The challenger plays the part of the client in the **Query** protocol with input  $(K, q_i)$  and sends its output to the adversary. Finally,  $\mathcal{A}$  outputs a bit  $b$ .

**Ideal $_{\mathcal{E}, \mathcal{A}, \mathcal{S}}(\lambda)$ :** First,  $\mathcal{A}$  outputs a message  $M$ . The simulator  $\mathcal{S}$  is given  $\mathcal{L}_1(M)$ , and outputs a value  $CT$ . The adversary adaptively makes a polynomial number of queries  $q_1, \dots, q_t$ . For each query  $q_i$ , the simulator is given  $\mathcal{L}_2(M, q_1, \dots, q_i)$  and interacts with  $\mathcal{A}$ . Then the simulator produces a flag  $f_i$ ; if  $f_i = \perp$ , the challenger sends  $\perp$  to the adversary, otherwise it sends  $\mathcal{F}(M, q_i)$ . Finally,  $\mathcal{A}$  outputs a bit  $b$ .

We say that  $\mathcal{E}$  is  $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 secure against malicious adversaries if, for all PPT adversaries  $\mathcal{A}$ , there exists a simulator  $\mathcal{S}$  such that

$$|\Pr[\text{Real}_{\mathcal{E}, \mathcal{A}}(\lambda) = 1] - \Pr[\text{Ideal}_{\mathcal{E}, \mathcal{A}, \mathcal{S}}(\lambda) = 1]| \leq \text{negl}(\lambda)$$

## 5 Substring-Searchable Encryption Construction

In this section, we construct a substring-searchable encryption scheme – a queryable encryption scheme that supports the functionality  $\mathcal{F}$ , where  $\mathcal{F}(s, p)$  returns the indices of all occurrences of  $p$  as a substring of  $s$ .

### 5.1 Suffix Trees

Our scheme draws upon substring search algorithms for unencrypted data. Several substring search algorithms exist [2, 8, 32, 37], varying in their preprocessing efficiency and query efficiency. Many algorithms have preprocessing time  $O(m)$  and query time  $O(n)$ , where  $n$  is the length of the string  $s$  and  $m$  is the length of the query substring  $p$ . In contrast, suffix trees [19, 28, 47] have preprocessing time  $O(n)$  and query time  $O(m)$ . This is ideal for our applications, where the client stores one string or set of strings on the server, and later performs queries for many search strings. Therefore, we will focus on substring search using suffix trees as the basis for our scheme.

Here we give a brief overview of suffix trees. We follow the terminology of [28].

**Definition 5.1.** A *suffix tree* for a string  $s = s_1 \dots s_n$  is a rooted, directed tree with the following properties:

- Each edge is labeled with a non-empty substring of  $s$ , called its *edge label*.
- Every internal node has at least two children.
- No two edges out of a node have edge labels starting with the same character.
- The tree has  $n$  leaves, labeled 1 to  $n$ . These are in one-to-one correspondence with the  $n$  suffixes to  $s$ . Specifically, for each  $i$ , the suffix  $s[i..n]$  is the concatenation of the edge labels on the path from the root to the leaf labeled  $i$ .

**Definition 5.2.** The *path label* of a node is the concatenation of the edge labels on the path from the root to that node.

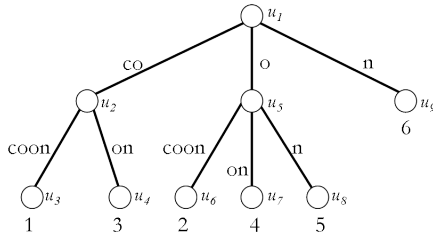
Figure 1 shows a suffix tree for the string “cocoon”.

Note that for a suffix tree to exist for a string  $s$ , it must be the case that no suffix of  $s$  is a prefix of another suffix of  $s$ . If this is not the case, one can append a special termination character  $\$$  that does not appear elsewhere in the string.

#### Substring search procedure

Searching for a substring using a suffix tree relies on the following key observation: a string  $p$  is a substring of  $s$  if and only if it is a prefix of some suffix of  $s$ . Thus, to search for  $p$  in  $s$ , we look for a path from the root whose label matches  $p$ .

To do this, match the characters of  $p$  sequentially with a path from the root. Since no two edges out of a



**Fig. 1.** A suffix tree for the string  $s = \text{“cocoon”}$ . The six suffixes of “cocoon” correspond to the paths from the root to the six leaves. Each leaf is labeled with the position in  $s$  where the corresponding suffix begins. Additionally, the nodes have arbitrary labels that are provided for future reference.

node start with the same character, this path is unique. Specifically, at each node on the path, find the outgoing edge whose label starts with the next character of  $p$  (if one exists) and continue matching along that edge. Continue until either the next character of  $p$  does not match, meaning  $p$  is not a substring of  $s$ , or all of  $p$  has been matched, meaning  $p$  is a substring of  $s$ . If  $p$  is a substring of  $s$ , the indices of the occurrences of  $p$  as a substring of  $s$  are exactly the indices labeling the leaves in the subtree below the end of the matching path.

**Efficiency**

A suffix tree can be constructed in  $O(n)$  time for a string of length  $n$  [19, 47]. It can be shown that a suffix tree has at most  $2n$  nodes. However, storing the edge label for all edges would require  $O(n^2)$  storage in the worst case. To represent a suffix tree in  $O(n \log n)$  space, one stores for each edge the start and end indices in  $s$  of the first occurrence of the edge label as a string of  $s$ , along with a copy of the string  $s$ .

Searching for a substring  $p$  of length  $m$  takes  $O(m)$  time to find a single occurrence, or  $O(m+k)$  time to find all occurrences, where  $k$  is the number of occurrences.

**Observations**

We make a few observations that will be useful for our construction. We can identify with each node  $u$  an *initial path label*, which is the concatenation of the path label of the parent of  $u$  and the first character of the edge label from the parent to  $u$ . Note that if a string  $p$  matches the initial path label of a node  $u$  and if  $p$  is a substring of  $s$ , then either  $p$ ’s matching path ends somewhere along the edge to  $u$ , or it ends somewhere in the subtree rooted at  $u$ . Note also that the indices

in  $s$  of occurrences of a node’s path label are exactly the indices of the occurrences of the node’s initial path label.

**5.2 Notation**

Before we describe our substring-searchable encryption scheme, we introduce some notation. Some of the notation will be relative to a string  $s$  and its suffix tree  $Tree_s$ , even though they are not explicit parameters.

$u$ :	a node in $Tree_s$
$\epsilon$ :	the empty string
$path(u)$ :	the path label of $u$ , i.e., the concatenation of the edge labels on the path from the root to $u$ . If $u$ is the root, $path(u) = \epsilon$ .
$initpath(u)$ :	the initial path label of $u$ , i.e., the concatenation of the path label of $u$ ’s parent and the first character of the edge label from $u$ ’s parent to $u$ . If $u$ is the root, $initpath(u) = \epsilon$ .
$leaf_i$ :	the $i$ th leaf in $Tree_s$ , where the leaves are numbered left to right
$len(u)$ :	the length of $initpath(u)$
$ind(u)$ :	the index in $s$ of the first occurrence of $path(u)$ (equivalently, of $initpath(u)$ ). If $path(u) = \epsilon$ , $ind(u) = 0$ .
$leafpos(u)$ :	the position (between 1 and $n$ ) in the tree of the leftmost leaf in the subtree rooted at $u$ .
$num(u)$ :	the number of occurrences in $s$ of $path(u)$ (equivalently, of $initpath(u)$ ) as a substring. If $path(u) = \epsilon$ , $num(u) = 0$ . For non-root nodes $u$ , $num(u)$ is equal to the number of leaves in the subtree rooted at $u$ .

To illustrate the notation above, let us look at the suffix tree in Figure 1 for the string “cocoon”. In this tree, we have  $path(u_3) = \text{“cocoon”}$ ,  $initpath(u_3) = \text{“coc”}$ ,  $leaf_3 = u_6$ ,  $ind(u_2) = 1$ ,  $leafpos(u_5) = 3$ ,  $num(u_2) = 2$ .

**5.3 Intuition**

Here we provide some intuition and work our way up to the full construction.

We will use a symmetric encryption scheme  $\Pi$ , a PRF  $F$ , and a PRP  $P$ . The key generation algorithm will generate keys  $K_D, K_C, K_L$  for  $\Pi$ , keys  $K_1, K_2$  for the PRF, and keys  $K_3, K_4$  for the PRP. We will explain how the keys are used as we develop the intuition for the construction.

### A first attempt

We first aim to construct a queryable encryption scheme for a simpler functionality  $\mathcal{F}'$ , where  $\mathcal{F}'(s, p)$  returns whether  $p$  occurs as a substring in  $s$ , and, if so, the index of the first occurrence in  $s$  of  $p$ . We will also only consider correctness and security against an honest-but-curious server, that is, a server that follows the protocol honestly.

Let  $\Pi$  be a CPA-secure symmetric encryption scheme. We will encrypt a string  $s = s_1 \dots s_n$  in the following way. First, construct a suffix tree  $Tree_s$  for  $s$ . Then construct a dictionary  $D$  that contains, for each node  $u$ , the key/value pair  $(F_{K_1}(\text{path}(u)), \Pi.\text{Enc}_{K_D}(\text{ind}(u)))$ . This dictionary is the ciphertext.

In the query protocol for a string  $p$ , the client sends  $F_{K_1}(p)$ . The server then looks up  $F_{K_1}(p)$  in the dictionary. If there is a corresponding value, the server returns it to the client. The client then decrypts using  $K_D$  to get the index of the first occurrence in  $s$  of  $p$ .

The problem with this approach is that it only works for search strings that fully match a node's path label (i.e., end exactly at a node); it does not work for finding substrings that end partway down an edge.

### Returning a possible match

To address this problem, we will identify each node with its initial path label instead of its path label. Note that if  $u$  is the last node (farthest from the root) for which any prefix of  $p$  equals  $\text{initpath}(u)$ , then either  $p$  is not a substring of  $s$ , or  $p$  ends somewhere on the path to  $u$ , and the indices in  $s$  of the occurrences of  $\text{initpath}(u)$  are the same as the indices of the occurrences of  $p$ .

In the dictionary  $D$ , we will now use  $\text{initpath}(u)$  instead of  $\text{path}(u)$  as the search key for a node  $u$ . We will say that a prefix  $p[1..i]$  is a *matching prefix* if  $p[1..i] = \text{initpath}(u)$  for some  $u$ ; otherwise, we say  $p[1..i]$  is a *non-matching prefix*. The ciphertext will also include an array  $C$  of encryptions of each character of  $s$ , with  $C[i] = \Pi.\text{Enc}_{K_C}(s_i)$ .

In the query protocol, the client will send  $T_1, \dots, T_m$ , where  $T_i = F_{K_1}(p[1..i])$ . The server finds the entry  $D[T_j]$ , where  $p[1..j]$  is the longest matching prefix of  $p$ . The server will return the encrypted index  $\Pi.\text{Enc}_{K_D}(\text{ind})$  stored in  $D[T_j]$ . The client will then decrypt it to get  $\text{ind}$ , the index of the first occurrence of the possible match, and requests the server to send  $C[\text{ind}], \dots, C[\text{ind} + m - 1]$ . The client then decrypts the result to check whether the decrypted string is equal to

the search string  $p$  and thus, whether  $p$  is a substring of  $s$ .

### Returning all occurrences

We would like to return not just the first occurrence or a constant number of occurrences, but all of the occurrences of the search string. However, in order to keep the ciphertext size  $O(n)$ , we need the storage for each node to remain a constant size. In a naive approach, in each dictionary entry we would store encryptions of indices of all of the occurrences of the corresponding string. However, this would take  $O(n^2)$  storage in the worst case.

To maintain constant storage for each node, we use the fact that the occurrences of a node's path label (or initial path label) as a substring of  $s$  are exactly the occurrences of the path labels of the leaves in the subtree rooted at that node, each of which is a suffix of  $s$ .

We construct a leaf array  $L$  of size  $n$ , with the leaves numbered 1 to  $n$  from left to right. Each element  $L[i]$  stores an encryption of the index in  $s$  of the path label of the  $i$ th leaf. That is,  $L[i] = \Pi.\text{Enc}_{K_L}(\text{ind}(\text{leaf}_i))$ . In the encrypted tuple in the dictionary entry for a node  $u$  we also store  $\text{leafpos}(u)$ , the position in the tree of the leftmost leaf in the subtree rooted at  $u$ , and  $\text{num}(u)$ , the number of leaves in the subtree rooted at  $u$ . That is, the value in the dictionary entry for a node  $u$  is now  $\Pi.\text{Enc}_{K_D}(\text{ind}(u), \text{leafpos}(u), \text{num}(u))$  instead of  $\Pi.\text{Enc}_{K_D}(\text{ind}(u))$ .

In the query protocol, the server will return the encryption of  $\text{ind}(u), \text{leafpos}(u), \text{num}(u)$  for the last node  $u$  matched by a prefix of  $p$ . The client then decrypts this and asks for  $C[\text{ind}], \dots, C[\text{ind} + m - 1]$ , decrypts to determine whether  $p$  is a substring of  $s$ , and if so, asks for  $L[\text{leafpos}(u)], \dots, L[\text{leafpos}(u) + \text{num} - 1]$  to retrieve all occurrences of  $p$  in  $s$ .

### Hiding common non-matching prefixes among queries

The scheme outlined so far works; it supports the desired substring search functionality, against an honest-but-curious adversary. However, it leaks a lot of unnecessary information to the server; we now add a number of improvements to reduce the information that is leaked.

In the scheme sketched so far, the server will learn from the  $T_i$  values when any two queries share a prefix, even if the shared prefix is not a substring of  $s$ . Although memory accesses will necessarily reveal when two queries share a *matching* prefix (contained in the



dictionary), but we would like to hide when queries share non-matching prefixes.

To hide when queries share non-matching prefixes, we change each  $T_i$  to be an encryption of  $f_1^{(i)} = F_{K_1}(p[1..i])$  under the key  $f_2^{(i)} = F_{K_2}(p[1..i])$ . The dictionary entry for a node  $u$  will now also contain values  $f_{2,i}$  for its children nodes, where  $f_{2,i} = F_{K_2}(\text{initpath}(v_i))$  for each of the children  $v_i$  of  $u$ .

In the query protocol, the server starts at the root node, and after reaching any node, the server tries using each of the  $f_{2,i}$  for that node to decrypt each of the next  $T_j$ 's, until it either succeeds and reaches the next node or it reaches the end of the search string.

### Hiding node degrees, order of children, and number of nodes in suffix tree

Since the maximum degree of any node is the size  $d$  of the alphabet, we can hide the degree of each node by creating dummy random  $f_{2,i}$  values so that there are  $d$  in total. To hide the order of the children and hide which of the  $f_{2,i}$  are dummy values, we store the  $f_{2,i}$  in a random permuted order in the dictionary entry.

Similarly, since a suffix tree for a string of length  $n$  contains at most  $2n$  nodes, we will hide the exact number  $N$  of nodes in the suffix tree by constructing  $2n - N$  dummy entries in  $D$ .

### Hiding string indices and leaf positions

In order to hide the actual values of the string indices  $\text{ind}, \dots, \text{ind} + m - 1$  and the leaf positions  $\text{leafpos}, \dots, \text{leafpos} + \text{num} - 1$ , we make use of a pseudorandom permutation family  $P$  of permutations  $[n] \rightarrow [n]$ . Instead of sending  $(\text{ind}, \dots, \text{ind} + m - 1)$ , the client applies the permutation  $P_{K_3}$  to  $\text{ind}, \dots, \text{ind} + m - 1$  and outputs the resulting values in a randomly permuted order. Similarly, instead of sending  $(\text{leafpos}, \dots, \text{leafpos} + \text{num} - 1)$ , the client applies the permutation  $P_{K_4}$  to  $\text{leafpos}, \dots, \text{leafpos} + \text{num} - 1$  and outputs the resulting values in a randomly permuted order. Note that while the server does not learn the actual indices or leaf positions, it still learns when two queries ask for the same or overlapping indices or leaf positions.

### Handling malicious adversaries

The scheme described so far satisfies security against an honest-but-curious adversary, but not against a malicious adversary; an adversary could potentially send

malformed or incorrect ciphertexts during the query protocol.

To handle a malicious adversary, we will require  $\Pi$  to be an authenticated encryption scheme. Thus, an adversary will not be able to construct a ciphertext that is not part of the dictionary  $D$  or the arrays  $C$  or  $L$ . We add auxiliary information to the encrypted messages to allow the client to check that any ciphertext returned by the server is the one expected by the honest algorithm. For example, for the characters of  $s$  we will encrypt  $(s_i, i)$  instead of just  $s_i$  so that the client can check that it is receiving the correct piece of the ciphertext. For dictionary entries, we will add auxiliary information in the encrypted tuple so that the client can check that the ciphertext returned corresponds to the longest matching prefix of  $p$ .

## 5.4 Construction

Let  $F : \{0, 1\}^\lambda \times \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$  be a PRF, and let  $P : \{0, 1\}^\lambda \times [n] \rightarrow [n]$  be a PRP. Let  $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$  be an authenticated, key-hiding symmetric-key encryption scheme. Our substring-searchable encryption scheme  $\mathcal{E}$  for an alphabet  $\Sigma$  with  $|\Sigma| = d$  is as follows.

**Gen**( $1^\lambda$ ): Choose random strings  $K_D, K_C, K_L, K_1, K_2, K_3, K_4 \xleftarrow{\text{R}} \{0, 1\}^\lambda$ .<sup>3</sup> The secret key is

$$K = (K_D, K_C, K_L, K_1, K_2, K_3, K_4).$$

**Enc**( $K, s$ ): Let  $s = s_1 \dots s_n \in \Sigma^n$ . Construct a suffix tree  $Tree_s$  for  $s$ .

1. Construct a dictionary  $D$  as follows.

For any node  $u$ , define  $f_1(u) := F_{K_1}(\text{initpath}(u))$  and  $f_2(u) := F_{K_2}(\text{initpath}(u))$ .

For each node  $u$  in  $Tree_s$  (including the root and leaves):

- Let  $v_1, \dots, v_j$  denote the children of  $u$ .
- For  $i = 1, \dots, j$ , let  $g_{2,i} = f_2(v_i)$ .
- For  $i = j + 1, \dots, d$  let  $g_{2,i} \xleftarrow{\text{R}} \{0, 1\}^\lambda$ .
- Choose a random permutation  $\pi_u : [d] \rightarrow [d]$ .
- For  $i = 1, \dots, d$ , let  $f_{2,i}(u) = g_{2,\pi_u(i)}(u)$ .

<sup>3</sup> We will assume for simplicity that  $\Pi.\text{Gen}$  simply chooses a random key  $k \xleftarrow{\text{R}} \{0, 1\}^\lambda$ , so throughout the construction we will use random values as  $\Pi$  keys. To allow for general  $\Pi$  algorithms, instead of using a random value  $r$  directly as a key, we could use a key generated by  $\Pi.\text{Gen}$  with  $r$  providing  $\Pi.\text{Gen}$ 's random coins.

- Let  $X_u = (\text{ind}(u), \text{leafpos}(u), \text{num}(u), \text{len}(u), f_1(u), f_{2,1}(u), \dots, f_{2,d}(u))$ , and then let  $W_u = \Pi.\text{Enc}_{K_D}(X_u)$ .
- Store  $V_u = (f_{2,1}(u), \dots, f_{2,d}(u), W_u)$  with search key  $\kappa_u = f_1(u)$  in  $D$ .

Let  $N$  denote the number of nodes in *Trees*. Construct  $2n - N$  dummy entries in  $D$  as follows. For each dummy entry, choose random strings  $f_1, f_{2,1}, \dots, f_{2,d} \stackrel{\text{R}}{\leftarrow} \{0, 1\}^\lambda$ , and store  $(f_{2,1}, \dots, f_{2,d}, \Pi.\text{Enc}_{K_D}(0))$  with search key  $f_1$  in  $D$ .

2. Construct an array  $C$  as follows: for  $i = 1, \dots, n$ , set  $C[P_{K_3}(i)] = \Pi.\text{Enc}_{K_C}(s_i, i)$ .
3. Construct an array  $L$  as follows: For  $i = 1, \dots, n$ , set  $L[P_{K_4}(i)] = \Pi.\text{Enc}_{K_L}(\text{ind}(\text{leaf}_i), i)$ .

Output the ciphertext  $CT = (D, C, L)$ .

**Query**( $K, p, CT$ ): The interactive query protocol, between a client with  $K$  and  $p$  and a server with  $CT$ , runs as follows.

Let  $p = p_1 \dots p_m \in \Sigma^m$ , and let  $CT = (D, C, L)$ .

1. The client computes, for  $i = 1, \dots, m$ ,  $f_1^{(i)} = F_{K_1}(p_1 \dots p_i)$ ,  $f_2^{(i)} = F_{K_2}(p_1 \dots p_i)$ , and sets  $T_i = \Pi.\text{Enc}_{f_2^{(i)}}(f_1^{(i)})$ . Also, compute  $\text{root} = F_{K_1}(\epsilon)$ . The client sends the server  $(\text{root}, T_1, \dots, T_m)$ .
2. The server proceeds as follows, maintaining variables  $f_1, f_{2,1}, \dots, f_{2,d}, W$ . Initialize  $(f_{2,1}, \dots, f_{2,d}, W)$  to equal  $D[\text{root}]$ .  
For  $i = 1, \dots, m$ :  
For  $j = 1, \dots, d$ :  
Let  $f_1 \leftarrow \Pi.\text{Dec}_{f_{2,j}}(T_i)$ . If  $f_1 \neq \perp$ , update  $(f_{2,1}, \dots, f_{2,d}, W)$  to equal  $D[f_1]$ , and break (proceed to the next value of  $i$ ). Otherwise, do nothing.  
At the end, the server sends  $W$  to the client.
3. The client runs  $X \leftarrow \Pi.\text{Dec}_{K_D}(W)$ . If  $X = \perp$ , output  $\perp$  and end the protocol. Otherwise, parse  $X$  as  $(\text{ind}, \text{leafpos}, \text{num}, \text{len}, f_1, f_{2,1}, \dots, f_{2,d})$ . Check whether  $F_{K_1}(p[1..\text{len}]) = f_1$ . If not, output  $\perp$  and end the protocol. Otherwise, check whether  $\Pi.\text{Dec}(f_{2,i}, T_j) \neq \perp$  for any  $j \in \{\text{len} + 1, \dots, m\}$  and  $i \in \{1, \dots, d\}$ . If so, output  $\perp$  and end the protocol. If  $\text{ind} = 0$ , output  $\emptyset$ . Otherwise, choose a random permutation  $\pi_1 : [m] \rightarrow [m]$ . For  $i = 1, \dots, m$ , let  $x_{\pi_1(i)} = P_{K_3}(\text{ind} + i - 1)$ . The client sends  $(x_1, \dots, x_m)$  to the server.
4. The server sets  $C_i = C[x_i]$  for  $i = 1, \dots, m$  and sends  $(C_1, \dots, C_m)$  to the client.

5. For  $i = 1, \dots, m$ , the client runs  $Y \leftarrow \Pi.\text{Dec}_{K_C}(C_{\pi_1(i)})$ . If  $Y = \perp$ , output  $\perp$  and end the protocol. Otherwise, let the result be  $(p'_i, j)$ . If  $j \neq \text{ind} + i - 1$ , output  $\perp$ . Otherwise, if  $p'_1 \dots p'_m \neq p$ , then the client outputs  $\emptyset$  as its answer and ends the protocol. Otherwise, the client chooses a random permutation  $\pi_2 : [\text{num}] \rightarrow [\text{num}]$ . For  $i = 1, \dots, \text{num}$ , let  $y_{\pi_2(i)} = P_{K_4}(\text{leafpos} + i - 1)$ . The client sends  $(y_1, \dots, y_{\text{num}})$  to the server.
6. The server sets  $L_i = L[y_i]$  for  $i = 1, \dots, \text{num}$ , and sends  $(L_1, \dots, L_{\text{num}})$  to the client.
7. For  $i = 1, \dots, \text{num}$ , the client runs  $\Pi.\text{Dec}_{K_L}(L_{\pi_2(i)})$ . If the result is  $\perp$ , the client outputs  $\perp$  as its answer. Otherwise, let the result be  $(a_i, j)$ . If  $j \neq \text{leafpos} + i - 1$ , output  $\perp$ . Otherwise, output the answer  $A = \{a_1, \dots, a_{\text{num}}\}$ .

### Extension to multiple data strings

While we describe the protocol in terms of a single data string  $s$ , we note that it can easily be extended to support a functionality where the client encrypts a set of strings initially, and then can search for occurrences of a substring within all of them. This is done by building a *generalized suffix tree* [28] that contains suffixes from multiple strings and then using our encryption scheme. For simplicity, however, in the following analysis we restrict ourselves to the single string version.

## 5.5 Efficiency

We will make the standard RAM model assumption that values of size  $O(\log n)$  bits can be read or written in constant time.

We assume encryption and decryption using  $\Pi$  take  $O(\lambda)$  time. Also, we assume the dictionary is implemented in such a way that dictionary lookups take constant time (using hash tables, for example).

### Efficient batch implementation of PRFs

Assuming the evaluation of a PRF takes time linear in the length of its input, in a naive implementation of our scheme, computing the PRFs  $f_1(u)$  and  $f_2(u)$  for all nodes  $u$  would take  $O(n^2)$  time, since the sum of the lengths of the strings  $\text{initpath}(u)$  can be  $O(n^2)$ . Similarly, computing the PRFs used for  $T_1, \dots, T_m$  would take  $O(m^2)$  time. It turns out that we can take advan-

tage of the way the strings we are applying the PRFs to are related, to speed up the batch implementation of the PRFs for all of the nodes of the tree. We will use two tools: the *polynomial hash* and *suffix links* (described below).

The polynomial hash is defined as follows. View a message  $x$  as a sequence  $(x_1, \dots, x_n)$  of  $\ell$ -bit strings. For any  $k$  in the finite field  $\text{GF}(2^\ell)$ , the hash function  $H_k(x)$  is defined as the evaluation of the polynomial  $p_x$  over  $\text{GF}(2^\ell)$  defined by coefficients  $x_1, \dots, x_n$ , at the point  $k$ . That is,  $H_k(x) = p_x(k) = \sum_{i=1}^n x_i k^{i-1}$ , where all operations are in  $\text{GF}(2^\ell)$ . The polynomial hash is an almost universal hash function, so to compute the PRF of a string, we can first apply the polynomial hash and then compute the PRF.

First, we use a trick that is used in the Rabin-Karp rolling hash (see Cormen et al. [15], e.g.). (A rolling hash is a hash function that can be computed efficiently on a sliding window of input; the hash of each window reuses computation from the previous window.) The Rabin-Karp hash is the polynomial hash, with each character of the string viewed as a coefficient of the polynomial applied to the random key of the hash.

The key observation is that the polynomial hash  $H$  allows for constant-time computation of  $H_k(x_1 \dots x_n)$  from  $H_k(x_2 \dots x_n)$ , and also of  $H_k(x_1 \dots x_n)$  from  $H_k(x_1 \dots x_{n-1})$ . To see this, notice that  $H_k(x_1 \dots, x_n) = x_1 + k \cdot H_k(x_2 \dots x_n)$ , and  $H_k(x_1 \dots x_n) = H_k(x_1 \dots x_{n-1}) + x_n k^{n-1}$ . Using this trick, for any string  $x$  of length  $\ell$ , we can compute the hashes  $H_k(x[1..i])$  for all  $i = 1, \dots, m$  in total time  $O(\lambda m)$ . Thus, the  $T_1, \dots, T_m$  can be computed in time  $O(\lambda m)$ .

To compute the hashes of  $\text{initpath}(u)$  for all nodes  $u$  in time  $O(n)$ , we need one more trick. Many efficient suffix tree construction algorithms include *suffix links*. Each non-leaf node  $u$  with associated string  $\text{path}(u) = a||B$ , where  $a$  is a single character, has a pointer called a suffix link pointing to the node  $u'$  such that  $\text{path}(u')$  is  $B$ . It turns out that connecting the nodes in a suffix tree using the suffix links forms another tree, in which the parent of a node  $u$  is the node  $u'$  to which  $u$ 's suffix link points.

Since  $\text{initpath}(u) = \text{path}(\text{par}(u))||u_1$ , where  $\text{par}(u)$  is the parent of  $u$ , we can first compute the hashes of  $\text{path}(u)$  for all non-leaf nodes  $u$ , and then compute  $\text{initpath}(u)$  for each node  $u$  in constant time from  $\text{path}(\text{par}(u))$ . To compute  $\text{path}(u)$  for all nodes  $u$ , we traverse the tree formed by the suffix links, starting at the root, and compute the hash of  $\text{path}(u)$  for each  $u$  using  $\text{path}(u')$ , where  $u'$  is  $u$ 's parent in the suffix link tree.

Each of these computations takes constant time, since  $\text{path}(u)$  is the same as  $\text{path}(u')$  but with one character appended to the front. Therefore, computing the hashes of  $\text{path}(u)$  for all non-leaf nodes  $u$  (and thus, computing the hashes of  $\text{initpath}(u)$  for all nodes  $u$ ) takes total time  $O(n)$ .

### Encryption efficiency

Using the efficient batch implementation of PRFs described above, the PRFs  $f_1(u)$  and  $f_2(u)$  can be computed for all nodes  $u$  in the tree in total time  $O(\lambda n)$ . Therefore, the dictionary  $D$  of  $2n$  entries can be computed in total time  $O(\lambda n)$ . The arrays  $C$  and  $L$  each have  $n$  elements and can be computed in time  $O(\lambda n)$ . The PRPs can actually be implemented by applying a PRF and then sorting the resulting output. Therefore, encryption takes time  $O(\lambda n)$  and the ciphertext is of size  $O(\lambda n)$ .

### Query protocol efficiency

In the query protocol, the client first computes  $T_1, \dots, T_m$ . Using the efficient batch PRF implementation above, computing the  $f_1^{(i)}$  and  $f_2^{(i)}$  for  $i = 1, \dots, m$  takes total time  $O(m)$ , and computing each  $\Pi.\text{Enc}_{f_2^{(i)}}(f_1^{(i)})$  takes  $O(\lambda)$  time, so the total time to compute  $T_1, \dots, T_m$  is  $O(\lambda m)$ .

To find  $W$ , the server performs at most  $md$  decryptions and dictionary lookups, which takes total time  $O(\lambda m)$ .<sup>4</sup> The client then computes  $x_1, \dots, x_m$  and the server retrieves  $C[x_1], \dots, C[x_m]$ , in time  $O(m)$ . If the answer is not  $\emptyset$ , the client then computes  $y_1, \dots, y_{\text{num}}$  and the server retrieves  $L[y_1], \dots, L[y_{\text{num}}]$  in time  $O(\text{num})$ , in time  $O(\text{num})$ . Thus, both the client and the server take computation time  $O(\lambda m + \text{num})$  in the query protocol. (Since we are computing an upper bound on the query computation time, we can ignore the possibility that the server cheats and the client aborts the protocol.) The query protocol takes three rounds of communication, and the total size of the messages exchanged is  $O(\lambda m + \text{num})$ .

## 5.6 Security

Before describing the leakage functions for our scheme, we provide some notation.

<sup>4</sup> We treat the alphabet size  $d$  as a constant.

We first give some notation for the leakage of this scheme. We say that a query  $p$  visits a node  $u$  in the suffix tree  $Tree_s$  if  $\text{initpath}(u)$  is a prefix of  $p$ .

Let:

- $\text{num}_i$  denote the number of occurrences of  $p_i$  as a substring of  $s$
- $\text{ind}_i$  denote the index in  $s$  of the first occurrence of the longest prefix of  $p_i$  that is a substring of  $s$ , or 0 if no such prefix exists
- $\text{leafpos}_i$  denote the index in the tree of the leftmost leaf whose path label has  $p_i$  as a prefix
- $u_{i,j}$  denote the  $j$ th node visited by the query for  $p_i$
- $\text{len}_{i,j}$  denote the length of  $\text{initpath}(u_{i,j})$
- $n_i$  denote the number of nodes visited by the query for  $p_i$

The leakage from queries includes the query prefix pattern, the index intersection pattern, and the leaf intersection pattern, which we now define.

The query prefix pattern for a query  $p_i$  indicates, for each node visited for  $p_i$ , which of the previous queries also visited that node.

**Definition 5.3.** The *query prefix pattern*  $\text{QP}(s, p_1, \dots, p_i)$  is a sequence of length  $n_i$ , where the  $j$ th element is a list  $\text{list}_j$  of indices  $i' < i$  such that the  $i'$ th query also visited  $u_{i,j}$ .

The index intersection pattern for a query  $p_i$  indicates when any of the retrieved indices  $\text{ind}_i, \dots, \text{ind}_i + |p_i| - 1$  are equal to any of the retrieved indices for previous queries.

**Definition 5.4.** The *index intersection pattern*  $\text{IP}(s, p_1, \dots, p_i)$  is a sequence of length  $i$ , where the  $j$ th element is equal to  $\{r_1[\text{ind}_j], \dots, r_1[\text{ind}_j + m_j - 1]\}$  for a fixed random permutation  $r_1 : [n] \rightarrow [n]$ .

The leaf intersection pattern for a query  $p_i$  indicates when any of the retrieved leaves  $\text{leafpos}_i, \dots, \text{leafpos}_i + \text{num}_i - 1$  are equal to any of the retrieved leaves for previous queries.

**Definition 5.5.** The *leaf intersection pattern*  $\text{LP}(s, p_1, \dots, p_i)$  is a sequence of length  $i$ , where the  $j$ th element is equal to  $\{r_2[\text{leafpos}_j], \dots, r_2[\text{leafpos}_j + \text{num}_j - 1]\}$  for a fixed random permutation  $r_2 : [n] \rightarrow [n]$ .

The leakage of the scheme  $\mathcal{E}$  is as follows.  $\mathcal{L}_1(s)$  is just  $n = |s|$ .  $\mathcal{L}_2(s, p_1, \dots, p_i)$  consists of

- $m_i = |p_i|$

- $\{\text{len}_{i,j}\}_{j=1}^{n_i}$
- $\text{QP}(s, p_1, \dots, p_i)$
- $\text{IP}(s, p_1, \dots, p_i)$
- $\text{LP}(s, p_1, \dots, p_i)$

### Leakage Example

To illustrate the leakage of our scheme, consider the following toy example.

Consider the string  $s = \text{“cocoon”}$ , whose suffix tree is shown in Figure 1, and a sequence of three queries,  $p_1 = \text{“co”}$ ,  $p_2 = \text{“coco”}$ , and  $p_3 = \text{“cocoa”}$ .

The query for “co” visits node  $u_2$ , the retrieved indices into  $s$  are 1, 2, and the retrieved leaf positions are 1, 2. The query for “coco” visits nodes  $u_2$  and  $u_3$ , the indices retrieved are 1, 2, 3, 4, and the leaf position retrieved is 1. The query for “cocoa” visits nodes  $u_2$  and  $u_3$ , the indices retrieved are 1, 2, 3, 4, 5, and no leaf positions are retrieved (because there is not a match).

The leakage  $\mathcal{L}_1(s)$  is  $n = 6$ . The leakage  $\mathcal{L}_2$  from all three queries combined is as follows.

- The lengths of the search strings: 2, 4, and 5,
- The lengths 1 and 3 of the initial paths of the nodes  $u_2$  and  $u_3$  visited by the three queries,
- The query prefix pattern, which says that  $p_1, p_2, p_3$  visited the same first node, and then  $p_2$  and  $p_3$  visited the same second node,
- The index intersection pattern, which says that two of the indices returned for  $p_2$  are the same as the two indices returned for  $p_1$ , and four of the indices returned for  $p_3$  are the same as the four indices returned for  $p_2$ , and
- The leaf intersection pattern, which says that the leaf returned for  $p_2$  is one of the two leaves returned for  $p_1$ , and that the queries for  $p_1, p_2$ , and  $p_3$  returned two leaves, one leaf, and no leaves, respectively.

### Security

Our security theorem is as follows.

**Theorem 5.6.** *Let  $\mathcal{L}_1$  and  $\mathcal{L}_2$  be as defined above. If  $F$  is a PRF,  $P$  is a PRP, and  $\Pi$  is an authenticated, key-hiding, symmetric-key encryption scheme, then the substring-searchable encryption scheme  $\mathcal{E}$  satisfies malicious  $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 security.*

The proof is given in Appendix A.

## Discussion

As mentioned in Section 1, our work follows a line of work starting with [16] that allows some information leakage while ensuring that this leakage is formally specified. In all of this work, a major challenge is interpreting the impact of this leakage for any particular application, and this challenge become greater as the schemes become more complex (see e.g. [11] or [31]). Our work presents similar challenges in this respect.

We make a few brief observations. It may be possible to reduce the leakage heuristically at the cost of some efficiency by, for example, having the client cache the result of previous queries and do some of the query evaluation locally, or by incorporating ORAM techniques. However, it seems difficult to reduce the leakage significantly without reducing the efficiency of the scheme significantly. The leakage in our scheme corresponds roughly to the information an adversary would gain if it were allowed to observe only the memory access pattern when a search is evaluated on an unencrypted suffix tree. In this sense, our leakage seems inherent in any suffix-tree-based approach. Constructing an efficient scheme with a significantly less leakage would seem to require a different data structure whose memory access patterns leak less information. We pose the problem of identifying or designing such a data structure as an interesting open problem.

## 6 Conclusion

We presented a definition of queryable encryption schemes and defined security against malicious adversaries making chosen query attacks. Our security definitions are parameterized by leakage functions that specify the information that is revealed about the message and the queries by the ciphertext and the query protocols.

We constructed an efficient substring-searchable encryption scheme – a queryable encryption scheme that supports finding all occurrences of a search string  $p$  as a substring of an encrypted string  $s$ . Our approach is based on suffix trees. Our construction uses only basic symmetric-key primitives (pseudorandom functions and permutations and an authenticated, which-key-concealing encryption scheme). The ciphertext size and encryption time are  $O(\lambda n)$  and query time and message size are  $O(\lambda m + k)$ , where  $\lambda$  is the security parameter,  $n$  is the length of the string,  $m$  is the length of the search string, and  $k$  is the number of occurrences

of the search string. Querying requires three rounds of communication.

While we have given a formal characterization of the leakage of our substring-searchable encryption scheme, it is an open problem to analyze the practical cost of the leakage. Given the leakage from several typical queries, what can a server infer about the message and the queries? We believe our scheme provides a worthwhile efficiency/leakage tradeoff for many applications, especially when current alternatives are either no encryption at all or existing searchable encryption schemes that are designed for keyword search but inefficient for substring search.

## Acknowledgments

We thank Ben Fuller, Seny Kamara, Ron Rivest, Mayank Varia, and Arkady Yerukhimovich for helpful discussions and comments on the manuscript. Additionally, we thank the anonymous reviewers of this paper for useful suggestions.

## References

- [1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.
- [2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, June 1975.
- [3] J. Baron, K. El Defrawy, K. Minkovich, R. Ostrovsky, and E. Tressler. 5PM: Secure pattern matching. In I. Visconti and R. D. Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 222–240. Springer, Sept. 2012.
- [4] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer, Dec. 2000.
- [5] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In T. Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 317–330. Springer, Dec. 2000.
- [6] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 111–131. Springer, Aug. 2011.
- [7] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In Y. Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Mar. 2011.
- [8] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.
- [9] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In R. Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, Oct. 2011.
- [10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, Jan. 2012.
- [11] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Aug. 2013. 10.1007/978-3-642-40041-4\_20.
- [12] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, Feb. 2014.
- [13] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 577–594. Springer, Dec. 2010.
- [14] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 151–168. Springer, Aug. 2011.
- [15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- [16] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. Vimercati, editors, *ACM CCS 06*, pages 79–88. ACM Press, Oct. / Nov. 2006.
- [17] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multi-party computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Aug. 2012.
- [18] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Mar. 2009.
- [19] M. Farach. Optimal suffix tree construction with large alphabets. In *38th FOCS*, pages 137–143. IEEE Computer Society Press, Oct. 1997.
- [20] M. Fischlin. Pseudorandom function tribe ensembles based on one-way permutations: Improvements and applications. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 432–445. Springer, May 1999.
- [21] C. Fletcher, M. van Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC 2012*, 2012.
- [22] K. B. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. In *DBSec '09*, pages 81–94, 2009.
- [23] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Aug. 2010.
- [24] R. Gennaro, C. Hazay, and J. S. Sorensen. Text search protocols with simulation based security. In P. Q. Nguyen and D. Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 332–350. Springer, May 2010.
- [25] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.
- [26] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Aug. 2012.
- [27] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, Apr. 2012.
- [28] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [29] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Journal of Cryptology*, 23(3):422–456, July 2010.
- [30] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 258–274. Springer, Apr. 2013. 10.1007/978-3-642-39884-1\_22.
- [31] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 965–976. ACM Press, Oct. 2012.
- [32] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Develop-*

- ment, 31(2):249–260, March 1987.
- [33] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.
  - [34] J. Katz and L. Malka. Secure text processing with applications to private DNA matching. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM CCS 10*, pages 485–492. ACM Press, Oct. 2010.
  - [35] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In B. Schneier, editor, *FSE 2000*, volume 1978 of *LNCS*, pages 284–299. Springer, Apr. 2001.
  - [36] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In N. P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 146–162. Springer, Apr. 2008.
  - [37] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2): 323–350, 1977.
  - [38] K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In A. D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 285–298. Springer, Feb. / Mar. 2012.
  - [39] P. Mohassel, S. Niksefat, S. S. Sadeghian, and B. Sadeghiyan. An efficient protocol for oblivious DFA evaluation and applications. In O. Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 398–415. Springer, Feb. / Mar. 2012.
  - [40] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Aug. 2012.
  - [41] R. Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, MIT, 1992.
  - [42] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 457–473. Springer, Mar. 2009.
  - [43] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS 2012*. The Internet Society, Feb. 2012.
  - [44] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC '12*, 2012.
  - [45] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*. The Internet Society, Feb. 2014.
  - [46] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient dna searching through oblivious automata. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM CCS 07*, pages 519–528. ACM Press, Oct. 2007.
  - [47] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

## A Security Against Malicious Adversaries

We now prove that our substring-searchable encryption scheme satisfies malicious- $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 security for the leakage functions  $\mathcal{L}_1$  and  $\mathcal{L}_2$  defined in Section 5.6.

**Theorem A.1.** *Let  $\mathcal{L}_1$  and  $\mathcal{L}_2$  be defined as in Section 5.6. If  $F$  is a PRF,  $P$  is a PRP, and  $\Pi$  is a key-hiding, authenticated symmetric-key encryption scheme, then the substring-searchable encryption scheme  $\mathcal{E}$  satisfies malicious  $(\mathcal{L}_1, \mathcal{L}_2)$ -CQA2 security.*

*Proof.* As explained in Section 4.2, we show that our scheme leaks only  $\mathcal{L}_1, \mathcal{L}_2$ , by showing that no adversary can distinguish interaction with the real client from interaction with a simulator who does not know  $s$  or the queries  $q_j$ , but is only given the leakage defined by  $\mathcal{L}_1, \mathcal{L}_2$ .

We define a simulator  $\mathcal{S}$  that works as follows.  $\mathcal{S}$  first chooses random keys  $K_D, K_C, K_L \xleftarrow{R} \{0, 1\}^\lambda$ .

### Ciphertext

Given  $\mathcal{L}_1(s) = n$ ,  $\mathcal{S}$  constructs a simulated ciphertext as follows.

1. Construct a dictionary  $D$  as follows. For  $i = 1, \dots, 2n$ , choose fresh random values  $\kappa_i, f_{2,1}, \dots, f_{2,d}, \xleftarrow{R} \{0, 1\}^\lambda$ , and then for each  $i$  store  $V_i = (f_{2,1}, \dots, f_{2,d}, W = \Pi.\text{Enc}(K_D, 0))$  with search key  $\kappa_i$  in  $D$ .<sup>5</sup>
2. Choose an arbitrary element  $\sigma_0 \in \Sigma$ . Construct an array  $C$ , where  $C[i] = \Pi.\text{Enc}(K_C, (\sigma_0, 0))$  for  $i = 1, \dots, n$ .
3. Construct an array  $L$ , where  $L[i] = \Pi.\text{Enc}(K_L, 0)$  for  $i = 1, \dots, n$ .

Output  $CT = (D, C, L)$ .

### Tables

In order to simulate the query protocol,  $\mathcal{S}$  will need to do some bookkeeping.

<sup>5</sup> Throughout our description, we assume each key is only used for a fixed message space. Thus, for example, 0 here is padded to the appropriate length to match the messages encrypted under  $K_D$  in the real scheme.

$\mathcal{S}$  will maintain two tables  $\mathcal{T}_1$  and  $\mathcal{T}_2$ , both initially empty.  $\mathcal{T}_1$  contains all currently defined tuples  $(i, j, \kappa)$  such that the entry in  $D$  with search key  $\kappa$  represents the  $j$ th node visited by the  $i$ th query. We write  $\mathcal{T}_1(i, j) = \kappa$  if  $(i, j, \kappa)$  is an entry in  $\mathcal{T}_1$ .

$\mathcal{T}_2$  contains all currently defined tuples  $(\kappa, f_2, \text{flag}, \text{flag}_1, \dots, \text{flag}_d)$ , where for the node  $u$  represented by the entry  $D[\kappa]$ ,  $\kappa = f_1(u)$ ,  $f_2 = f_2(u)$ , **flag** indicates whether  $u$  has been visited by any query, and **flag<sub>i</sub>** indicates whether  $\text{child}(u, \pi_u(i))$  has been visited. The value of each flag is either “visited” or “unvisited”. We write  $\mathcal{T}_2(\kappa) = (f_2, \text{flag}, \text{flag}_1, \dots, \text{flag}_d)$  if  $(\kappa, f_2, \text{flag}, \text{flag}_1, \dots, \text{flag}_d)$  is an entry in  $\mathcal{T}_2$ .

Choose an arbitrary entry  $(\kappa^*, V^*)$  in  $D$  to represent the root node of  $\text{Tree}_s$ . In  $\mathcal{T}_2(\kappa^*)$ , set all flags to “unvisited” and set  $f_2 = 0$ . (The  $f_2$  for the root node will never be used, so it is fine to set it to 0.) We implicitly define  $\mathcal{T}_1(i, 0) = \kappa^*$  for all  $i$ .

### Query protocols

For the  $j$ th token query  $p_j$ ,  $\mathcal{S}$  is given  $\mathcal{L}_2(s, p_1, \dots, p_j)$ , which consists of  $m_j = |p_j|$ ,  $\{\text{len}_{j,i}\}_{i=1}^{n_j}$ ,  $\text{QP}(s, p_1, \dots, p_j)$ ,  $\text{IP}(s, p_1, \dots, p_j)$ , and  $\text{LP}(s, p_1, \dots, p_j)$ .

For  $t = 1, \dots, n_j$ , if  $\text{list}_t = \text{QP}(p_j, s)[t]$  is non-empty (i.e., the node  $u_{j,t}$  was visited by a previous query), let  $j'$  be one of the indices in  $\text{list}_t$ . Let  $\kappa = \mathcal{T}_1(j', t)$  and let  $(f_2, \text{flag}, \text{flag}_1, \dots, \text{flag}_d) = \mathcal{T}_2(\kappa)$ .  $T_{\text{len}_{j,t}} = \Pi.\text{Enc}(f_2, \kappa)$ . Set  $\mathcal{T}_1(j, t) = \kappa$ .

If instead  $\text{list}_t$  is empty, choose a random unused entry  $(\kappa, V)$  in  $D$  to represent the node  $u_{j,t}$ , and set  $\mathcal{T}_1(j, t) = \kappa$ . Let  $\kappa' = \mathcal{T}_1(j, t-1)$  and let  $(f_2, \text{flag}, \text{flag}_1, \dots, \text{flag}_d) = \mathcal{T}_2(\kappa')$ . Choose a random  $i \in \{1, \dots, d\}$  such that **flag<sub>i</sub>** is “unvisited”, and set **flag<sub>i</sub>** to “visited”. Let  $f_{2,i}$  be  $D[\kappa']$ .  $f_{2,i}$ . Set  $T_{\text{len}_{j,t}} = \Pi.\text{Enc}(f_{2,i}, \kappa)$ , set  $\mathcal{T}_2(\kappa).f_2 = f_{2,i}$ , set  $\mathcal{T}_2(\kappa).\text{flag}$  to “visited”, and set  $\mathcal{T}_2(\kappa).\text{flag}_i$  to “unvisited” for  $i = 1, \dots, d$ .

For any  $i \neq \text{len}_t$  for any  $t = 1, \dots, n_j$ , choose a random  $f_2 \xleftarrow{\text{R}} \{0, 1\}^\lambda$ , and let  $T_i = \Pi.\text{Enc}(f_2, 0)$ .

Let  $\kappa^*$  be the key chosen for the root entry of the dictionary in the simulated ciphertext. Send  $(\kappa^*, T_1, \dots, T_m)$  to the adversary.

Upon receiving a  $W$  from the adversary, check whether  $W = D[\mathcal{T}_1(j, n_j)].W$ . If not, output  $\perp$  and set the flag  $f_j$  to  $\perp$ . Otherwise, let  $(x_1, \dots, x_m)$  be a random ordering of the elements of the set  $\text{IP}(p_j, s)[j]$ , and send  $(x_1, \dots, x_m)$  to the adversary. (If  $\text{IP}(p_j, s)[j]$  shows no indices for the  $j$ th query, then end the protocol.)

Upon receiving  $C_1, \dots, C_m$  from the adversary, check whether  $C_i = C[x_i]$  for each  $i$ . If not, output  $\perp$  and set the flag  $f_j$  to  $\perp$ . Otherwise, let  $(y_1, \dots, y_{\text{num}})$  be a random ordering of the elements of  $\text{LP}(p_j, s)[j]$ , and send  $(y_1, \dots, y_{\text{num}})$  to the adversary. (If  $\text{LP}(p_j, s)[j]$  shows no leaf positions for the  $j$ th query, then end the protocol.)

Upon receiving  $L_1, \dots, L_{\text{num}}$  from the adversary, check whether  $L_i = L[y_i]$  for each  $i$ . If not, output  $\perp$  and set the flag  $f_i$  to  $\perp$ .

This concludes the description of the simulator  $\mathcal{S}$ .

### Sequence of games

We now show that the real and ideal experiments are indistinguishable by any PPT adversary  $\mathcal{A}$  except with negligible probability. To do this, we consider a sequence of games  $G_0, \dots, G_{17}$  that gradually transform the real experiment into the ideal experiment. We will show that each game is indistinguishable from the previous one, except with negligible probability.

**Game  $G_0$ .** This game corresponds to an execution of the real experiment, namely,

- The challenger begins by running  $\text{Gen}(1^\lambda)$  to generate a key  $K$ .
- The adversary  $\mathcal{A}$  outputs a string  $s$  and receives  $CT \leftarrow \text{Enc}(K, s)$  from the challenger.
- $\mathcal{A}$  adaptively chooses search strings  $p_1, \dots, p_q$ . For each  $p_i$ ,  $\mathcal{A}$  first interacts with the challenger, who is running the client part of **Query** honestly with input  $(K, p_i)$ . Then the challenger sends its output from **Query** to  $\mathcal{A}$ .

**Game  $G_1$ .** This game is the same as  $G_0$ , except that in  $G_1$  the challenger is replaced by a simulator that does not generate keys  $K_1, K_2$  and replaces  $F_{K_1}$  and  $F_{K_2}$  with random functions. Specifically, the simulator maintains tables  $R_1, R_2$ , initially empty. Whenever the challenger in  $G_0$  computes  $F_{K_i}(x)$  for some  $x$ , the simulator uses  $R_i(x)$  if it is defined; otherwise, it chooses a random value from  $\{0, 1\}^\lambda$ , stores it as  $R_i(x)$ , and uses that value.

A straightforward hybrid argument shows that  $G_1$  is indistinguishable from  $G_0$  by the PRF property of  $F$ .

**Game  $G_2$ .** This game is the same as  $G_1$ , except that in  $G_2$  the simulator does not generate keys  $K_3, K_4$  and replaces  $P_{K_3}$  and  $P_{K_4}$  with random permutations. Specifically, the simulator maintains tables  $R_3$  and  $R_4$ , initially empty. Whenever the simulator in  $G_1$  computes  $P_{K_i}(x)$  for some  $x$ , the simulator in  $G_2$  uses  $R_i(x)$ , if it is defined; otherwise, it chooses a



random value in  $[n]$  that has not yet been defined as  $R_i(y)$  for any  $y$ , and uses that value.

A straightforward hybrid argument shows that  $G_1$  and  $G_2$  are indistinguishable by the PRP property of  $P$ .

**Game  $G_3$ .** This is the same as  $G_2$ , except that we modify the simulator as follows. For any query, when the simulator receives a  $W$  from the adversary in response to  $T_1, \dots, T_m$ , the simulator's decision whether to output  $\perp$  will not be based on the decryption of  $W$ . Instead, it will output  $\perp$  if  $W$  is not the ciphertext in the dictionary entry  $D[R_1(p[1..i])]$ , where  $p[1..i]$  is the longest matching prefix of  $p$ . Otherwise, the simulator proceeds as in  $G_2$ .

We argue that games  $G_2$  and  $G_3$  are indistinguishable by the ciphertext integrity of  $\Pi$ .

**Lemma A.2.** *If  $\Pi$  has ciphertext integrity, then  $G_2$  and  $G_3$  are indistinguishable, except with negligible probability.*

*Proof.* We analyze the cases in which  $G_2$  and  $G_3$  each output  $\perp$  in response to a  $W$ .

$G_2$  runs  $\Pi.\text{Dec}(K_D, W)$  to get either  $\perp$  or a tuple  $X$ , which it parses as  $(\text{ind}, \text{leafpos}, \text{num}, \text{len}, f_1, f_{2,1}, \dots, f_{2,d})$ .  $G_2$  outputs  $\perp$  if any of the following events occur:

- (Event  $W.1$ )  $\Pi.\text{Dec}(K_D, W) = \perp$ , or
- (Event  $W.2$ )  $W$  decrypts successfully, but  $f_1 \neq R_1(p[1..\text{len}])$ , or
- (Event  $W.3$ )  $W$  decrypts successfully and  $f_1 = R_1(p[1..\text{len}])$ , but  $\Pi.\text{Dec}(f_{2,i}, T_j) \neq \perp$  for some  $i \in \{1, \dots, d\}$ ,  $j > \text{len}$ .

$G_3$  outputs  $\perp$  if  $W$  is not the ciphertext in the dictionary entry  $D[R_1(p[1..i])]$ , where  $p[1..i]$  is the longest matching prefix of  $p$ , which is the case if any of the following events occur:

- (Event  $W.1'$ )  $W$  is not a ciphertext in  $D$ ,
- (Event  $W.2'$ )  $W$  is a ciphertext in  $D$  but not for any prefix of  $p$ . That is,  $W = D[\kappa]$  where  $\kappa$  is not equal to  $R_1(p[1..i])$  for any  $i$ .
- (Event  $W.3'$ )  $W$  is a ciphertext in  $D$  for a prefix of  $p$ , but there is a longer matching prefix of  $p$ . That is,  $W = D[R_1(p[1..i])]$  for some  $i$ , but there exists a  $j > i$  such that there is an entry  $D[R_1(p[1..j])]$ .

If  $G_3$  outputs  $\perp$  in response to  $W$  for any query, then  $G_2$  also outputs  $\perp$ : If event  $W.1'$  occurs, then  $W.1$  occurs with all but negligible probability by the ciphertext integrity of  $\Pi$ . If event  $W.2'$  occurs, then event  $W.2$  occurs with probability all but at most

$1/2^\lambda$  (the probability that  $F_{K_1}(p[1..\text{len}]) = f_1$  when  $f_1$  is an independent, random value). If event  $W.3'$  occurs, then clearly  $W.3$  also occurs.

If  $G_2$  outputs  $\perp$ , then  $G_3$  also outputs  $\perp$ , since if  $W$  is the ciphertext in  $D[R_1(p[1..i])]$ , then  $W$  will decrypt successfully, with  $f_1 = R_1(p[1..\text{len}])$ , and  $\Pi.\text{Dec}(f_{2,k}, T_j) = \perp$  for all  $k \in \{1, \dots, d\}$ ,  $j > i$ .

Thus,  $G_2$  and  $G_3$  are indistinguishable except with negligible probability.  $\square$

**Game  $G_4$ .** This is the same as  $G_3$ , except that we modify the simulator as follows. For any query, when the simulator receives  $C_1, \dots, C_m$  from the adversary in response to indices  $x_1, \dots, x_m$ , the simulator's decision whether to output  $\perp$  is not based on the decryptions of  $C_1, \dots, C_m$ . Instead, it outputs  $\perp$  if  $C_i \neq C[x_i]$  for any  $i$ . Otherwise, the simulator proceeds as in  $G_3$ .

We argue that games  $G_4$  and  $G_3$  are indistinguishable by the ciphertext integrity of  $\Pi$ .

**Lemma A.3.** *If  $\Pi$  has ciphertext integrity, then  $G_3$  and  $G_4$  are indistinguishable, except with negligible probability.*

*Proof.* We analyze the cases in which  $G_3$  and  $G_4$  each output  $\perp$  in response to  $C_1, \dots, C_m$ .

For each  $i$ ,  $G_3$  outputs  $\perp$  if either of the following events occur:

- (Event  $C.1$ )  $\Pi.\text{Dec}(K_C, C_i) = \perp$ , or
- (Event  $C.2$ )  $\Pi.\text{Dec}(K_C, C_i) = (p'_i, j)$  where  $j$  is not the correct index.

For each  $i$ ,  $G_4$  outputs  $\perp$  if  $C_i \neq C[x_i]$ , which happens if either of the following events occur:

- (Event  $C.1'$ )  $C_i$  is not among  $C[1], \dots, C[n]$ , or
- (Event  $C.2'$ )  $C_i = C[k]$  where  $k \neq x_i$ .

If  $G_4$  outputs  $\perp$  for some  $i$  then  $G_3$  outputs  $\perp$  except with negligible probability: For any  $i$ , if event  $C.1'$  occurs, then event  $C.1$  occurs with all but negligible probability, by the ciphertext integrity of  $\Pi$ . If event  $C.2'$  occurs, then event  $C.2$  occurs, since if  $C_i = C[k]$  for some  $k \neq x_i$ ,  $C_i$  will decrypt to  $(s_j, j)$  for an incorrect index  $j$ .

If  $G_3$  outputs  $\perp$ , if event  $C.1$  occurred, then  $C.1'$  also occurred, since  $C_i$  will decrypt successfully if it is one of  $C[1], \dots, C[n]$ . If event  $C.2$  occurred, then either  $C.1'$  or  $C.2'$  occurred, since  $C_i$  will decrypt to the correct value if  $C_i = C[x_i]$ . Therefore, if  $G_3$  outputs  $\perp$  for some  $i$ , so does  $G_4$ .

Thus,  $G_3$  and  $G_4$  are indistinguishable except with negligible probability.  $\square$

**Game  $G_5$ .** This game is the same as  $G_4$ , except for the following differences. The simulator does not decrypt the  $C_1, \dots, C_m$  from the adversary. For any query  $p$ , instead of deciding whether to output  $\emptyset$  based on the decryptions of  $C_1, \dots, C_m$ , the simulator outputs  $\emptyset$  if  $p$  is not a substring of  $s$ . Otherwise, the simulator proceeds as in  $G_4$ .

As we showed in Lemmas A.2 and A.3, if the adversary does not send the correct  $W$ , the client will respond with  $\perp$ , and if the adversary does not send the correct  $C_1, \dots, C_m$ , the client will also respond with  $\perp$ . Therefore, if the simulator has not yet output  $\perp$  when it is deciding whether to output  $\emptyset$ , then  $C_1, \dots, C_m$  are necessarily the correct ciphertexts, and the decryptions  $p'_1, \dots, p'_m$  computed in  $G_4$  match  $p$  if and only if  $p$  is a substring of  $s$ . Therefore,  $G_4$  and  $G_5$  are indistinguishable.

**Game  $G_6$ .** This game is the same as  $G_5$ , except that in  $G_6$ , for  $i = 1, \dots, n$ , instead of setting  $c_i = \Pi.\text{Enc}(K_C, (s_i, i))$ , the simulator sets  $c_i = \Pi.\text{Enc}(K_C, (\sigma_0, 0))$ , where  $\sigma_0$  is an arbitrary element of  $\Sigma$ .

Note that in both  $G_5$  and  $G_6$ ,  $K_C$  is hidden and the  $c_i$ 's are never decrypted. A hybrid argument shows that games  $G_5$  and  $G_6$  are indistinguishable by CPA security of  $\Pi$ .

**Game  $G_7$ .** This game is the same as  $G_6$ , except that we eliminate the use of the random permutation  $R_3$ , in the following way. For  $i = 1, \dots, n$ , the simulator set  $C[i] = c_i$  instead of  $C[R_3(i)] = c_i$ , where  $c_i = \Pi.\text{Enc}(K_C, (\sigma_0, 0))$ . Furthermore, for any query  $p_j$ , the simulator is given an additional input  $\text{IP}(s, p_1, \dots, p_j)$  (as defined in Section 5.6). To generate  $(x_1, \dots, x_m)$  in the query protocol, the simulator outputs a random ordering of the elements in  $\text{IP}(s, p_1, \dots, p_j)[j]$ .

Since each  $c_i$  is an encryption under  $K_C$  of  $(\sigma_0, 0)$ , it does not matter whether the  $c_i$ 's are permuted in  $C$ ; if we permute the  $c_i$ 's or not, the result is identical. After we eliminate the use of  $R_3$  in generating  $C$ ,  $R_3$  is only used by the simulator to compute  $(x_1, \dots, x_m)$ . Thus, we can replace the computation of  $(x_1, \dots, x_m)$  for each query  $p_j$  with a random ordering of the elements of  $\text{IP}(s, p_1, \dots, p_j)[j]$ , and the result will be identical.

**Game  $G_8$ .** This is the same as  $G_7$ , except that we modify the simulator as follows. For any query, when the simulator receives  $L_1, \dots, L_{\text{num}}$  from the adversary in response to indices  $y_1, \dots, y_{\text{num}}$ , the simulator's decision whether to output  $\perp$  is not based on the decryptions of the  $L_1, \dots, L_{\text{num}}$ ; instead, it outputs

$\perp$  if  $L_i \neq L[y_i]$  for any  $i$ ; otherwise, it proceeds to compute the answer  $A$  as in  $G_7$ .

**Lemma A.4.** *If  $\Pi$  has ciphertext integrity, then  $G_3$  and  $G_4$  are indistinguishable, except with negligible probability.*

This follows from a very similar argument to the proof of Lemma A.3.

**Game  $G_9$ .** This game is the same as  $G_8$ , except for the following differences. The simulator does not decrypt the  $L_1, \dots, L_{\text{num}}$  from the adversary. For any query  $p_j$ , instead of computing the answer  $A_j$  using the decryptions of  $L_1, \dots, L_{\text{num}}$ , if  $A_j$  has not already been set to  $\perp$  or  $\emptyset$ , the simulator outputs  $A_j = \mathcal{F}(s, p_j)$ .

As we showed in Lemmas A.2, A.3, and A.4, if any of the  $W$ ,  $C_1, \dots, C_m$  or  $L_1, \dots, L_{\text{num}}$  from the adversary are incorrect, the client will respond to the incorrect message with  $\perp$ .

Moreover, if the simulator has not yet output  $\perp$  when it is computing  $A_j$ , then it follows directly from the protocol description that the output will be  $A_j = \mathcal{F}(s, p_j)$  (by correctness of  $\mathcal{E}$ ). Therefore,  $G_8$  and  $G_9$  are indistinguishable.

**Game  $G_{10}$ .** This game is the same as  $G_9$ , except that in  $G_{10}$ , for each  $i = 1, \dots, n$ , the simulator generates each  $\ell_i$  as  $\Pi.\text{Enc}(K_L, 0)$  instead of  $\Pi.\text{Enc}(K_L, (\text{ind}_{\text{leaf}_i}, i))$ .

A straightforward hybrid argument shows that  $G_9$  and  $G_{10}$  are indistinguishable by the CPA security of  $\Pi$ .

**Game  $G_{11}$ .** This game is the same as  $G_{10}$ , except that we eliminate the use of the random permutation  $R_4$ , in the following way. For  $i = 1, \dots, n$ , the simulator set  $L[i] = \ell_i$  instead of  $L[R_4(i)] = \ell_i$ , where  $\ell_i = \Pi.\text{Enc}(K_L, 0)$ . Furthermore, for any query  $p_j$ , the simulator is given an additional input  $\text{LP}(s, p_1, \dots, p_j)$  (as defined in Section 5.6). To generate  $(y_1, \dots, y_{\text{num}})$  in the query protocol, the simulator outputs a random ordering of the elements in  $\text{LP}(s, p_1, \dots, p_j)[j]$ .

The argument for game  $G_{11}$  is analogous to the one for game  $G_7$ . Since each  $\ell_i$  is an encryption under  $K_L$  of 0, it does not matter whether the  $\ell_i$ 's are permuted in  $L$ ; if we permute the  $\ell_i$ 's or not, the result is identical. After we eliminate the use of  $R_4$  in generating  $L$ ,  $R_4$  is only used by the simulator to compute  $(y_1, \dots, y_{\text{num}})$ . Thus, we can replace the computation of  $(y_1, \dots, y_{\text{num}})$  for each query  $p_j$  with a random ordering of the elements of  $\text{LP}(s, p_1, \dots, p_j)[j]$ , and the result will be identical.

**Game  $G_{12}$ .** This is the same as  $G_{11}$ , except that the simulator in  $G_{12}$  does not decrypt the  $W$  from the adversary in the query protocol.

Since the simulator in  $G_{11}$  no longer uses any values from the decryption of  $W$ ,  $G_{12}$  is indistinguishable from  $G_{11}$ .

**Game  $G_{13}$ .** This is the same as  $G_{12}$ , except that in  $G_{13}$ , for each node  $u$  the simulator generates  $W_u$  as  $\Pi.\text{Enc}(K_D, 0)$  instead of  $\Pi.\text{Enc}(K_D, X_u)$ .

A straightforward hybrid argument shows that  $G_{12}$  and  $G_{13}$  are indistinguishable by the CPA security of  $\Pi$ .

**Game  $G_{14}$ .** This is the same as game  $G_{13}$ , except that in the query protocol, for any non-matching prefix  $p[1..i]$ , the simulator replaces  $T_i$  with an encryption under a fresh random key. That is, for any query  $p$ , for any prefix  $p[1..i]$ ,  $i = 1, \dots, m$ , if  $p[1..i]$  is a non-matching prefix, the simulator chooses a fresh random value  $r$  and sets  $T_i \leftarrow \Pi.\text{Enc}(r, R_1(p[1..i]))$ ; otherwise, it sets  $T_i \leftarrow \Pi.\text{Enc}(R_2(p[1..i]), R_1(p[1..i]))$  as in game  $G_{13}$ .

For any  $k$  and  $i$ , let  $p_k$  denote the  $k$ th query, and let  $T_{k,i}$  denote the  $T_i$  produced by the simulator for the  $k$ th query. The only way an adversary  $\mathcal{A}$  may be able to tell apart  $G_{13}$  and  $G_{14}$  is if two queries share a non-matching prefix; that is, there exist  $i, j, j'$  such that  $j \neq j'$  and  $p_j[1..i] = p_{j'}[1..i]$ . In this case,  $G_{14}$  will use different encryption keys to generate  $T_{i,j}$  and  $T_{i,j'}$ , while  $G_{13}$  will use the same key. Note that the decryption keys for  $T_{i,j}$  and  $T_{i,j'}$  will never be revealed to  $\mathcal{A}$  in either game. Thus, a straightforward hybrid argument shows that  $G_{13}$  and  $G_{14}$  are indistinguishable by the key-hiding property of  $\Pi$ .

**Game  $G_{15}$ .** This is the same as game  $G_{14}$ , except that in the query protocol for any string  $p$ , for any non-matching prefix  $p[1..i]$ , the simulator replaces  $T_i$  with an encryption of 0. That is, for any query  $p$ , for any prefix  $p[1..i]$ ,  $i = 1, \dots, m$ , if  $p[1..i]$  is non-matching, the simulator chooses a fresh random value  $r$  and sets  $T_i \leftarrow \Pi.\text{Enc}(r, 0)$ ; otherwise, it sets  $T_i \leftarrow \Pi.\text{Enc}(R_2(p[1..i]), R_1(p[1..i]))$  as in game  $G_{14}$ . The only way an adversary  $\mathcal{A}$  may be able to tell apart  $G_{14}$  and  $G_{15}$  is if a prefix  $p_j[1..i]$  is non-matching. In this case, in  $G_{15}$ ,  $T_{j,i}$  will be an encryption of 0, while in  $G_{14}$ ,  $T_{j,i}$  will be an encryption of  $R_1(p_j[1..i])$ . The decryption key for  $T_{j,i}$  will never be revealed to  $\mathcal{A}$  in either game. Thus, a straightforward hybrid argument shows that games  $G_{14}$  and  $G_{15}$  are indistinguishable by the CPA security of  $\Pi$ .

**Game  $G_{16}$ .** This is the final game, which corresponds to an execution of the ideal experiment. In  $G_{16}$ , the

simulator is replaced with the simulator  $\mathcal{S}$  defined above.

The differences between  $G_{15}$  and  $G_{16}$  are as follows. In  $G_{16}$ , the simulator no longer uses the string  $s$  when creating the dictionary  $D$ , and for each query  $p$ , it no longer uses  $p$  when creating  $T_1, \dots, T_m$ . When constructing  $D$ , whenever the simulator in  $G_{15}$  generates a value by applying a random function to a string,  $\mathcal{S}$  generates a fresh random value without using the string. Note that all of the  $\text{initpath}(u)$  strings used in  $D$  are unique, so  $\mathcal{S}$  does not need to ensure consistency between any of the random values, thus the resulting  $D$  will clearly be identical. For any query  $p_j$ , for each matching prefix  $p_j[1..i]$ ,  $\mathcal{S}$  constructs  $T_i$  to be consistent with  $D$  and with prefix queries using the query prefix pattern  $\text{QP}(s, p_1, \dots, p_j)$ . The simulator in game  $G_{15}$  behaves the same except that it again uses random functions (applied to distinct strings) in place of randomly sampled strings; as above the result is identical. Also, while the simulator in  $G_{15}$  associates entries in  $D$  to strings when it first constructs  $D$ ,  $\mathcal{S}$  associates entries in  $D$  to strings as it answers each new query; this however has no effect on the game. Finally, in game  $G_{15}$ , after each query the simulator outputs either  $\perp$  or  $A_j = \mathcal{F}(s, p_j)$  which is sent to the adversary. In game  $G_{16}$  the simulator outputs a flag  $f_j = \perp$  exactly when the  $G_{15}$  simulator outputs  $\perp$ . By definition of the ideal game, the game sends  $\perp$  to the adversary whenever  $f_j = \perp$  and sends  $\mathcal{F}(s, p_j)$  otherwise. Thus, both simulators produce identical views. □