Pattern Matching Encryption

Melissa Chase
Microsoft Research
melissac@microsoft.com

Emily Shen
MIT Lincoln Laboratory
emily.shen@ll.mit.edu

August 18, 2014

Abstract

In this paper, we consider a setting where a user wants to outsource storage of a large amount of private data, and then perform pattern matching queries on the data; that is, given a data string s and a "pattern" string p, find all occurrences of p as a substring of s.

We formalize the security properties desired in this type of setting by defining a type of encryption called *queryable encryption*. In a queryable encryption scheme, a user can encrypt a message M under a secret key, and using the secret key can generate tokens for queries q. Applying a token for a query q to an encryption of M gives the answer to the query q on M. We consider security against both honest-but-curious and malicious adversaries, and define properties guaranteeing both the correctness of the user's results and the privacy of the user's data. Following the line of work started by [CGKO06], to allow for efficient constructions, we allow the protocol to leak some information about the user's data, however we ensure that this leakage can be precisely captured in the definition. In addition, we allow the query protocol to involve a small constant number of rounds of interaction.

We construct a queryable encryption scheme for pattern matching queries that is correct and secure in the malicious model. Our construction is based on efficient symmetric-key building blocks and scales well with the size of the input: encryption of a data string of length n with security parameter λ takes $O(n)$ time and produces a ciphertext of size $O(n\lambda)$, and a query for a pattern string of length m that occurs k times takes $O(m + k)$ time and three rounds of communication.

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 message 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 in the setting where the data consists of a set of documents, and the client wishes to search for combinations of keywords in those documents. (These solutions are often referred to as symmetric searchable encryption or SSE; see Section 1.1 below.) But what about other types of data or other types of queries? For example, suppose a medical research lab wants to store its 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 and query the data efficiently. Here we consider the case where researchers want to be able to make substring queries on the stored data (e.g., to search for genetic markers). The owner of the data would like that the process of performing these queries not reveal much information to the server about the genomic data or the search strings.

We note that this problem could be solved with SSE, by considering every substring as a separate keyword; however, this approach would result in a fairly significant storage blowup (SSE scales linearly with the total number of keywords). Our goal here is to avoid this storage overhead and achieve efficiency comparable to the unencrypted scenario.

Queryable encryption. In this paper, we formalize a type of encryption that we call *queryable encryption*. 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 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 pattern matching queries, a query q is a pattern 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.

For security, we will think of the server as an adversary trying to learn information about the message and the queries. Ideally, we would like that an adversary that is given a ciphertext and that engages in query protocols for several queries learns nothing about the message or the queries. However, in order to achieve an 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. We define notions of security that specify explicitly what information is leaked, and guarantee that an adversary learns nothing more than the specified leakage. The idea that it may be acceptable for a queryable encryption to leak some information to gain efficiency was seen previously in the case of structured encryption [CK10], and to some extent in the case of searchable encryption [CGKO06].

This is similar in spirit to previous definitions for structured encryption and SSE. However, we have to generalize in two ways. First, since previous solutions focused on document retrieval, all previous definitions require that the adversary to learn the list of documents to be returned; here there are no documents, and all that is required is that the client learn the result of the query. The second difference is that our query protocol will require a few rounds of communication. In particular for malicious adversaries this must be treated carefully - while malicious activity in the one round SSE setting may result in the client getting incorrect results, a malicious attacker in an interactive protocol could potentially also try to learn extra information.

We define correctness and security within two adversarial models: honest-but-curious and malicious. In the honest-but-curious model, the adversary executes the protocol honestly but tries to learn information about the message and the queries along the way. In the malicious model, the adversary tries to learn information, possibly by not following the protocol honestly.

Pattern matching encryption. Next, we focus on constructing a pattern matching encryption scheme, that is, a queryable encryption scheme that supports pattern matching queries – given a string s and a pattern string p, return all occurrences of p as a substring of s. For example, in the genomic data application researchers may wish to query the database to determine whether a particular cancer marker sequence appears in any of the data, to count whether a certain probe sequence is rare enough to be useful, or to .....

For efficiency, our goal is space and computation complexity comparable to that of evaluating pattern matching queries in the unencrypted setting. This means general techniques such as fully homomorphic encryption [Gen09, BV11, BGV12, GHS12] and functional encryption [BSW11, KSW08, SSW09] will not be practical. By focusing on the specific functionality of pattern matching queries, we are able to achieve a scheme with much better efficiency.

To construct a pattern matching encryption scheme, we use suffix trees, a data structure used to efficiently perform pattern matching 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 pattern matching on encrypted data, without revealing significant information about the string or the queries.

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

1.1 Related Work

Searchable encryption and structured encryption. We draw on related work on symmetric searchable encryption (SSE) [CGKO06] and its generalization to structured encryption [CK10]. 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 will be able to 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.[1] 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.

Recently there have been many advances in SSE. [CJJ+13] propose an efficient construction for searches involving multiple keywords; [CJJ+14, SPS14, KP13, KPR12] look at allowing updates to the stored documents; and [KO12] propose a UC definition. [WRYU12] use a tree data structure to implement SSE, essentially storing the encrypted keywords in a tree rather than a hash table. (Note that they are forming a

[1] Note that this is true even if we use fully homomorphic encryption (e.g., [Gen09, BV11, BGV12, GHS12]) or functional encryption [BSW11, KSW08, SSW09].

tree on encrypted data, while we form a tree out of the plaintext and then encrypt the tree structure.) However, all of these works focus on the problem of retrieving documents based on keywords; there has been very little work that considers encrypting other, more complex types of data structures.

Predicate encryption and fully homomorphic encryption. Predicate encryption (a special case of functional encryption [BSW11]) allows the secret key owner to generate "tokens" for various predicates; a token for a predicate f can be evaluated on a ciphertext that encrypts of m to determine whether $f(m)$ is satisfied. State-of-the-art predicate encryption schemes (e.g., [KSW08, SSW09]) 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 pattern matching encryption scheme, where the patterns 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 [Gen09] and further developed in subsequent work, e.g., [BV11, BGV12, GHS12], allows one to evaluate any arbitrary circuit on encrypted data without being able to decrypt. FHE would solve the pattern matching 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 [Ost92], 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 process in making oblivious RAMs efficient; however, even the most efficient constructions to date (see, e.g., Stefanov et al. [SSS12]) 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 pattern matching. There have been several works on secure two-party or multiparty computation (e.g., [DPSZ12, NNOB12]) and specifically on secure pattern matching and other text processing in the two-party setting (see [BDM$^+$12, MNSS11, HL10, GHS10, KM10, Fri09, TPKC07]). 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 pattern 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 pattern matching protocols.

Memory delegation and integrity checking. We consider both honest-but-curious and 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., [CKLR11]) and memory checking (e.g., [DNRV09]), verifiable computation (e.g., [BGV11, GGP10]), integrity checking (e.g., [SvDJO12]), and encrypted computation on untrusted programs (e.g., [FvDD12]); 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 pattern matching 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.

2 Basic Definitions

Here we review some cryptographic primitives we will use. See Appendix B for more details, and for some notation and definitions of standard cryptographic primitives (e.g., PRFs, PRPs) we will use.

ϵ-Almost-Universal Hash Functions An ϵ-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 when the hash function is chosen randomly from $\mathcal{H}$ is at most ϵ.

Polynomial Hash The following hash function is ϵ-almost-universal for $\epsilon = (n-1)/2^\ell$: View a message x as a sequence $(x_1, \ldots, x_n)$ of ℓ-bit strings. For any k in the finite field GF(2^ℓ), the hash function $H_k(x)$ is defined as the evaluation of the polynomial p_x over GF(2^ℓ) defined by coefficients $x_1, \ldots, x_n$, at the point k. That is, $H_k(x) = p_x(k) = \Sigma_{i=1}^n x_i k^{i-1}$, where all operations are in GF(2^ℓ).

Levin's trick We will sometimes want to compute a PRF on a long input. In this case, it can be more efficient to first hash the input down to a short string, and then apply the PRF to the hash output. If the hash function is ϵ-almost-universal for some negligible ϵ, then the resulting construction is still a PRF. This observation is due to Levin [Lev87] and is known sometimes as Levin's trick.

Symmetric-Key Encryption We will use symmetric-key encryption schemes that are *CPA-secure*, *which-key concealing*, and in some cases *authenticated*. The which-key concealing property was introduced by Abadi and Rogaway [AR02] and (under the name "key hiding") by Fischlin [Fis99], and says that an adversary cannot tell whether ciphertexts are encrypted under the same key or different keys. The notions of ciphertext integrity and *authenticated encryption* (defined below) were introduced by [BR00, KY01, BN00]. Ciphertext integrity says that an adversary given encryptions of messages of its choice cannot construct any new ciphertexts that decrypt successfully (i.e., decrypt to a value other than $\perp$). A symmetric encryption scheme is *authenticated* if it has CPA security and ciphertext integrity. For a simple construction from PRFs in the random oracle model, see Appendix B.

3 Queryable Encryption

We now present the main definitions for our construction.

Definition 3.1. A *queryable encryption scheme* for message space $\mathcal{M}$, query space $\mathcal{Q}$, answer space A, and query functionality $\mathcal{F} : \mathcal{M} \times \mathcal{Q} \to \mathsf{A}$, consists of the following probabilistic poly-time (PPT) algorithms.

$K \leftarrow$ *Gen*(1^λ): The key generation algorithm takes a security parameter 1^λ and generates a secret key K.

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

$A \leftarrow$ *IssueQuery*$(K, q) \leftrightarrow$ *AnswerQuery*(CT): The interactive algorithms *IssueQuery* and *AnswerQuery* compose a query protocol between a client and a server. The client takes as input the secret key K and a query q, and the server takes as input a ciphertext CT. At the end of the query protocol, the client outputs an answer $A \in \mathsf{A}$; the server has no output. A is a private output that is not seen by the server.

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

This correctness property ensures correct output if all algorithms are executed honestly. (This is the usual way in which correctness is defined for similar types of schemes, such as searchable encryption, structured encryption, and functional encryption.) However, it does not say anything about the client's output if the server does not honestly execute *AnswerQuery*.

Correctness against malicious adversaries. We also consider a stronger property, correctness against malicious adversaries, which says that the client's output will be correct if all algorithms are executed honestly, but the client will output $\perp$ if the server does not execute *AnswerQuery* honestly. Thus, the server may misbehave, but it cannot cause the client to unknowingly produce incorrect output.

More formally, correctness against malicious adversaries requires the following. For all PPT algorithms $\mathcal{A}$, for all $\lambda \in \mathbb{N}$, $q \in \mathcal{Q}$, $M \in \mathcal{M}$, let $K \leftarrow \textit{Gen}(1^\lambda)$, $CT \leftarrow \textit{Enc}(K, M)$, and $A \leftarrow \textit{IssueQuery}(K, q) \leftrightarrow \mathcal{A}(CT)$. If $\mathcal{A}$ honestly executes *AnswerQuery*, then $\Pr[A = \mathcal{F}(M, q)] = 1 - \text{negl}(\lambda)$. If $\mathcal{A}$ deviates from *AnswerQuery* in its input-output behavior, then $\Pr[A = \perp] = 1 - \text{negl}(\lambda)$.

Discussion. Note that, although the above is called a queryable "encryption scheme", it does not include an explicit decryption algorithm, as the client might not ever intend to retrieve the entire original message. However, we could easily augment the functionality F with a query 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. *AnswerQuery* could return the entire ciphertext, and *IssueQuery* could decrypt the ciphertext to get M and compute $\mathcal{F}(M, q)$ directly.

Our queryable encryption definition generalizes previous definitions of searchable encryption [CGKO06] and structured encryption [CK10], in the following ways.

Queryable encryption allows any general functionality 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 difference between queryable encryption and functional encryption is in the security requirements, which we will describe in the next section.

Also, queryable encryption allows the query protocol to be interactive. In searchable encryption, structured encryption, and functional encryption, the query protocol consists of two algorithms $TK \leftarrow \textit{Token}(K, q)$ and $A \leftarrow \textit{Query}(TK, CT)$. 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 has having a one-round interactive query protocol. Our definition allows for arbitrary interactive protocols, which may allow for better efficiency or privacy.

We do not need 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. Since the server does not see the final answer, we are able to achieve stronger privacy guarantees.

3.1 Honest-but-Curious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 Security

We first consider security in an honest-but-curious adversarial model. In this model, we assume that the server is honest (it executes the algorithms honestly), but curious (it can use any information it sees in the honest execution to learn what it can about the message and queries).

Ideally, we would like to guarantee that ciphertexts and query protocols reveal nothing about the message or the queries. However, such a strict requirement often makes it very difficult to achieve an efficient scheme. Therefore, we relax the security requirement somewhat so that some information may be revealed (leaked) to the server. The security definition will be parameterized by two "leakage" functions $\mathcal{L}_1, \mathcal{L}_2$. $\mathcal{L}_1(M)$ denotes the information about the message that is leaked by the ciphertext. For any j, $\mathcal{L}_2(M, q_1, \ldots, q_j)$ denotes the information about the message and all queries made so far that is leaked by the jth query.

We would like 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 considers a real experiment and an ideal experiment, and requires that the view of any adaptive adversary in the real experiment be simulatable given only the information specified by $\mathcal{L}_1$ and $\mathcal{L}_2$.

Following [CK10], we call the definition $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security, where the name "CQA2" comes from "chosen query attack", somewhat analogous to CCA2 (chosen ciphertext attack) for symmetric encryption, where an adversary can make adaptive decryption queries after receiving the challenge ciphertext.

Definition 3.2 (Honest-but-Curious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 Security)**.** Let $\mathcal{E} = (\mathit{Gen}, \mathit{Enc}, \mathit{Query})$ be a queryable encryption scheme for message space $\mathcal{M}$, query space $\mathcal{Q}$, answer space A, and query functionality $F : \mathcal{M} \times \mathcal{Q} \to \mathsf{A}$. 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 first runs $\mathit{Gen}(1^\lambda)$ to generate secret key K. The adversary $\mathcal{A}$ outputs a message M. The challenger runs $\mathit{Enc}(K, M)$ to generate a ciphertext CT, and sends CT to $\mathcal{A}$. The adversary adaptively chooses a polynomial number of queries, $q_1, \ldots, q_t$. For each query q_i, the challenger sends the adversary the view v_i of an honest server in the interactive protocol $\mathit{IssueQuery}(K, q_i) \leftrightarrow \mathit{AnswerQuery}(CT)$. Finally, $\mathcal{A}$ outputs a bit b, and b is output by the experiment.

Ideal$_{\mathcal{E},\mathcal{A},\mathcal{S}}(\lambda)$**:** First, $\mathcal{A}$ outputs a message M. The simulator $\mathcal{S}$ is given $\mathcal{L}_1(M)$ (not M itself), and outputs a value CT. The adversary adaptively chooses a polynomial number of queries, $q_1, \ldots, q_t$. For each query q_i, the simulator is given $\mathcal{L}_2(M, q_1, \ldots, q_i)$ (not q_i itself), and it outputs a simulated view v_i. Finally, $\mathcal{A}$ outputs a bit b, and b is output by the experiment.

We say that $\mathcal{E}$ is $(\mathcal{L}_1, \mathcal{L}_2)$-*CQA2 secure against honest-but-curious adversaries* if, for all PPT adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $|\Pr[\mathbf{Real}_{\mathcal{E},\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{E},\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mathrm{negl}(\lambda)$.

Discussion. The above definition is based heavily on the definition for structured encryption [CK10] and generalizes it to interactive query protocols. It is also loosely related to simulation-based definitions for functional encryption [BSW11], with one important difference: In our definition, we only consider a single ciphertext; security is not guaranteed to hold if multiple ciphertexts are encrypted under the same key. Note that security for only one ciphertext only makes sense in the symmetric-key setting, since in the public-key setting one can encrypt any number of messages with the public key. In our application, it will be reasonable to expect that each instantiation of the scheme will be used to encrypt only one message.

Although in some applications, it may be interesting and sufficient to model the server as an honest-but-curious adversary, often we will be interested in a stronger adversarial model. That is, we would like to ensure privacy against even a malicious adversary – one that does not execute its algorithms honestly. In the next section, we present a definition of security against malicious adversaries.

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

The definition of $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security against malicious adversaries is similar to the one for honest-but-curious adversaries, except for the following two differences. First, for each query q_i, instead of just receiving the view of the server, the adversary will interact with either an honest challenger running $\mathit{IssueQuery}(K, q_i)$ in the real game, or with the simulator given $\mathcal{L}_2(M, q_1, \ldots, q_i)$ in the ideal game.

Second, at the end of the protocol for q_i, the adversary outputs the description of a function g_i of its choice. In the real game, the adversary receives $g_i(A_1, \ldots, A_i)$, where A_j is the private answer output by the client for query q_j. In the ideal game, the adversary receives $g_i(A'_1, \ldots, A'_i)$, where $A'_j = \perp$ if the client output $\perp$ in the query protocol for q_j; otherwise, $A'_j = F(M, q_j)$.

This last step of the game is necessary in the malicious case because the adversary may learn extra information based on the client's responses to the incorrectly formed messages from the adversary. The client's private output, although not sent to the server in the actual query protocol, can be thought of as a response to the last message sent by the adversary. We want to capture the notion that, even if the adversary were able to learn some function g_i of the client's private outputs $A_1, \ldots, A_i$, it would not learn more than $g_i(F(M, q_1), \ldots, F(M, q_i))$. For any $A_j = \perp$, in the evaluation of g_i, $F(M, q_j)$ is replaced by $\perp$.

Definition 3.3 (Malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security)**.** Let $\mathcal{E} = (\mathit{Gen}, \mathit{Enc}, \mathit{Query})$ be a queryable encryption scheme for message space $\mathcal{M}$, query space $\mathcal{Q}$, answer space A, and query functionality $F : \mathcal{M} \times \mathcal{Q} \to \mathsf{A}$. 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 $\mathit{Gen}(1^\lambda)$ to generate a secret key K. The adversary $\mathcal{A}$ outputs a message M. The challenger runs $\mathit{Enc}(K, M)$ to generate a ciphertext CT, and sends CT to $\mathcal{A}$. The adversary adaptively makes a polynomial number of queries $q_1, \ldots, q_t$. For each query q_i, first $\mathcal{A}$ interacts with the challenger, which runs $\mathit{IssueQuery}(K, q_i)$. Let A_i be the challenger's private output from the protocol for q_i. Then $\mathcal{A}$ outputs a description of a function g_i, and it receives $h_i \leftarrow g_i(A_1, \ldots, A_i)$. 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 $\mathcal{L}_1(M)$ (not M itself), and outputs a value CT. The adversary adaptively makes a polynomial number of queries $q_1, \ldots, q_t$. For each query q_i, first the simulator is given $\mathcal{L}_2(M, q_1, \ldots, q_i)$ (not q_i itself), and $\mathcal{A}$ interacts with the simulator. Then $\mathcal{A}$ outputs a description of a function g_i, and it receives $h_i \leftarrow g_i(A'_1, \ldots, A'_i)$, where $A'_i = \perp$ if the simulator output $\perp$ in the query protocol for q_i; otherwise, $A'_i = \mathcal{F}(M, q_i)$. 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

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

3.3 Pattern Matching Encryption

Definition 3.4 (Pattern matching encryption)**.** A *pattern matching encryption scheme* for an alphabet Σ is a queryable encryption scheme for:

- message space $\mathcal{M} = \Sigma^*$,
- query space $\mathcal{Q} = \Sigma^*$,
- answer space $\mathsf{A} = \mathcal{P}(\mathbb{N})$, 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, $\mathcal{F}(s, p) = \{i | s[i..i + m - 1] = p\}$, where $m = |p|$.

4 Pattern Matching Encryption Construction

Our goal is to construct a pattern matching 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.

Suffix Trees We first look to pattern matching algorithms for unencrypted data. There are several known pattern matching algorithms [KJP77, BM77, KR87, AC75], varying in their preprocessing efficiency and query efficiency. Most of these 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 pattern p. Pattern matching using suffix trees, however, has preprocessing time $O(n)$ and query time $O(m)$. This is ideal for our applications, where the client stores one string s encrypted on the server, and performs queries for many pattern strings p. Therefore, we will focus on pattern matching using suffix trees as the basis for our scheme.

A *suffix tree* for a string s of length n is defined as a tree such that the paths from the root to the leaves are in one-to-one correspondence with the n suffixes of s, edges spell non-empty strings, each internal node has at least two children, and any two edges edges coming out of a node start with different characters. For a suffix tree for a string s to exist, s must be prefix-free; if it is not, we can first append a special symbol \$ to make s prefix-free. Figure 1 in Appendix A shows a suffix tree for an example string, "cocoon".

Pattern matching using suffix trees uses the following important observation: a pattern p is a substring of s if and only if it is a prefix of some suffix of s. Thus, to search s for a pattern p, search for a path from the root of which p is a prefix.

For a string of length n, a suffix tree can be constructed in $O(n)$ time [Ukk95, Far97]. It can be easily shown that a suffix tree has at most $2n$ nodes. However, if for each node we were to store the entire string spelled on the edge to that node, the total storage would be $O(n^2)$ in the worst case. (To see this, consider the suffix tree for the string $s_1 \dots s_n$, where each s_i is unique. The suffix tree would contain a distinct edge for each of the n suffixes $s_1 \dots s_n, s_2 \dots s_n, \dots s_n$; these suffixes have a total length $O(n^2)$.) To represent a suffix tree in $O(n)$ space, for each node u other than the root, one stores the start and end indices into s of the first occurrence of the substring on the edge to u. One also stores the string s. Using this representation, a suffix tree takes $O(n)$ storage and can be used to search for any pattern p of length m in $O(m)$ time, and to return the indices of all occurrences of p in $O(m + k)$ time, where k is the number of occurrences.

A few observations about suffix trees will be useful in our construction: For any node u, let $\rho(u)$ be the string spelled out on the path from the root to u. The string $\rho(u)$ uniquely identifies a node u in a suffix tree, i.e., no two distinct nodes u and u' have $\rho(u) = \rho(u')$. Let us also define $\hat{\rho}(u)$ to be the string spelled out on the path from the root to the parent of u, followed by the first character on the edge to u. Since no two edges coming out of a node start with the same character, $\hat{\rho}(u)$ also uniquely identifies u. Furthermore, the set of indices in s of occurrences of $\rho(u)$ is exactly the same as the set of indices of occurrences of $\hat{\rho}(u)$.

For any given string s, a suffix tree is generally not unique (the children of each node may be ordered in any way). For the remainder of the paper, we will assume that when a suffix tree is constructed, the children of every node are "ordered" lexicographically according to some canonical order of the alphabet. Thus, for a given string s, we talk about *the* unique suffix tree for s; we can also talk about the ith child of a node in a well-defined way. In the example in Figure 1, the suffix tree for "cocoon" is constructed with respect to the ordering (c, o, n). In Figure 1, u_5 and u_6 are the first and second children, respectively, of u_2.

4.1 Notation

Before we describe our pattern matching encryption scheme, we introduce some helpful 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*$_s$
ϵ:	the empty string
par(u):	the parent node of u. If u is the root, par(u) is undefined.
child(u, i):	the ith child node of u. If u has fewer than i children, child(u, i) is undefined.
deg(u):	the out-degree (number of children) of u
$\rho(u)$:	the string spelled on the path from the root to u. $\rho(u) = \epsilon$ if u is the root.
$\hat{\rho}(u)$:	For any non-root node u, $\hat{\rho}(u) = \rho(\text{par}(u)) \| u_1$, where u_1 is the first character on the edge from par(u) to u. If u is the root, $\hat{\rho}(u) = \epsilon$.
$leaf_i$:	the ith leaf node in *Tree*$_s$, where the leaves are numbered 1 to n, left to right
len_u:	the length of the string $\hat{\rho}(u)$

ind_u:	the index in s of the first occurrence of $\rho(u)$ (equivalently, of $\hat{\rho}(u)$) as a substring. That is, ind_u is the smallest i such that $\hat{\rho}(u) = s[i..i + len_u - 1]$. If $\rho(u) = \epsilon$, ind_u is defined to be 0.
$lpos_u$:	the position (between 1 and n) of the leftmost descendant of u. That is, $leaf_{lpos_u}$ is the leftmost leaf in the subtree rooted at u.
num_u:	the number of occurrences in s of $\rho(u)$ (equivalently, of $\hat{\rho}(u)$) as a substring. If $\rho(u) = \epsilon$, num_u is defined to be 0. Note that 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 $u_2 = \text{par}(u_3), u_3 = \text{child}(u_2, 1), \text{deg}(u_2) = 2, \rho(u_3) = \text{"cocoon"}, \hat{\rho}(u_3) = \text{"coc"}, leaf_5 = u_8, ind_{u_2} = 1, lpos_{u_2} = 1, num_{u_2} = 2$.

4.2 Intuition

Our construction will make use of a dictionary, which is a data structure that stores key-value pairs (k, V), such that for any key k the corresponding value V can be retrieved efficiently (in constant time).

We will use a symmetric encryption scheme $\mathcal{E}_{\text{SKE}}$, a PRF F, and a PRP P. The key generation algorithm will generate three keys K_D, K_C, K_L for $\mathcal{E}_{\text{SKE}}$, and four keys K_1, K_2, K_3, K_4. (We will explain how the keys are used as we develop the intuition for the construction.)

First attempt. We first focus on constructing a queryable encryption scheme for a simpler functionality $\mathcal{F}'$, where $\mathcal{F}'(s, p)$ computes 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 for now.

As a first attempt, let $\mathcal{E}_{\text{SKE}}$ be a CPA-secure symmetric encryption scheme, and encrypt a string $s = s_1 \ldots s_n$ in the following way. First, construct the suffix tree $Tree_s$ for s. Then construct a dictionary D, where for each node u in $Tree_s$, there is an entry with search key $F_{K_1}(\rho(u))$ and value $\mathcal{E}_{\text{SKE}}.Enc(K_D, ind_u)$, and let the ciphertext consist of the dictionary D. Then, in the query protocol for a query p, the client sends $F_{K_1}(p)$. The server then checks whether D contains an entry with search key $F_{K_1}(p)$. If so, it returns $D(F_{K_1}(p))$, which the client decrypts using K_D to get the index of the first occurrence in s of p.

For example, for our example string "cocoon", the ciphertext in this first attempt would consist of the dictionary shown in Figure 2 in Appendix A.

The obvious problem with this approach is that it only works for patterns that are substrings of s that end exactly at a node; it does not work for finding substrings of s that end partway down an edge.

Returning a possible match. To address this problem, we observe that we can uniquely identify each node u by $\hat{\rho}(u)$ instead of $\rho(u)$. Furthermore, if u is the last node (farthest from the root) for which any prefix of p equals $\hat{\rho}(u)$, then either p is not a substring of s, or p ends partway down the path to u and the indices in s of the occurrences of $\hat{\rho}(u)$ are the same as the indices in s of the occurrences of p.

In the dictionary D, we will now use $\hat{\rho}(u)$ instead of $\rho(u)$ as the search key for a node u. We will say that a prefix $p[1..i]$ is a *matching prefix* if $p[1..i] = \hat{\rho}(u)$ for some u, i.e., there is a dictionary entry corresponding to $p[1..i]$; otherwise, $p[1..i]$ is a *non-matching prefix*.

The ciphertext will also include an array C of character-wise encryptions of s, with $C[i] = \mathcal{E}_{\text{SKE}}.Enc(K_C, s_i)$. In the query protocol, the client will send $T_1, \ldots, T_m$, 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 $\mathcal{E}_{\text{SKE}}.Enc(K_D, ind)$ stored in $D(T_j)$. The client will then decrypt it to get ind, and request the server to send

$C[ind], \ldots, C[ind+m-1]$. The client can decrypt the result to check whether the decrypted string is equal to the pattern p and thus, whether p is a substring of s.

Returning all occurrences. We would like to return the indices of all occurrences of p in s, not just the first occurrence or a constant number of occurrences. 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.

We will use the observation that the occurrences of the prefix associated with a node are exactly the occurrences of the strings associated with the leaves in the subtree rooted at that node. Each leaf corresponds to exactly one suffix. So, 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 string on the path to the ith leaf. That is, $L[i] = \mathcal{E}_{\text{SKE}}.Enc(K_L, ind_{leaf_i})$. In the encrypted tuple in the dictionary entry for a node u we also store $lpos_u$, the leaf position of the leftmost leaf in the subtree rooted at u, and num_u, the number of occurrences of $\hat{\rho}(u)$. That is, the value in the dictionary entry for a node u is now $\mathcal{E}_{\text{SKE}}.Enc(K_D, (ind_u, lpos_u, num_u))$ instead of $\mathcal{E}_{\text{SKE}}.Enc(K_D, ind_u)$. The server will return $\mathcal{E}_{\text{SKE}}.Enc(K_D, (ind_u, lpos_u, num_u))$ for the last node u matched by a prefix of p. The client then decrypts to get $ind_u, lpos_u, num_u$, asks for $C[ind], \ldots, C[ind+m-1]$, decrypts to determine whether p is a substring of s, and if so, asks for $L[lpos_u], \ldots, L[lpos_u+num-1]$ 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 pattern matching query functionality, against an honest-but-curious adversary. However, it leaks a lot of unnecessary information; we add a number of improvements to reduce the information that is leaked.

For any two queries p and p' whose first j prefixes are the same, the values $T_1, \ldots, T_j$ in the query protocol will be the same. Therefore, the server will learn that $p[1..j] = p[1..j']$, even though $p[1..j]$ may not be contained in s at all. Memory accesses will reveal to the server when two queries share a prefix $p[1..j]$ that is a matching prefix (i.e., 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 $\mathcal{E}_{\text{SKE}}$ 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 contain values $f_{2,i}$ for its children nodes, where $f_{2,i} = F_{K_2}(\hat{\rho}(\text{child}(u,i))$. Note that $f_{2,i}$ is the key used to encrypt T_j for any pattern p whose prefix $p[1..j]$ is equal to $\hat{\rho}(\text{child}(u,i))$. 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 pattern.

Hiding node degrees, lexicographic order of children, number of nodes in suffix tree. Since the maximum degree of any node is the size d of the alphabet, we hide the degree of each node by creating dummy random $f_{2,i}$ values so that there are d in total. To hide the lexicographic 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. For each dummy entry, the search key is a random value f_1, and the value is $(f_{2,1}, \ldots, f_{2,d}, W)$, where $f_{2,1}, \ldots, f_{2,d}$ are random and W is an encryption of 0.

Hiding string indices and leaf positions. In order to hide the actual values of the string indices $ind, \ldots, ind+m-1$ and the leaf positions $lpos, \ldots, lpos+num-1$, we make use of a pseudorandom permutation family P of permutations $[n] \to [n]$. Instead of sending $(ind, \ldots, ind+m-1)$, the client applies the permutation P_{K_3} to $ind, \ldots, ind+m-1$ and outputs the resulting values in a randomly permuted order as

$(x_1, \ldots, x_m)$. Similarly, instead of sending $(lpos, \ldots, lpos + num - 1)$, the client applies the permutation P_{K_4} to $lpos, \ldots, lpos + num - 1$ and outputs the resulting values in a randomly permuted order as $(y_1, \ldots, y_{num})$. 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 correctness against an honest-but-curious adversary, but not a malicious adversary, since the client does not perform any checks to ensure that the server is sending correct messages. The scheme also would not satisfy security against a malicious adversary for reasonable leakage functions, since an adversary could potentially gain information by sending malformed or incorrect ciphertexts during the query protocol.

To handle a malicious adversary, we will require $\mathcal{E}_{\text{SKE}}$ to be an authenticated encryption scheme. Thus, an adversary will not be able to obtain the decryption of any ciphertext that is not part of the dictionary D or the arrays C or L. Also, we add auxiliary information to the messages encrypted, to allow the client to check that any ciphertext returned by the adversary is the one expected by the honest algorithm. The client will be able to detect whether the server returned a ciphertext that is in D but not the correct one, for example.

Specifically, 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. Similarly, we will encrypt (ind_{leaf_i}, i) instead of just ind_{leaf_i}. For the dictionary entries, in addition to ind_u,num_u, and $lpos_u$, we will include $len_u, f_1(u), f_{2,1}(u), \ldots, f_{2,d}(u)$ in the tuple that is encrypted. The client can then check whether the W sent by the adversary corresponds to the longest matching prefix of p, by verifying that $F_{K_1}(p[1..len]) = f_1$, and that none of the $f_{2,1}, \ldots, f_{2,d}$ successfully decrypts any of the T_j for $j > len$.

4.3 Final construction

Let $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a PRF, and let $P : \{0,1\}^\lambda \times [n] \to [n]$ be a PRP. Let $\mathcal{E}_{\text{SKE}} = (\mathit{Gen}, \mathit{Enc}, \mathit{Dec})$ be an authenticated, which-key-concealing symmetric encryption scheme. Our pattern matching encryption scheme $\mathcal{E}_{\text{PM}}$ for an alphabet Σ with $|\Sigma| = d$ is as follows.

Gen(1^λ)**:** Choose random strings $K_D, K_C, K_L, K_1, K_2, K_3, K_4 \xleftarrow{R} \{0,1\}^\lambda$.[2] The secret key is

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

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

1. Construct a dictionary D as follows.
   For any node u, define $f_1(u) := F_{K_1}(\hat{\rho}(u))$ and $f_2(u) := F_{K_2}(\hat{\rho}(u))$.
   For each node u in Tree_s (including the root and leaves), proceed as follows:
   - Choose a random permutation $\pi_u : [d] \to [d]$.
   - For $i = 1, \ldots, d$, let $f_{2,i}(u) = f_2(\text{child}(u, \pi_u(i))$ if $1 \le \pi_u(i) \le \deg(u)$; otherwise let $f_{2,i}(u) \xleftarrow{R} \{0,1\}^\lambda$.
   - Let $X_u = (ind_u, lpos_u, num_u, len_u, f_1(u), f_{2,1}(u), \ldots, f_{2,d}(u))$, and let $W_u = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_D, X_u)$.
   - Store $V_u = (f_{2,1}(u), \ldots, f_{2,d}(u), W_u)$ with search key $\kappa_u = f_1(u)$ in D.

   Let N denote the number of nodes in Tree_s. Construct $2n - N$ dummy entries in D as follows. For each dummy entry, choose random strings $f_1, f_{2,1}, \ldots, f_{2,d} \xleftarrow{R} \{0,1\}^\lambda$, and store $(f_{2,1}, \ldots, f_{2,d}, \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_D, 0))$ with search key f_1 in D.

[2] We will assume for simplicity that $\mathcal{E}_{\text{SKE}}.\mathit{Gen}$ simply chooses a random key $k \xleftarrow{R} \{0,1\}^\lambda$, so throughout the construction we will use random values as $\mathcal{E}_{\text{SKE}}$ keys. To allow for general $\mathcal{E}_{\text{SKE}}.\mathit{Gen}$ algorithms, instead of using a random value r directly as a key, we could use a key generated by $\mathcal{E}_{\text{SKE}}.\mathit{Gen}$ with r providing $\mathcal{E}_{\text{SKE}}.\mathit{Gen}$'s random coins.

2. Construct an array C as follows: for $i = 1, \ldots, n$, set $C[P_{K_3}(i)] = \mathcal{E}_{\text{SKE}}.\textit{Enc}(K_C, (s_i, i))$.
3. Construct an array L as follows: For $i = 1, \ldots, n$, set $L[P_{K_4}(i)] = \mathcal{E}_{\text{SKE}}.\textit{Enc}(K_L, (ind_{leaf_i}, i))$.

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

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

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

1. The client computes, for $i = 1, \ldots, m$, $f_1^{(i)} = F_{K_1}(p_1 \ldots p_i)$, $f_2^{(i)} = F_{K_2}(p_1 \ldots p_i)$, and sets $T_i = \mathcal{E}_{\text{SKE}}.\textit{Enc}(f_2^{(i)}, f_1^{(i)})$. The client sends the server $(T_1, \ldots, T_m)$.
2. The server proceeds as follows, maintaining variables $f_1, f_{2,1}, \ldots, f_{2,d}, W$. Initialize $(f_{2,1}, \ldots, f_{2,d}, W)$ to equal $D(F_{K_1}(\epsilon))$, where ϵ denotes the empty string.

   For $i = 1, \ldots, m$:

   For $j = 1, \ldots, d$:

   Let $f_1 \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Dec}(f_{2,j}, T_i)$. If $f_1 \neq \perp$, update $(f_{2,1}, \ldots, f_{2,d}, W)$ 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 $X \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Dec}(K_D, W)$. If $X = \perp$, output $\perp$ and end the protocol. Otherwise, parse X as $(ind, lpos, num, len, f_1, f_{2,1}, \ldots, f_{2,d})$. Check whether $F_{K_1}(p[1..len]) = f_1$. If not, output $\perp$ and end the protocol. Otherwise, check whether $\mathcal{E}_{\text{SKE}}.\textit{Dec}(f_{2,i}, T_j) = \perp$ for any $j \in \{len + 1, \ldots, m\}$ and $i \in \{1, \ldots, d\}$. If so, output $\perp$ and end the protocol. If $ind = 0$, output $\emptyset$. Otherwise, choose a random permutation $\pi_1 : [m] \to [m]$. For $i = 1, \ldots, m$, let $x_{\pi_1(i)} = P_{K_3}(ind + i - 1)$. The client sends $(x_1, \ldots, x_m)$ to the server.
4. The server sets $C_i = C[x_i]$ for $i = 1, \ldots, m$ and sends $(C_1, \ldots, C_m)$ to the client.
5. For $i = 1, \ldots, m$, the client runs $Y \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Dec}(K_C, C_{\pi_1(i)})$. If $Y = \perp$, output $\perp$ and end the protocol. Otherwise, let the result be (p'_i, j). If $j \neq ind + i - 1$, output $\perp$. Otherwise, if $p'_1 \ldots p'_m \neq p$, then the client outputs $\emptyset$ as its answer and ends the protocol. Otherwise, the client chooses a random permutation $\pi_2 : [num] \to [num]$. For $i = 1, \ldots, num$, let $y_{\pi_2(i)} = P_{K_4}(lpos + i - 1)$. The client sends $(y_1, \ldots, y_{num})$ to the server.
6. The server sets $L_i = L[y_i]$ for $i = 1, \ldots, num$, and sends $(L_1, \ldots, L_{num})$ to the client.
7. For $i = 1, \ldots, num$, the client runs $\mathcal{E}_{\text{SKE}}.\textit{Dec}(K_L, L_{\pi_2(i)})$. If the result is $\perp$, the client outputs $\perp$ as its answer. Otherwise, let the result be (a_i, j). If $j \neq lpos + i - 1$, output $\perp$. Otherwise, output the answer $A = \{a_1, \ldots, a_{num}\}$.

4.4 Efficiency

In analyzing the efficiency of our construction, we will assume data is stored in computer words that hold $\log n$ bits; therefore, we will treat values of size $O(\log n)$ as constant size.

We assume encryption and decryption using $\mathcal{E}_{\text{SKE}}$ 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 $O(n^2)$. This is because even though there are only at most $2n$ nodes, the sum of the lengths of the strings $\hat{\rho}(u)$ associated with the nodes u can be $O(n^2)$. Similarly, computing the PRFs used for $T_1, \ldots, T_m$ would take $O(m^2)$ time. It turns out that we can take advantage 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).

To compute the PRF of a string x, we will first hash x to λ bits using the polynomial hash, and then apply the PRF (which takes time $O(\lambda)$ on the hashed input). To efficiently compute the hashes of all of the strings $\hat{\rho}(u)$, we use a trick that is used in the Rabin-Karp rolling hash (see Cormen et al. [CLRS09],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 $H_k(x_1 \ldots x_n)$ from $H_k(x_2 \ldots x_n)$, and also of $H_k(x_1 \ldots x_n)$ from $H_k(x_1 \ldots x_{n-1})$. To see this, notice that $H_k(x_1 \ldots, x_n) = x_1 + k \cdot H_k(x_2 \ldots x_n)$, and $H_k(x_1 \ldots x_n) = H_k(x_1 \ldots x_{n-1}) + x_n k^{n-1}$.

Using this trick, for any string x of length ℓ, we can compute the hashes $H_k(x[1..i])$ for all $i = 1, \ldots, m$ in total time $O(\lambda m)$. Thus, the $T_1, \ldots, T_m$ can be computed in time $O(\lambda m)$.

To compute the hashes of $\hat{\rho}(u)$ 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 $\rho(u) = a||B$, where a is a single character, has a pointer called a suffix link pointing to the node u' whose associated string $\rho(u')$ 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. To see this, notice that each internal node has an outgoing suffix link, and each node's suffix link points to a node with a shorter associated string of one fewer character, so there can be no cycles.

Since $\hat{\rho}(u) = \rho(\text{par}(u))||u_1$, we can first compute the hashes of $\rho(u)$ for all non-leaf nodes u, and then compute $\hat{\rho}(u)$ for all nodes u in constant time from $\rho(\text{par}(u))$. To compute $\rho(u)$ for all nodes u, we traverse the tree formed by the suffix links, starting at the root, and compute the hash of $\rho(u)$ for each u using $\rho(u')$, where u' is u's parent in the suffix link tree. Each of these computations takes constant time, since $\rho(u)$ is the same as $\rho(u')$ but with one character appended to the front. Therefore, computing the hashes of $\rho(u)$ for all non-leaf nodes u (and thus, computing the hashes of $\hat{\rho}(u)$ for all nodes u) takes total time $O(n)$.

Encryption efficiency. Using the efficient batch implementation of PRFs suggested 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 $O(\lambda n)$. 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 $T_1, \ldots, T_m$. Using the efficient batch PRF implementation above, computing the $f_1^{(i)}$ and $f_2^{(i)}$ for $i = 1, \ldots, m$ takes total time $O(m)$, and computing each $\mathcal{E}_{\text{SKE}}.\textit{Enc}(f_2^{(i)}, f_1^{(i)})$ takes $O(\lambda)$ time, so the total time to compute $T_1, \ldots, T_m$ is $O(\lambda m)$.

To find W, the server performs at most md decryptions and dictionary lookups, which takes total time $O(\lambda m)$. The client then computes $x_1, \ldots, x_m$ and the server retrieves $C[x_1], \ldots, C[x_m]$, in time $O(m)$. If the answer is not $\emptyset$, the client then computes $y_1, \ldots, y_{num}$ and the server retrieves $L[y_1], \ldots, L[y_{num}]$ in time $O(num)$, in time $O(num)$. Thus, both the client and the server take computation time $O(\lambda m + num)$ 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 by outputting $\perp$.) The query protocol takes three rounds of communication, and the total size of the messages exchanged is $O(\lambda m + num)$.

4.5 Security

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*$_s$ for s if $\hat{\rho}(u)$ is a prefix of p. For any j let p_j denote the jth query, and let $m_j = |p_j|$. Let n_j denote the number of nodes visited by the query for p_j in s, let $u_{j,i}$ denote the ith such node, and let $len_{j,i} = |\hat{\rho}(u_{j,i})|$. Let num_j denote the number of occurrences of p_j as a substring of s. Let ind_j denote the index ind in the ciphertext W returned by *AnswerQuery* for p_j. Note that ind_j is the index in s of the longest matching prefix of p_j, which is also the index in s of the longest prefix of p_j that is a substring of s. Let $lpos_j$ denote the leaf index $lpos$ in the ciphertext W returned by *AnswerQuery* for p_j. If p_j is a substring of s, $lpos_j$ is equal to the position (between 1 and n, from left to right) of the leftmost leaf ℓ for which p_j is a prefix of $\hat{\rho}(\ell)$.

Briefly: The query prefix pattern QP for a query p_j tells which of the previous queries $p_1, \ldots, p_{j-1}$ visited each of the nodes visited by p_j. The index intersection pattern IP for a query p_j essentially tells when any of the indices $ind_j, \ldots, ind_j + m_j - 1$ are equal to or overlap with any of the indices $ind_i, \ldots, ind_i + m_i - 1$ for any previous queries p_i. The leaf intersection pattern LP for a query p_j essentially tells when any of the leaf positions $lpos_j, \ldots, lpos_j + num_j - 1$ are equal to or overlap with any of the leaf positions $lpos_i, \ldots, lpos_i + num_i - 1$ for any previous queries p_i.

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

$$(m_j = |p_j|,\ \{len_{j,i}\}_{i=1}^{n_j},\ \text{QP}(s, p_1, \ldots, p_j),\ \text{IP}(s, p_1, \ldots, p_j),\ \text{LP}(s, p_1, \ldots, p_j)) \ .$$

For a more formal definition and an example of the leakage, see Appendix D.1.

Security Theorems See Appendix D.2 and C for proofs of the following theorems.

Theorem 4.1. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be as defined above. If F is a PRF, P is a PRP, and $\mathcal{E}_{\text{SKE}}$ is a CPA-secure, key-private symmetric-key encryption scheme, then the pattern matching encryption scheme $\mathcal{E}_{\text{PM}}$ satisfies malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security.*

Theorem 4.2. *If $\mathcal{E}_{\text{SKE}}$ is an authenticated encryption then $\mathcal{E}_{\text{PM}}$ is correct against malicious adversaries.*

5 Conclusion

We presented a definition of queryable encryption schemes and defined security against both honest-but-curious and 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 pattern matching scheme – a queryable encryption scheme that supports finding all occurrences of a pattern p in 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 λ is the security parameter, n is the length of the string, m is the length of the pattern, and k is the number of occurrences of the pattern. Querying requires only 3 rounds of communication.

While we have given a formal characterization of the leakage of our pattern matching 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? For some applications, the efficiency may be worth the leakage tradeoff, especially in applications where current practice does not use encryption at all.

References

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, June 1975.

[AR02] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[BDJR97] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS '97)*, pages 394–403. IEEE Press, 1997.

[BDM+12] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. In *SCN*, volume 7485 of *Lecture Notes in Computer Science*. Springer, 2012.

[BGV11] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO '11*, volume 6841 of *Lecture Notes in Computer Science*, pages 111–131. Springer, 2011.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*, 2012.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT '00*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[BR00] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT '00*, volume 1976 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2000.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography Conference (TCC '11)*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *52nd Annual IEEE Symposium on. Foundations of Computer Science (FOCS '11)*, 2011.

[CGKO06] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer, 2013.

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encyrption in very-large databases: Data structures and implementation. In *NDSS*, 2014.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *Advances in Cryptology – ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[CKLR11] Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO '11*, volume 6841 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2011.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[DNRV09] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography Conference (TCC '09)*, volume 5444 of *Lecture Notes in Computer Science*, pages 503–520, 2009.

[Dod] Yevgeniy Dodis. Lecture notes, introduction to cryptography. `http://www.cs.nyu.edu/courses/spring12/CSCI-GA.3210-001/lect/lecture10.pdf`.

[DPSZ12] Ivan Damgard, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Rei Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO '12*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS '97)*, pages 137–143. IEEE Press, 1997.

[Fis99] Marc Fischlin. Pseudorandom function tribe ensembles based on one-way permutations: Improvements and applications. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 432–445. Springer-Verlag, 1999.

[Fri09] Keith B. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. In *23rd Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec '09)*, pages 81–94, 2009.

[FvDD12] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *ACM Scalable Trusted Computing Workshop (STC)*, 2012.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC '09)*, pages 169–178. ACM Press, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.

[GHS10] Rosario Gennaro, Carmit Hazay, and Jeffrey S. Sorensen. Text search protocols with simulation based security. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography (PKC)*, pages 332–350, 2010.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Rei Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO '12*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[HL10] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *J. Cryptology*, 23(3):422–456, 2010.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KM10] Jonathan Katz and Lior Malka. Secure text processing with applications to private DNA matching. In *ACM Conference on Computer and Communications Security (CCS '10)*, pages 485–492, 2010.

[KO12] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 285–298. Springer, 2012.

[KP13] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography*, pages 258–274, 2013.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security*, pages 965–976, 2012.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[KSW08] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology - EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2008.

[KY01] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In Bruce Schneier, editor, *FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.

[Lev87] Leonid A. Levin. One-way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[MNSS11] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious DFA evaluation and applications. IACR Cryptology ePrint Archive, `http://eprint.iacr.org/2011/434`, 2011.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Rei Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO '12*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[Ost92] Rafail Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, MIT, 1992.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. 2014.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Song. Toward practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[SSW09] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *Theory of Cryptography Conference (TCC '09)*, pages 457–473. Springer, 2009.

[SvDJO12] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 229–238, 2012.

[TPKC07] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Utku Celik. Privacy preserving error resilient DNA searching through oblivious automata. In *ACM Conference on Computer and Communications Security (CCS '07)*, pages 519–528, 2007.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[WRYU12] Cong Wang, Kui Ren, Shucheng Yu, and Karthik Mahendra Raje Urs. Achieving usable and privacy-assured similarity search over outsourced cloud data. In *INFOCOM*, pages 451–459, 2012.

A Suffix Tree Figures

The following figures illustrate suffix trees and the "first attempt" scheme described in the intuition for the main construction.

B Notation and Primitives

B.1 Basic Notation

We write $x \xleftarrow{\mathrm{R}} X$ 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. We write $x||y$ to refer to the concatenation of strings x and y, and

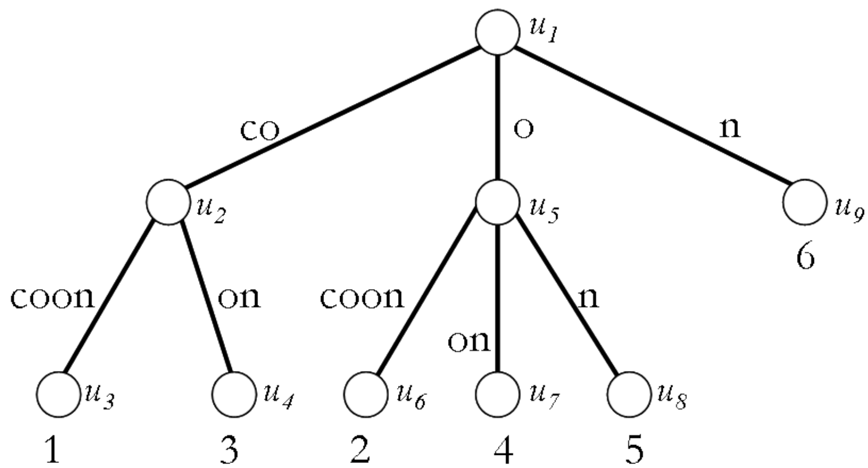

Figure 1: A suffix tree for the string $s =$"cocoon". We will use the node labels $u_1, \ldots, u_9$ later to explain how the pattern matching encryption scheme works.

node	key	value
u_1	$F_{K_1}(\epsilon)$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 0)$
u_2	$F_{K_1}(\text{"co"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 1)$
u_3	$F_{K_1}(\text{"cocoon"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 1)$
u_4	$F_{K_1}(\text{"coon"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 3)$
u_5	$F_{K_1}(\text{"o"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 2)$
u_6	$F_{K_1}(\text{"ocoon"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 2)$
u_7	$F_{K_1}(\text{"oon"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 4)$
u_8	$F_{K_1}(\text{"on"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 5)$
u_9	$F_{K_1}(\text{"n"})$	$\mathcal{E}_{\text{SKE}}.Enc(K_D, 6)$

Figure 2: The dictionary composing the ciphertext for the string "cocoon" in the "first attempt" scheme. Note that the node identifiers $u_1, \ldots, u_9$ are not a part of the dictionary; they are provided for the purpose of cross-referencing with the suffix tree in Figure 1.

$|x|$ to refer to the length of a string x. If $x = x_1 \dots x_n$ is a string of n characters, and a and b are integers, $1 \le a, b \le n$, then $x[a..b]$ denotes the substring $x_a x_{a+1} \dots x_b$. We sometimes use ϵ to denote the empty string. In other places ϵ will be used to denote a quantity that is negligible in the security parameter; the intended meaning of ϵ will be clear from context.

If T is a tuple of values with variable names $(a, b, \dots)$, then $T.a, T.b, \dots$ refer to the values in the tuple. If n is a positive integer, we use $[n]$ to denote the set $\{1, \dots, n\}$. If S is a set, $\mathcal{P}(S)$ is the corresponding power set, i.e., the set of all subsets of S.

We use λ to refer to the security parameter, and we assume all algorithms implicitly take λ as input. A function $\nu : \mathbb{N} \to \mathbb{N}$ is negligible in λ if for every positive polynomial $p(\cdot)$ there exists an integer $\lambda_p > 0$ such that for all $\lambda > \lambda_p$, $\nu(\lambda) < 1/p(\lambda)$. We let negl$(\lambda)$ denote an unspecified negligible function in λ.

Following standard GMR notation [GMR88], if $p(\cdot, \cdot, \dots)$ is a predicate, the notation $\Pr[a \leftarrow A; b \leftarrow B; \dots : p(a, b, \dots)]$ denotes the probability that $p(a, b, \dots)$ is true after $a \leftarrow A, b \leftarrow B, \dots$ are executed in order. We write $\mathcal{A}^{\mathcal{O}}$ to represent that algorithm $\mathcal{A}$ can make oracle queries to algorithm $\mathcal{O}$. We will assume that adversaries are stateful algorithms; that is, an adversary $\mathcal{A}$ maintains state across multiple invocations by implicitly taking its previous state as input and outputting its updated state.

If f is a function with domain D, and $S \subseteq D$, then $f[S]$ denotes the image of S under f. If $F : \mathcal{K} \times D \to R$ is a family of functions from D to R, where $\mathcal{K}$, D, and R are finite sets, we write F_K for the function defined by $F_K(x) = F(K, x)$.

B.2 Pseudorandom Functions and Permutations

A pseudorandom function family (PRF) is a family F of functions such that no probabilistic polynomial-time (PPT) adversary can distinguish a function chosen randomly from F from a uniformly random function, except with negligible advantage.

Definition B.1 (Pseudorandom Function Family)**.** Let D and R be finite sets, and let $F : \{0,1\}^\lambda \times D \to R$ be a family of functions. Let $\mathcal{R}$ denote the set of all possible functions $Z : D \to R$. F is a *pseudorandom function family (PRF)* if for all PPT adversaries A,

$$|\Pr[K \xleftarrow{R} \{0,1\}^\lambda : \mathcal{A}^{F_K}(1^\lambda) = 1] - \Pr[Z \xleftarrow{R} \mathcal{R} : \mathcal{A}^{Z}(1^\lambda) = 1]| \le \text{negl}(\lambda) \ .$$

Similarly, a pseudorandom permutation family (PRP) is a family of functions such that no PPT adversary can distinguish a function randomly chosen from F and a uniformly random permutation, except with negligible advantage.

Definition B.2 (Pseudorandom Permutation Family)**.** Let D be a finite set, and let $F : \{0,1\}^\lambda \times D \to D$ be a family of functions. Let $\mathcal{P}$ denote the set of all possible permutations (one-to-one, onto functions) $P : D \to D$. F is a *pseudorandom permutation family (PRP)* if for all PPT adversaries A,

$$|\Pr[K \xleftarrow{R} \{0,1\}^\lambda : \mathcal{A}^{F_K}(1^\lambda) = 1] - \Pr[P \xleftarrow{R} \mathcal{P} : \mathcal{A}^{P}(1^\lambda) = 1]| \le \text{negl}(\lambda) \ .$$

B.3 ϵ-Almost-Universal Hash Functions

An ϵ-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 when the hash function is chosen randomly from $\mathcal{H}$ is at most ϵ.

Definition B.3 (ϵ-Almost-Universal Hash Function)**.** Let U and B be finite sets, and let $H : \{0,1\}^\lambda \times U \to B$ be a family of hash functions. H is *ϵ-almost-universal* if for any $x, x' \in U$, $x \neq x'$,

$$\Pr[t \xleftarrow{\mathrm{R}} \{0,1\}^\lambda : H_t(x) = H_t(x')] \leq \epsilon \ .$$

Let us look at an example of a known ϵ-almost-universal hash construction, which we shall use later.

Example B.4. [Polynomial hash] We view a message x as a sequence $(x_1, \ldots, x_n)$ of ℓ-bit strings. For any k in the finite field GF(2^ℓ), the hash function $H_k(x)$ is defined as the evaluation of the polynomial p_x over GF(2^ℓ) defined by coefficients $x_1, \ldots, x_n$, at the point k. That is, $H_k(x) = p_x(k) = \Sigma_{i=1}^n x_i k^{i-1}$, where all operations are in GF(2^ℓ).

The hash function family defined above is ϵ-almost-universal, for $\epsilon = (n-1)/2^\ell$. To see this, suppose $H_k(x) = H_k(x')$ for some $x \neq x'$. Then $p_x(k) = p_{x'}(k)$. This means $p_{x-x'}(k) = \Sigma_{i=1}^n (x_i - x_i')k^{i-1} = 0$, where at least one of $(x_i - x_i')$ is not 0. Since $p_{x-x'}(\cdot)$ is a non-zero polynomial of degree at most $n-1$, it can have at most $n-1$ roots. The probability that a k chosen randomly from GF(2^ℓ) will be one of the at most $n-1$ roots is at most $(n-1)/2^\ell$.

B.4 PRF Composed with Almost Universal Hashing

When computing a PRF on a long input, it can be more efficient to first hash the input down to a short string, and then apply the PRF to the hash output. If the hash function is ϵ-almost-universal for some negligible ϵ, then the resulting construction is still a PRF. This observation is due to Levin [Lev87] and is known sometimes as Levin's trick.

The following theorem says that a PRF composed with an ϵ-almost-universal hash function, where ϵ is negligible, gives another PRF. A proof of this theorem has been given previously in [Dod]; we include a version of that proof here, for completeness.

Theorem B.5. *Let p be some polynomial. Let $F : \{0,1\}^\lambda \times B \to R$ be a PRF, and let $H : \{0,1\}^{p(\lambda)} \times U \to B$ be an ϵ-almost-universal hash function for some $\epsilon = \mathrm{negl}(\lambda)$. Then $F(H) : \{0,1\}^{\lambda + p(\lambda)} \times U \to R$, defined by $F_{K,t}(x) = F_K(H_t(x))$, is a PRF.*

Proof. Let $\mathcal{A}$ be an adversary attacking the PRF property of $F(H)$. We wish to show that $\mathcal{A}$'s advantage in distinguishing $F_K(H_t(\cdot))$ for random K, t from $Z(\cdot)$, where Z is a uniformly random function from U to R, is negl(λ). To do so, we first argue that $\mathcal{A}$'s advantage in distinguishing $F_K(H_t(\cdot))$ from $Y(H_t(\cdot))$, where Y is a uniformly random function from B to R, is negl(λ). We then argue that $\mathcal{A}$'s advantage in distinguishing $Y(H_t(\cdot))$ from $Z(\cdot)$ is negl(λ). Therefore, $\mathcal{A}$'s total advantage in distinguishing $F_K(H_t(\cdot))$ from $Z(\cdot)$ is negl(λ).

By the PRF property of F, we immediately have that $\mathcal{A}$'s advantage in distinguishing $F_K(H_t(\cdot))$ for a random K from $Y(H_t(\cdot))$ is at most negl(λ).

Next, to see that $\mathcal{A}$ cannot distinguish $Y(H_t(\cdot))$ for a random t from $Z(\cdot)$, let $x_1, \ldots, x_q$ be the queries $\mathcal{A}$ makes to its oracle. (Without loss of generality, assume $x_1, \ldots, x_q$ are distinct.) If all of the hashes $H_t(x_1), \ldots, H_t(x_q)$ are distinct, then $Y(H_t(\cdot))$ and $Z(\cdot)$ will both output q uniformly random, independent values, so $\mathcal{A}$ will not be able to distinguish the two functions.

Therefore, $\mathcal{A}$'s advantage in distinguishing $Y(H_t(\cdot))$ from $Z(\cdot)$ is at most the probability of a collision among $H_t(x_1), \ldots, H_t(x_q)$. Let X denote the event that a collision occurs among $H_t(x_1), \ldots, H_t(x_q)$. Since Y is a uniformly random function, each output of $Y(H_t(\cdot))$ is a uniformly random, independent value (independent of the input and of t), until and unless X occurs. Once X occurs, the subsequent outputs of

$Y(H_t(\cdot))$ do not affect the probability of X. Therefore, to analyze the probability of X, we can think of $x_1, \ldots, x_q$ as being chosen before and independently of t.

There are at most q^2 pairs $i < j$, and by the ϵ-almost universality of H, for each pair there is at most an ϵ probability of a collision. Thus, the probability of X is at most $q^2\epsilon$, which is $\text{negl}(p(\lambda)) = \text{negl}(\lambda)$.

All together, $\mathcal{A}$'s distinguishing advantage is at most $\text{negl}(\lambda)$. □

B.5 Symmetric-Key Encryption

Definition B.6 (Symmetric-Key Encryption)**.** A *symmetric* (or *symmetric-key*) *encryption scheme* consists of the following PPT algorithms.

Gen(1^λ)**:** The key generation algorithm takes a security parameter λ and generates a secret key K.

Enc(K, m)**:** The encryption algorithm takes a secret key K and a message m and returns a ciphertext c. Note that *Enc* will be randomized, but we omit the randomness as an explicit input.

Dec(K, c)**:** The decryption algorithm is a deterministic algorithm that takes a secret key K and a ciphertext c and returns a message m or a special symbol $\perp$.

Correctness. For correctness, we require that for all λ and for all m, letting $K \leftarrow \mathit{Gen}(1^\lambda)$, we have $\mathit{Dec}(K, \mathit{Enc}(K, m)) = m$.

CPA Security. We require *indistinguishability under chosen-plaintext attacks (IND-CPA)*, or *CPA security*, which is defined using the following game. First, the challenger runs $\mathit{Gen}(1^\lambda)$ to generate a secret key K, which is kept hidden from the adversary. Next, the adversary is allowed to make any number of queries to an encryption oracle $\mathit{Enc}(K, \cdot)$. The adversary then outputs two equal-length challenge messages m_0 and m_1 and receives a challenge ciphertext equal to $\mathit{Enc}(K, m_b)$ for a random choice of $b \in \{0, 1\}$. The adversary can make more queries to the encryption oracle. Finally, it outputs a guess b' of the bit b. The adversary wins the game if $b' = b$.

The adversary's advantage is the difference between the probability that it wins the game and $1/2$ (from guessing randomly). CPA security says that no PPT adversary can win the above game with more than negligible advantage.

Definition B.7 (CPA security)**.** A symmetric encryption scheme $(\mathit{Gen}, \mathit{Enc}, \mathit{Dec})$ is *CPA-secure* if for all PPT adversaries $\mathcal{A}$,

$$|\Pr[K \leftarrow \mathit{Gen}(1^\lambda); (m_0, m_1) \leftarrow \mathcal{A}^{\mathit{Enc}(K,\cdot)}(1^\lambda); b \xleftarrow{\text{R}} \{0,1\}; c \leftarrow \mathit{Enc}(K, m_b); \\ b' \leftarrow \mathcal{A}^{\mathit{Enc}(K,\cdot)}(c) :\ b' = b] - 1/2| \leq \text{negl}(\lambda) \ ,$$

where the two messages (m_0, m_1) output by $\mathcal{A}$ must be of equal length.

Which-Key Concealing. We will also require symmetric encryption schemes to satisfy a property called *which-key concealing*. The which-key concealing property was introduced by Abadi and Rogaway [AR02] and (under the name "key hiding") by Fischlin [Fis99].

The which-key-concealing requirement says, roughly, that an adversary cannot tell whether ciphertexts are encrypted under the same key or different keys. More formally, which-key concealing is defined via a game, in which the adversary tries to distinguish between the following two experiments. In one experiment, *Gen*(1^λ) is run twice, to generate two keys K and K'. The adversary is given a "left" oracle *Enc*$(K, \cdot)$ and a "right" oracle *Enc*$(K', \cdot)$, to both of which it is allowed to make any number of queries. The adversary then outputs a bit. The other experiment is the same, except that only one key K is generated, and both of the left and right oracles output *Enc*$(K, \cdot)$. The adversary's advantage is the difference between the probability that it outputs 1 in the two experiments. Which-key concealing says that no PPT adversary can win the above game with more than negligible advantage. Note that in order for an encryption scheme to be which-key-concealing, clearly it must be randomized.

Definition B.8 (Which-Key Concealing)**.** A symmetric encryption scheme (*Gen*, *Enc*, *Dec*) is *which-key-concealing* if for all PPT adversaries $\mathcal{A}$,

$$|\Pr[K \leftarrow \mathit{Gen}(1^\lambda); K' \leftarrow \mathit{Gen}(1^\lambda); \mathcal{A}^{\mathit{Enc}(K,\cdot),\mathit{Enc}(K',\cdot)}(1^\lambda) = 1] - \Pr[K \leftarrow \mathit{Gen}(1^\lambda); \mathcal{A}^{\mathit{Enc}(K,\cdot),\mathit{Enc}(K,\cdot)}(1^\lambda) = 1]| \leq \mathrm{negl}(\lambda) \ .$$

Let us now see an example of a symmetric encryption scheme that is CPA-secure and which-key-concealing.

Example B.9 ($\mathcal{E}_{\text{xor}}$)**.** Let p be a polynomial, and let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^{p(\lambda)}$ be a PRF. The encryption scheme $\mathcal{E}_{\text{xor}}$ for message space $\mathcal{M} = \{0,1\}^{p(\lambda)}$ is defined as follows.

Gen(1^λ)**:** Let $K \stackrel{\mathrm{R}}{\leftarrow} \{0,1\}^\lambda$ be the secret key.

Enc(K, m)**:** Let $r \stackrel{\mathrm{R}}{\leftarrow} \{0,1\}^\lambda$, and output $c = (r, F_K(r) \oplus m)$.

Dec(K, c)**:** Let $c = (r, x)$. Output $m = x \oplus F_K(r)$.

Bellare et al. [BDJR97] proved that the above scheme is CPA-secure:

Theorem B.10. *[BDJR97] If F is a PRF, then $\mathcal{E}_{\text{xor}}$ is CPA-secure.*

We will prove that $\mathcal{E}_{\text{xor}}$ is which-key-concealing.

Theorem B.11. *If F is a PRF, then $\mathcal{E}_{\text{xor}}$ is which-key-concealing.*

Proof. Let $\mathcal{A}$ be an adversary playing the which-key-concealing game.

We first replace $\mathcal{E}_{\text{xor}}$ in the which-key-concealing game with a modified scheme $\mathcal{E}'_{\text{xor}}$. $\mathcal{E}'_{\text{xor}}$ is the same as $\mathcal{E}_{\text{xor}}$, except that F_K is replaced with a uniformly random function $R : \{0,1\}^\lambda \to \{0,1\}^{p(\lambda)}$. By the PRF property of F, replacing $\mathcal{E}_{\text{xor}}$ with $\mathcal{E}'_{\text{xor}}$ can change $\mathcal{A}$'s advantage in the which-key-concealing game by at most a negligible quantity.

So, suppose $\mathcal{A}$ is playing the which-key-concealing game for $\mathcal{E}'_{\text{xor}}$. Suppose $\mathcal{A}$ makes a total of q queries, $m_1, \ldots, m_q$, to its encryption oracles, where each m_i is a query to either the left or the right oracle. Let (r_i, x_i) denote the answer to the ith query, and let $y_i = x_i \oplus m_i$.

If there are any i and j such that $r_i = r_j$ and m_i is a query to the left oracle while m_j is a query to the right oracle, then $\mathcal{A}$ will be able to distinguish whether the two oracles use the same key based on whether $y_i = y_j$. However, if all of $r_1, \ldots, r_q$ are distinct, then for any key the encryption algorithm will choose each y_i as a uniformly random, independent value, so $\mathcal{A}$ will gain no information about which experiment it is in and can do no better than a random guess.

Thus, $\mathcal{A}$'s advantage in winning the which-key-concealing game for $\mathcal{E}'_{\text{xor}}$ is at most the probability that any of $r_1, \ldots, r_q$ are equal, which is upper bounded by $q^2/2^\lambda$. Combining this with the negligible difference in $\mathcal{A}$'s advantage against $\mathcal{E}'_{\text{xor}}$ and against $\mathcal{E}_{\text{xor}}$, we have that $\mathcal{A}$'s advantage in winning the which-key-concealing game for $\mathcal{E}_{\text{xor}}$ is negligible. $\square$

Ciphertext integrity and authenticated encryption. We will also sometimes require a symmetric encryption scheme to have a property called *ciphertext integrity*. The notions of ciphertext integrity and *authenticated encryption* (defined below) were introduced by [BR00, KY01, BN00]. Ciphertext integrity says, roughly, that an adversary given encryptions of messages of its choice cannot construct any new ciphertexts that decrypt successfully (i.e., decrypt to a value other than $\perp$).

Formally, ciphertext integrity is defined using the following game. First, the challenger runs *Gen*(1^λ) to generate a secret key K, which is kept hidden from the adversary. The adversary then adaptively makes a polynomial number of queries, $m_1, \dots, m_q$. To each query m_i the challenger responds by sending $c_i \leftarrow$ *Enc*(K, m_i) to the adversary. Finally, the adversary outputs a value c. The adversary wins the game if c is not among the previously received ciphertexts $\{c_1, \dots, c_q\}$ and *Dec*$(K, c) \neq \perp$.

We define the advantage of an adversary $\mathcal{A}$ in attacking the ciphertext integrity of a symmetric encryption scheme as the probability that $\mathcal{A}$ wins the above game.

Definition B.12 (Ciphertext integrity)**.** A symmetric encryption scheme $(\mathit{Gen}, \mathit{Enc}, \mathit{Dec})$ has ciphertext integrity if for all PPT adversaries $\mathcal{A}$, $\mathcal{A}$'s advantage in the above game is at most $\mathrm{negl}(\lambda)$.

Definition B.13 (Authenticated encryption)**.** A symmetric encryption scheme is an *authenticated encryption scheme* if it has CPA security and ciphertext integrity.

Let us now see an example of an authenticated encryption scheme. One way to construct an authenticated encryption scheme is "encrypt-then-MAC" [BN00]. (A MAC is a message authentication code, the details of which we do not give here; instead, we will just use the fact that a PRF defines a secure MAC.) Using encrypt-then-MAC, one first encrypts a message m with a CPA-secure scheme to get a ciphertext c', and then computes a MAC of c' to get a tag t. The ciphertext is then $c = (c', t)$. The decryption algorithm verifies that t is a valid tag for c' and then decrypts c' using the CPA-secure scheme. In the following example, we apply encrypt-then-MAC to $\mathcal{E}_{\text{xor}}$ to obtain an authenticated encryption scheme $\mathcal{E}_{\text{xor-auth}}$ (which, like $\mathcal{E}_{\text{xor}}$, is also which-key-concealing).

Example B.14 ($\mathcal{E}_{\text{xor-auth}}$)**.** Let p be a polynomial, and let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^{p(\lambda)}$ and $G : \{0,1\}^\lambda \times \{0,1\}^{\lambda+p(\lambda)} \to \{0,1\}^\lambda$ be PRFs. The encryption scheme $\mathcal{E}_{\text{xor-auth}}$ for message space $\mathcal{M} = \{0,1\}^{p(\lambda)}$ is defined as follows.

Gen(1^λ)**:** Choose $K_1, K_2 \xleftarrow{\text{R}} \{0,1\}^\lambda$ and let $K = (K_1, K_2)$ be the secret key.

Enc(K, m)**:** Let $r \xleftarrow{\text{R}} \{0,1\}^\lambda$, and output $c = (r, x = F_{K_1}(r) \oplus m, t = G_{K_2}(r||x))$.

Dec(K, c)**:** Let $c = (r, x, t)$. If $t \neq G_{K_2}(r||x)$, output $\perp$. Otherwise, output $m = x \oplus F_{K_1}(r)$.

Theorem B.15. $\mathcal{E}_{\text{xor-auth}}$ *is an authenticated encryption scheme.*

Proof. The theorem follows directly from the following facts: (1) G is a PRF and therefore defines a MAC, (2) $\mathcal{E}_{\text{xor}}$ is CPA-secure, and (3) applying encrypt-then-MAC to a CPA-secure scheme gives an authenticated encryption scheme [BN00]. $\square$

$\mathcal{E}_{\text{xor-auth}}$ also retains the which-key-concealing property of $\mathcal{E}_{\text{xor}}$.

Theorem B.16. $\mathcal{E}_{\text{xor-auth}}$ *is which-key-concealing.*

Proof. The proof is very similar to the which-key-concealing proof for $\mathcal{E}_{\text{xor}}$. We first replace $\mathcal{E}_{\text{xor-auth}}$ with a modified scheme $\mathcal{E}'_{\text{xor-auth}}$ in which F and G are replaced with random functions R_1 and R_2, respectively. By the PRF property of F and G, this changes $\mathcal{A}$'s advantage in winning the which-key-concealing game by at most a negligible quantity. Now, suppose $\mathcal{A}$ makes q encryption queries, $m_1, \ldots, m_q$. Let (r_i, x_i, t_i) denote the response to the ith query, and let $y_i = x_i \oplus m_i$. If all of $r_1, \ldots, r_q$ are distinct and all of $r_1||x_1, \ldots, r_q||x_q$ are distinct, then for any key the encryption algorithm will choose all of the y_i and t_i as uniformly random, independent values, so $\mathcal{A}$ will gain no information about which experiment it is in. But if all of the r_i are distinct, then so are all of the $r_i||x_i$. Therefore, $\mathcal{A}$'s advantage against $\mathcal{E}'_{\text{xor-auth}}$ is at most the probability that any of the r_i are equal, which is upper bounded by $q^2/2^\lambda$. All together, $\mathcal{A}$'s advantage against $\mathcal{E}_{\text{xor-auth}}$ is negligible. □

C Correctness Against Malicious Adversaries

We will show that $\mathcal{E}_{\text{PM}}$ is correct against malicious adversaries.

Theorem C.1. *If $\mathcal{E}_{\text{SKE}}$ is an authenticated encryption scheme, then $\mathcal{E}_{\text{PM}}$ is correct against malicious adversaries.*

Proof. It is fairly straightforward to see that if the adversary executes *AnswerQuery* honestly, then the client's output will be correct.

We will argue that for each of the places where $\mathcal{A}$ could output an incorrect value, the client will detect $\mathcal{A}$'s cheating and output $\perp$, with all but negligible probability.

Lemma C.2. *If $\mathcal{E}_{\text{SKE}}$ is an authenticated encryption scheme, then if an adversary $\mathcal{A}$ outputs an incorrect W in the query protocol, the client's response to W will be $\perp$, with all but negligible probability.*

Proof. In the protocol for a query p, the client runs $\mathcal{E}_{\text{SKE}}.\textit{Dec}(K_D, W)$ to get either $\perp$ or a tuple X, which it parses as $(\textit{ind}, \textit{lpos}, \textit{num}, \textit{len}, f_1, f_{2,1}, \ldots, f_{2,d})$. The client outputs $\perp$ if any of the following events occur:

- (Event $W.1$) $\mathcal{E}_{\text{SKE}}.\textit{Dec}(K_D, W) = \perp$, or
- (Event $W.2$) W decrypts successfully, but $f_1 \neq F_{K_1}(p[1..\textit{len}])$, or
- (Event $W.3$) W decrypts successfully and $f_1 = F_{K_1}(p[1..\textit{len}])$, but $\mathcal{E}_{\text{SKE}}.\textit{Dec}(f_{2,i}, T_j) \neq \perp$ for some $i \in \{1, \ldots, d\}$, $j > \textit{len}$.

On the other hand, if the adversary cheats, then W is not the ciphertext in the dictionary entry $D(F_{K_1}(p[1..i]))$, where $p[1..i]$ is the longest matching prefix of p, which means one of the following events:

- (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 κ is not equal to $F_{K_1}(p[1..i])$ 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, $W = D(F_{K_1}(p[1..i]))$ for some i, but there exists a $j > i$ such that there is an entry $D(F_{K_1}(p[1..j]))$.

We want to show that if the adversary cheats, then the client will output $\perp$.

If event $W.1'$ occurs, then we will show below that $W.1$ occurs with all but negligible probability, by the ciphertext integrity of $\mathcal{E}_{\text{SKE}}$.

If event $W.2'$ occurs, then event $W.2$ occurs with all but negligible probability upper bounded by $1/2^{\lambda}$, the probability that $F_{K_1}(p[1..len]) = f_1$ when f_1 is an independent, random value.

If event $W.3'$ occurs, then clearly $W.3$ also occurs.

It remains to show that event $W.1'$ implies event $W.1$ with all but negligible probability.

Suppose an adversary $\mathcal{A}$ causes event $W.1'$ but not event $W.1$. Then the W output by $\mathcal{A}$ is not among the ciphertexts in the dictionary, but $\mathcal{E}_{\text{SKE}}.Dec(K_D, W) \neq \perp$. Then we can use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ that breaks ciphertext integrity of $\mathcal{E}_{\text{SKE}}$. Algorithm $\mathcal{B}$ executes $\mathcal{E}_{\text{PM}}$ honestly, except that in the encryption algorithm, instead of generating each W_u as $\mathcal{E}_{\text{SKE}}.Enc(K_D, X_u)$, it queries its encryption oracle on X_u and uses the resulting ciphertext as c_j. Then, when $\mathcal{A}$ outputs W, $\mathcal{B}$ outputs W in the ciphertext integrity game. Note that $\mathcal{A}$'s view is the same as when it is interacting with the real scheme $\mathcal{E}_{\text{PM}}$. If W is not among the ciphertexts in D, but $\mathcal{E}_{\text{SKE}}.Dec(K_D, W) \neq \perp$, then $\mathcal{B}$ wins the ciphertext integrity game. Therefore, if $\mathcal{A}$ has probability ϵ of causing event $W.1'$ but not event $W.1$, $\mathcal{B}$ wins the ciphertext integrity game with the same probability ϵ. □

Lemma C.3. *If $\mathcal{E}_{\text{SKE}}$ is an authenticated encryption scheme, then if an adversary $\mathcal{A}$ outputs incorrect $C_1, \ldots, C_m$ in the query protocol, the client's response to $C_1, \ldots, C_m$ will be $\perp$, with all but negligible probability.*

Proof. In the query protocol, for each i, the client outputs $\perp$ if either of the following events occur:

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

On the other hand, if the adversary cheats and outputs incorrect $C_1, \ldots, C_m$, then for some i, $C_i \neq C[x_i]$, which means either of the following events:

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

We want to show that if the adversary cheats, then the client will output $\perp$.

For any i, if event $C.1'$ occurs, then we will show below that event $C.1$ occurs with all but negligible probability, by the ciphertext integrity of $\mathcal{E}_{\text{SKE}}$.

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.

It remains to show that for any i event $C.1'$ implies event $C.1$, with all but negligible probability. Suppose an adversary $\mathcal{A}$ causes event $C.1'$ but not event $C.1$. Then C_i is not among $C[1], \ldots, C[n]$, but $\mathcal{E}_{\text{SKE}}.Dec(K_C, C_i) \neq \perp$. Then we can use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ that breaks ciphertext integrity of $\mathcal{E}_{\text{SKE}}$. $\mathcal{B}$ executes $\mathcal{E}_{\text{PM}}$ honestly, except that in the encryption algorithm, instead of generating each c_j as $\mathcal{E}_{\text{SKE}}.Enc(K_C, (s_j, j))$, it queries its encryption oracle on (s_j, j) and uses the resulting ciphertext as c_j. Then, when $\mathcal{A}$ outputs $C_1, \ldots, C_m$, $\mathcal{B}$ chooses a random $i' \xleftarrow{\text{R}} \{1, \ldots, m\}$ and outputs $C_{i'}$ in the ciphertext integrity game. Note that $\mathcal{A}$'s view is the same as when it is interacting with the real scheme $\mathcal{E}_{\text{PM}}$. If C_i is not among $C[1], \ldots, C[n]$, but $\mathcal{E}_{\text{SKE}}.Dec(K_C, C_i) \neq \perp$, then $\mathcal{B}$ wins the ciphertext integrity game if $i' = i$. Therefore, if $\mathcal{A}$ has probability ϵ of causing event $C.1'$ but not event $C.1$ for any i, $\mathcal{B}$ wins the ciphertext integrity game with probability at least ϵ/m. □

Lemma C.4. *If $\mathcal{E}_{\text{SKE}}$ is an authenticated encryption scheme, then if an adversary $\mathcal{A}$ outputs incorrect $L_1, \ldots, L_{num}$ in the query protocol, the client's response to $L_1, \ldots, L_{num}$ will be $\perp$, with all but negligible probability.*

The proof is omitted, since it almost identical to the proof of Lemma C.3.

We have shown that if an adversary $\mathcal{A}$ cheats when producing any of its outputs to the client, the client will output $\perp$ with all but negligible probability. Therefore, $\mathcal{E}_{\text{PM}}$ is correct against malicious adversaries. $\square$

D Security

We now prove that our pattern matching encryption scheme satisfies malicious-$(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security for certain leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$.

D.1 Leakage

Before we describe the leakage of our scheme, we define some relevant notions.

We say that a query p *visits* a node u in the suffix tree *Tree*$_s$ for s if $\hat{\rho}(u)$ is a prefix of p. For any j let p_j denote the jth query, and let $m_j = |p_j|$. Let n_j denote the number of nodes visited by the query for p_j in s, let $u_{j,i}$ denote the ith such node, and let $len_{j,i} = |\hat{\rho}(u_{j,i})|$. Let num_j denote the number of occurrences of p_j as a substring of s. Let ind_j denote the index ind in the ciphertext W returned by *AnswerQuery* for p_j. Note that ind_j is the index in s of the longest matching prefix of p_j, which is also the index in s of the longest prefix of p_j that is a substring of s. Let $lpos_j$ denote the leaf index $lpos$ in the ciphertext W returned by *AnswerQuery* for p_j. If p_j is a substring of s, $lpos_j$ is equal to the position (between 1 and n, from left to right) of the leftmost leaf ℓ for which p_j is a prefix of $\hat{\rho}(\ell)$.

The query prefix pattern for a query p_j tells which of the previous queries $p_1, \ldots, p_{j-1}$ visited each of the nodes visited by p_j.

Definition D.1 (Query prefix pattern)**.** The *query prefix pattern* QP$(s, p_1, \ldots, p_j)$ is a sequence of length n_j, where the ith element is a list $list_i$ of indices $j' < j$ such that the j'th query also visited $u_{j,i}$.

The index intersection pattern for a query p_j essentially tells when any of the indices $ind_j, \ldots, ind_j + m_j - 1$ are equal to or overlap with any of the indices $ind_i, \ldots, ind_i + m_i - 1$ for any previous queries p_i.

Definition D.2 (Index intersection pattern)**.** The *index intersection pattern* IP$(s, p_1, \ldots, p_j)$ is a sequence of length j, where the ith element is equal to $r_1[\{ind_i, \ldots, ind_i + m_i - 1\}]$ for a fixed random permutation $r_1 : [n] \to [n]$.

The leaf intersection pattern for a query p_j essentially tells when any of the leaf positions $lpos_j, \ldots, lpos_j + num_j - 1$ are equal to or overlap with any of the leaf positions $lpos_i, \ldots, lpos_i + num_i - 1$ for any previous queries p_i.

Definition D.3 (Leaf intersection pattern)**.** The *leaf intersection pattern* LP$(s, p_1, \ldots, p_j)$ is a sequence of length j, where the ith element is equal to $r_2[\{lpos_i, \ldots, lpos_i + num_i - 1\}]$ for a fixed random permutation $r_2 : [n] \to [n]$.

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

$$(m_j = |p_j|,\ \{len_{j,i}\}_{i=1}^{n_j},\ \text{QP}(s, p_1, \ldots, p_j),\ \text{IP}(s, p_1, \ldots, p_j),\ \text{LP}(s, p_1, \ldots, p_j))\ .$$

For example, consider the string s "cocoon" (whose suffix tree is shown in Figure 1) and a sequence of three queries, $p_1 =$ "co", $p_2 =$ "coco", and $p_3 =$ "cocoa". Then the leakage $\mathcal{L}_1(s)$ is $n = 6$.

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 positions retrieved are 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).

Thus, the leakage $\mathcal{L}_2(s, p_1, p_2, p_3)$ consists of:

- the lengths $2, 4, 5$ of the patterns,
- 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.

D.2 Malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 Security

Theorem D.4. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be defined as in Section D.1. If F is a PRF, P is a PRP, and $\mathcal{E}_{\text{SKE}}$ is a CPA-secure, key-private symmetric-key encryption scheme, then the pattern matching encryption scheme $\mathcal{E}_{\text{PM}}$ satisfies malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security.*

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

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

1. Construct a dictionary D as follows. For $i = 1, \ldots, 2n$, choose fresh random values $\kappa_i, f_{2,1}, \ldots, f_{2,d}, \xleftarrow{\text{R}} \{0,1\}^\lambda$, and store $V_i = (f_{2,1}, \ldots, f_{2,d}, W = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_D, 0))$ with search key κ_i in D.
2. Choose an arbitrary element $\sigma_0 \in \Sigma$. Construct an array C, where $C[i] = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (\sigma_0, 0))$ for $i = 1, \ldots, n$.
3. Construct an array L, where $L[i] = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_L, 0)$ for $i = 1, \ldots, n$.

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

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

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

$\mathcal{T}_2$ contains all currently defined tuples $(\kappa, f_2, flag, flag_1, \ldots, flag_d)$, 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 $flag_i$ indicates whether child$(u, \pi_u(i))$ has been visited. The value of each flag is either "visited" or "unvisited". We write $\mathcal{T}_2(\kappa) = (f_2, flag, flag_1, \ldots, flag_d)$ if $(\kappa, f_2, flag, flag_1, \ldots, flag_d)$ is an entry in $\mathcal{T}_2$.

Choose an arbitrary entry (κ^*, V^*) in D to represent the root node of *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.) Define $\mathcal{T}_1(i, 0) = \kappa^*$ for any i.

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

For $t = 1, \ldots, n_j$, if $list_t =$ 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 $list_t$. Let $\kappa = \mathcal{T}_1(j', t)$ and let $(f_2, flag, flag_1, \ldots, flag_d) = \mathcal{T}_2(\kappa)$. $T_{len_{j,t}} = \mathcal{E}_{\text{SKE}}.Enc(f_2, \kappa)$. Set $\mathcal{T}_1(j, t) = \kappa$.

If instead $list_t$ is empty, choose a random unused entry (κ, 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 $(f_2, flag, flag_1, \ldots, flag_d) = \mathcal{T}_2(\kappa')$. Choose a random $i \in \{1, \ldots, d\}$ such that $flag_i$ is "unvisited", and set $flag_i$ to "visited". Let $f_{2,i}$ be $D(\kappa').f_{2,i}$. Set $T_{len_t} = \mathcal{E}_{\text{SKE}}.Enc(f_{2,i}, \kappa)$, set $\mathcal{T}_2(\kappa).f_2 = f_{2,i}$, set $\mathcal{T}_2(\kappa).flag$ to "visited", and set $\mathcal{T}_2(\kappa).flag_i$ to "unvisited" for $i = 1, \ldots, d$.

For any $i \neq len_t$ for any $t = 1, \ldots, n_j$, choose a random $f_2 \xleftarrow{R} \{0,1\}^\lambda$, and let $T_i = \mathcal{E}_{\text{SKE}}.Enc(f_2, 0)$.

Send $(T_1, \ldots, T_m)$ to the adversary.

Upon receiving a W from the adversary, check whether $W = D(\mathcal{T}_1(j, n_j)).W$. If not, output $\perp$. Otherwise, let $(x_1, \ldots, x_m)$ be a random ordering of the elements of the set IP$(p_j, s)[j]$, and send $(x_1, \ldots, x_m)$ to the adversary.

Upon receiving $C_1, \ldots, C_m$ from the adversary, check whether $C_i = C[x_i]$ for each i. If not, output $\perp$. Otherwise, let $(y_1, \ldots, y_{num})$ be a random ordering of the elements of LP$(p_j, s)[j]$, and send $(y_1, \ldots, y_{num})$ to the adversary.

Upon receiving $L_1, \ldots, L_{num}$ from the adversary, check whether $L_i = L[y_i]$ for each i. If not, output $\perp$.

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 $G_0, \ldots, G_{16}$ 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 *Gen*(1^λ) to generate a key K.
- The adversary $\mathcal{A}$ outputs a string s and receives $CT \leftarrow$ *Enc*(K, s) from the challenger.
- $\mathcal{A}$ adaptively chooses patterns $p_1, \ldots, p_q$. For each p_i, $\mathcal{A}$ first interacts with the challenger, who is running *IssueQuery*(K, p_i) honestly. Then $\mathcal{A}$ outputs a description of a function g_i, and receives $g_i(A_1, \ldots, A_i)$ from the challenger, where A_i is the challenger's private output from the interactive protocol for p_i.

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

Lemma D.5. *If F is a PRF, then G_0 and G_1 are indistinguishable, except with negligible probability.*

Proof. We consider a hybrid game H_1. H_1 is the same as G_0 except that it uses R_1 in place of F_{K_1}.

Suppose an adversary $\mathcal{A}$ can distinguish G_0 from H_1. Then we can construct an algorithm $\mathcal{B}$ that attacks the PRF property of F with the same advantage. $\mathcal{B}$ acts as $\mathcal{A}$'s challenger in G_0, except that whenever there is a call to $F_{K_1}(x)$, $\mathcal{B}$ queries its oracle on x. When $\mathcal{A}$ outputs a guess bit, $\mathcal{B}$ outputs the same guess bit. If $\mathcal{B}$'s oracle is a function from F, $\mathcal{A}$'s view will be the same as in game G_0, while if it is a random function, $\mathcal{A}$'s view will be the same as in game H_1. Thus, $\mathcal{B}$ answers its challenge correctly whenever $\mathcal{A}$ does, and breaks the PRF property of F with the same advantage that $\mathcal{A}$ distinguishes games G_0 and H_1.

A similar argument shows that games H_1 and G_1 are indistinguishable by the PRF property of F. Thus, we conclude that G_0 and G_1 are indistinguishable. □

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 hybrid argument similar to the one used for G_0 and G_1 shows that G_1 and G_2 are indistinguishable by the PRP property of P.

Lemma D.6. *If P is a PRP, then G_2 and G_1 are indistinguishable, except with negligible probability.*

Proof. We consider a hybrid game H_1. Game H_1 is the same as G_1 except that it uses R_3 in place of P_{K_3}.

Suppose $\mathcal{A}$ can distinguish G_0 from H_1. Then we can construct an algorithm $\mathcal{B}$ that attacks the PRF property of F with the same advantage. $\mathcal{B}$ acts as $\mathcal{A}$'s challenger in G_1, except that whenever there is a call to $P_{K_3}(x)$, $\mathcal{B}$ queries its oracle on x. When $\mathcal{A}$ outputs a guess bit, $\mathcal{B}$ outputs the same guess bit. If $\mathcal{B}$'s oracle is a function from P, $\mathcal{A}$'s view will be the same as in game G_1, while if it is a random permutation, $\mathcal{A}$'s view will be the same as in game H_1. Thus, $\mathcal{B}$ answers its challenge correctly whenever $\mathcal{A}$ does, and breaks the PRP property of P with the same advantage that $\mathcal{A}$ distinguishes games G_1 and H_1.

A similar argument shows that games H_1 and G_2 are indistinguishable by the PRP property of P. Thus, we conclude that G_1 and G_2 are indistinguishable. □

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 $C_1, \ldots, C_m$ from the adversary in response to indices $x_1, \ldots, x_m$, the simulator's decision whether to output $\perp$ is not based on the decryptions of $C_1, \ldots, C_m$. Instead, it outputs $\perp$ if $C_i \neq C[x_i]$ for any i. Otherwise, the simulator proceeds as in G_2.

We argue that games G_3 and G_2 are indistinguishable by the ciphertext integrity of $\mathcal{E}_{\text{SKE}}$.

Lemma D.7. *If $\mathcal{E}_{\text{SKE}}$ 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 $C_1, \ldots, C_m$.

For each i, G_2 outputs $\perp$ if either of the following events occur:

- (Event $C.1$) $\mathcal{E}_{\text{SKE}}.\mathit{Dec}(K_C, C_i) = \perp$, or

- (Event $C.2$) $\mathcal{E}_{\text{SKE}}.\mathit{Dec}(K_C, C_i) = (p'_i, j)$ where j is not the correct index.

For each i, G_3 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 $C[1], \ldots, C[n]$, or
- (Event $C.2'$) $C_i = C[k]$ where $k \neq x_i$.

If G_3 outputs $\perp$ for some i then G_2 outputs $\perp$ except with negligible probability, as we already showed by ciphertext integrity of $\mathcal{E}_{\text{SKE}}$ in Lemma C.3 in the proof of correctness of $\mathcal{E}_{\text{PM}}$ against malicious adversaries.

If G_2 outputs $\perp$, if event $C.1$ occurred, then $C.1'$ also occurred, since C_i will decrypt successfully if it is one of $C[1], \ldots, C[n]$. 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_2 outputs $\perp$ for some i, so does G_3.

Thus, G_2 and G_3 are indistinguishable except with negligible probability. □

Game G_4**.** This game is the same as G_3, except for the following differences. The simulator does not decrypt the $C_1, \ldots, C_m$ from the adversary. For any query p, instead of deciding whether to output $\emptyset$ based on the decryptions of $C_1, \ldots, C_m$, the simulator outputs $\emptyset$ if p is not a substring of s. Otherwise, the simulator proceeds as in G_3.

As we showed in Lemmas C.2 and C.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 $C_1, \ldots, C_m$, 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 $C_1, \ldots, C_m$ are necessarily the correct ciphertexts, and the decryptions $p'_1, \ldots, p'_m$ computed in G_3 match p if and only if p is a substring of s. Therefore, G_3 and G_4 are indistinguishable.

Game G_5 . This game is the same as G_4, except that in G_5, for $i = 1, \ldots, n$, instead of setting $c_i = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (s_i, i))$, the simulator sets $c_i = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (\sigma_0, 0))$, where σ_0 is an arbitrary element of Σ.

Note that in both G_4 and G_5, K_C is hidden and the c_i's are never decrypted. A hybrid argument shows that games G_4 and G_5 are indistinguishable by CPA security of $\mathcal{E}_{\text{SKE}}$.

Lemma D.8. *If $\mathcal{E}_{\text{SKE}}$ is a CPA-secure encryption scheme, then G_4 and G_5 are indistinguishable, except with negligible probability.*

Proof. We show this via a series of $n+1$ hybrid games $H_0, \ldots, H_n$. Let σ_0 be an arbitrary element of Σ. In H_i, during the encryption phase, for $i' \leq i$, the simulator computes $c_{i'}$ as $\mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (\sigma_0, 0))$. For $i' > i$, it computes $c_{i'}$ as $\mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (s_{i'}, i'))$. The rest of the game proceeds as in G_4. Note that $H_0 = G_4$ and $H_n = G_5$.

If there is an adversary $\mathcal{A}$ that can distinguish H_{i-1} from H_i for any $i \in \{1 \ldots n\}$, then we can construct an algorithm $\mathcal{B}$ that attacks the CPA security of $\mathcal{E}_{\text{SKE}}$ with the same advantage.

$\mathcal{B}$ acts as the simulator in H_{i-1}, with the following exceptions. During the encryption phase, for $i' < i$, $\mathcal{B}$ generates $c_{i'}$ by querying the encryption oracle on $(\sigma_0, 0)$, and for $i' > i$, $\mathcal{B}$ generates $c_{i'}$ by querying the encryption oracle on $(s_{i'}, i')$. $\mathcal{B}$ outputs $(s_i, i), (\sigma_0, 0)$ as its challenge, and uses the challenge ciphertext as c_i.

Now, if $\mathcal{B}$'s challenger returns an encryption of (s_i, i), then $\mathcal{A}$'s view will be the same as in H_{i-1}, while if the challenger returns an encryption of $(\sigma_0, 0)$, then $\mathcal{A}$'s view will be the same as in H_i. Thus, $\mathcal{B}$ answers its challenge correctly whenever $\mathcal{A}$ does, and breaks the CPA security of $\mathcal{E}_{\text{SKE}}$ with the same advantage that $\mathcal{A}$ distinguishes games H_{i-1} and H_i.

Since there are a polynomial number of games $H_0, \ldots, H_n$, we conclude that $H_0 = G_4$ and $H_n = G_5$ are indistinguishable. $\square$

Game G_6**.** This game is the same as G_5, except that we eliminate the use of the random permutation R_3, in the following way. For $i = 1, \ldots, n$, the simulator set $C[i] = c_i$ instead of $C[R_3(i)] = c_i$, where $c_i = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_C, (\sigma_0, 0))$. Furthermore, for any query p_j, the simulator is given an additional input $\text{IP}(s, p_1, \ldots, p_j)$ (as defined in Section D.1). To generate $(x_1, \ldots, x_m)$ in the query protocol, the simulator outputs a random ordering of the elements in $\text{IP}(s, p_1, \ldots, p_j)[j]$.

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 indistinguishable. After we eliminate the use of R_3 in generating C, R_3 is only used by the simulator to compute $(x_1, \ldots, x_m)$. Thus, we can replace the computation of $(x_1, \ldots, x_m)$ for each query p_j with a random ordering of the elements of $\text{IP}(s, p_1, \ldots, p_j)[j]$, and the result will be indistinguishable.

Game G_7**.** This is the same as G_6, except that we modify the simulator as follows. For any query, when the simulator receives $L_1, \ldots, L_{num}$ from the adversary in response to indices $y_1, \ldots, y_{num}$, the simulator's decision whether to output $\perp$ is not based on the decryptions of the $L_1, \ldots, L_{num}$; 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_6.

A hybrid argument shows that games G_6 and G_7 are indistinguishable by the ciphertext integrity of $\mathcal{E}_{\text{SKE}}$.

Lemma D.9. *If $\mathcal{E}_{\text{SKE}}$ has ciphertext integrity, then G_6 and G_7 are indistinguishable, except with negligible probability.*

The proof is omitted since it is nearly identical to the proof for G_2 and G_3.

Game G_8**.** This game is the same as G_7, except for the following differences. The simulator does not decrypt the $L_1, \ldots, L_{num}$ from the adversary. For any query p_j, instead of computing the answer A_j using the decryptions of $L_1, \ldots, L_{num}$, if A_j has not already been set to $\perp$ or $\emptyset$, the simulator sets $A_j = \mathcal{F}(s, p_j)$.

As we showed in Lemmas C.2, C.3, and C.4, if any of the W, $C_1, \ldots, C_m$ or $L_1, \ldots, L_{num}$ from the adversary are incorrect, the client will respond to the incorrect message with $\perp$. Therefore, if the simulator has not yet output $\perp$ when it is computing A_j, then the adversary has executed *AnswerQuery* honestly, and $A_j = \mathcal{F}(s, p_j)$ (by correctness of $\mathcal{E}_{\text{PM}}$). Therefore, G_7 and G_8 are indistinguishable.

Game G_9**.** This game is the same as G_8, except that in G_9, for each $i = 1, \ldots, n$, the simulator generates each ℓ_i as $\mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_L, 0)$ instead of $\mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_L, (\mathit{ind}_{\mathit{leaf}_i}, i))$.

A hybrid argument shows that G_8 and G_9 are indistinguishable by the CPA security of $\mathcal{E}_{\text{SKE}}$.

Lemma D.10. *If $\mathcal{E}_{\text{SKE}}$ is a CPA-secure encryption scheme, then G_8 and G_9 are indistinguishable, except with negligible probability.*

The proof is omitted since it is nearly identical to the proof for G_4 and G_5.

Game G_{10}**.** This game is the same as G_9, except that we eliminate the use of the random permutation R_4, in the following way. For $i = 1, \ldots, n$, the simulator set $L[i] = \ell_i$ instead of $L[R_4(i)] = \ell_i$, where $\ell_i = \mathcal{E}_{\text{SKE}}.\mathit{Enc}(K_L, 0)$. Furthermore, for any query p_j, the simulator is given an additional input $\text{LP}(s, p_1, \ldots, p_j)$ (as defined in Section D.1). To generate $(y_1, \ldots, y_{num})$ in the query protocol, the simulator outputs a random ordering of the elements in $\text{LP}(s, p_1, \ldots, p_j)[j]$.

The argument for game G_{10} is analogous to the one for game G_6. Since each ℓ_i is an encryption under K_L of 0, it does not matter whether the ℓ_i's are permuted in L; if we permute the ℓ_i's or not, the result is indistinguishable. After we eliminate the use of R_4 in generating L, R_4 is only used by the simulator to compute $(y_1, \ldots, y_{num})$. Thus, we can replace the computation of $(y_1, \ldots, y_{num})$ for each query p_j with a random ordering of the elements of $\text{LP}(s, p_1, \ldots, p_j)[j]$, and the result will be indistinguishable.

Game G_{11}**.** This is the same as G_{10}, except that we modify the simulator as follows. For any query, when the simulator receives a W from the adversary in response to $T_1, \ldots, T_m$, the simulator's decision whether to output $\perp$ will not 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_{10}.

We argue that games G_{10} and G_{11} are indistinguishable by the ciphertext integrity of $\mathcal{E}_{\text{SKE}}$.

Lemma D.11. *If $\mathcal{E}_{\text{SKE}}$ has ciphertext integrity, then G_{10} and G_{11} are indistinguishable, except with negligible probability.*

Proof. We analyze the cases in which G_{10} and G_{11} each output $\perp$ in response to a W.

G_{10} runs $\mathcal{E}_{\text{SKE}}.\mathit{Dec}(K_D, W)$ to get either $\perp$ or a tuple X, which it parses as $(ind, lpos, num, len, f_1, f_{2,1}, \ldots, f_{2,d})$. G_{10} outputs $\perp$ if any of the following events occur:

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

G_{11} 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 $L.1'$) W is not a ciphertext in D,
- (Event $L.2'$) W is a ciphertext in D but not for any prefix of p. That is, $W = D(\kappa)$ where κ is not equal to $R_1(p[1..i])$ for any i.
- (Event $L.3'$) 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_{11} outputs $\perp$ in response to W for any query, then G_{10} also outputs $\perp$ except with negligible probability, as we already showed by ciphertext integrity of $\mathcal{E}_{\text{SKE}}$ in Lemma C.2 in the proof of correctness of $\mathcal{E}_{\text{PM}}$ against malicious adversaries.

If G_{10} outputs $\perp$, then G_{11} also outputs $\perp$, since if W is the ciphertext in $D(R_1(p[1..i]))$, then W will decrypt successfully, with $f_1 = R_1(p[1..len])$, and $\mathcal{E}_{\text{SKE}}.\textit{Dec}(f_{2,k}, T_j) = \perp$ for all $k \in \{1, \dots, d\}, j > i$.

Thus, G_{10} and G_{11} are indistinguishable except with negligible probability. □

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 $\mathcal{E}_{\text{SKE}}.\textit{Enc}(K_D, 0)$ instead of $\mathcal{E}_{\text{SKE}}.\textit{Enc}(K_D, X_u)$.

A hybrid argument shows that G_{12} and G_{13} are indistinguishable by the CPA security of $\mathcal{E}_{\text{SKE}}$.

Lemma D.12. *If $\mathcal{E}_{\text{SKE}}$ is a CPA-secure encryption scheme, then G_{12} and G_{13} are indistinguishable, except with negligible probability.*

The proof is omitted since it is nearly identical to the proof for G_4 and G_5.

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 $T_i \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Enc}(r, R_1(p[1..i]))$; otherwise, it sets $T_i \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Enc}(R_2(p[1..i]), R_1(p[1..i]))$ as in game G_{13}.

For any k and i, let p_k denote the kth query, and let $T_{k,i}$ denote the T_i produced by the simulator for the kth 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 hybrid argument shows that G_{13} and G_{14} are indistinguishable by the which-key-concealing property of $\mathcal{E}_{\text{SKE}}$.

Lemma D.13. *If $\mathcal{E}_{\text{SKE}}$ is a which-key-concealing encryption scheme, then games G_{13} and G_{14} are indistinguishable, except with negligible probability.*

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish G_{13} and G_{14}. Let q_{max} be an upper bound on the number of queries $\mathcal{A}$ chooses, and let m_{max} be an upper bound on the length of $\mathcal{A}$'s queries, where q_{max} and m_{max} are polynomial in λ.

Consider the following sequence of $q_{max}(m_{max}+1)$ hybrid games. For each $i \in \{0, \dots, m_{max}\}, j \in \{1, \dots, q_{max}\}$, game $H_{i,j}$ is the same as G_{13}, with the following exceptions.

- For $j' < j$, for each i', if $p_{j'}[1..i']$ is non-matching, choose a fresh random value r and set $T_{j',i'} \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Enc}(r, R_1(p_{j'}[1..i]))$, as in game G_{14}.
- For the jth query p_j, for $i' \leq i$, if $p_j[1..i']$ is non-matching, again choose a fresh random value r and set $T_{j,i'} \leftarrow \mathcal{E}_{\text{SKE}}.\textit{Enc}(r, R_1(p_j[1..i']))$, as in game G_{14}.

Note that $H_{0,1} = G_{13}$ and $H_{m_{max},q_{max}} = G_{14}$.

Now, we argue that if there is an adversary $\mathcal{A}$ that can distinguish $H_{i-1,j}$ from $H_{i,j}$ for any $i \in \{1, \ldots, m_{max}\}, j \in \{1, \ldots, q_{max}\}$, then we can construct an algorithm $\mathcal{B}$ that attacks the which-key-concealing property of $\mathcal{E}_{\text{SKE}}$ with the same advantage.

$\mathcal{B}$ will act as the simulator in $H_{i-1,j}$, with the following exception. If $p_j[1..i]$ is non-matching, $\mathcal{B}$ first queries its left encryption oracle on $R_1(p_j[1..i])$ and sets $T_{j,i}$ to the resulting ciphertext. $\mathcal{B}$ then remembers $p_j[1..i]$, and for any later queries $p_{j'}$ that share the prefix $p_j[1..i]$, $\mathcal{B}$ queries its right oracle on $R_1(p_j[1..i])$ and uses the resulting ciphertext as $T_{j',i}$. Otherwise, $\mathcal{B}$ proceeds as in $H_{i-1,j}$.

Now, if both of $\mathcal{B}$'s encryption oracles are for the same key, then $T_{j,i}$ and the $T_{j',i}$ for all future queries p'_j that share the prefix $p_j[1..i]$ will be encrypted under the same random key, and $\mathcal{A}$'s view will be the same as in $H_{i-1,j}$. On the other hand, if the two encryption oracles are for different keys, then $T_{j,i}$ will have been generated using a different random key from that used to generate $T_{j',i}$ for all future queries $p_{j'}$ that share the prefix $p_j[1..i]$, and $\mathcal{A}$'s view will be the same as in $H_{i,j}$.

Note that if $p_j[1..i]$ is a matching prefix, so $\mathcal{B}$ does not output a challenge, then $H_{i-1,j}$ and $H_{i,j}$ are identical, so $\mathcal{A}$'s view is the same as in both $H_{i-1,j}$ and $H_{i,j}$. Thus, $H_{i-1,j}$ and $H_{i,j}$ are indistinguishable by the key hiding property of $\mathcal{E}_{\text{SKE}}$.

We can show by a very similar reduction that games $H_{m_{max},j}$ and $H_{1,j+1}$ are indistinguishable. Since there are a polynomial number of hybrid games, we conclude then that games $H_{0,1} = G_{13}$ and $H_{m_{max},q_{max}} = G_{14}$ are indistinguishable. □

Game G_{15}**.** This is the same as game G_{14}, except that in the query protocol for any pattern 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 $p[1..i]$, $i = 1, \ldots, m$, if $p[1..i]$ is non-matching, the simulator chooses a fresh random value r and sets $T_i \leftarrow \mathcal{E}_{\text{SKE}}.\mathit{Enc}(r, 0)$; otherwise, it sets $T_i \leftarrow \mathcal{E}_{\text{SKE}}.\mathit{Enc}(r, R_1(p[1..i]))$ 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_{14}, $T_{j,i}$ will be an encryption of 0, while in G_{15}, $T_{j,i}$ 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 hybrid argument shows that games G_{14} and G_{15} are indistinguishable by the CPA security of $\mathcal{E}_{\text{SKE}}$.

Lemma D.14. *If $\mathcal{E}_{\text{SKE}}$ is a CPA-secure encryption scheme, then games G_{14} and G_{15} are indistinguishable, except with negligible probability.*

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish G_{14} and G_{15}. Let q_{max} be an upper bound on the number of queries that $\mathcal{A}$ chooses, and let m_{max} be an upper bound on the length of $\mathcal{A}$'s queries, where q_{max} and m_{max} are polynomial in λ.

We consider a sequence of $q_{max}(m_{max} + 1)$ hybrid games. For each
$i \in \{0, \ldots, m_{max}\}, j \in \{1, \ldots, q_{max}\}$, game $H_{i,j}$ is the same as G_{14}, with the following exceptions.

- For $j' < j$, for each i', if $p_{j'}[1..i']$ is non-matching, choose a fresh random value r and set $T_{j',i'} \leftarrow \mathcal{E}_{\text{SKE}}.\mathit{Enc}(r, 0)$, as in game G_{15}.
- For the jth query p_j, for $i' \leq i$, if $p_j[1..i']$ is non-matching, again choose a fresh random value r and set $T_{j,i} \leftarrow \mathcal{E}_{\text{SKE}}.\mathit{Enc}(r, 0)$ as in game G_{15}.

Note that $H_{0,1} = G_{14}$ and $H_{m_{max},q_{max}} = G_{15}$.

Now, we argue that if there is an adversary $\mathcal{A}$ that can distinguish $H_{i-1,j}$ from $H_{i,j}$ for any $i \in \{1, \ldots, m_{max}\}, j \in \{1, \ldots, q_{max}\}$, then we can construct an algorithm $\mathcal{B}$ that attacks the CPA security of $\mathcal{E}_{\text{SKE}}$ with the same advantage.

$\mathcal{B}$ acts as the simulator in $H_{i-1,j}$, with the following exception. If $p_j[1..i]$ is non-matching, it chooses a fresh random value r and outputs $r, 0$ as its challenge in the CPA-security game, and sets $T_{i,j}$ to be the resulting ciphertext. Otherwise, $\mathcal{B}$ proceeds as in $H_{i-1,j}$. Note that in both $H_{i-1,j}$ and $H_{i,j}$, the random key r is used to encrypt only one ciphertext.

Now, if the CPA-security challenger gave $\mathcal{B}$ an encryption of r, then $\mathcal{A}$'s view will be the same as in $H_{i-1,j}$. On the other hand if the CPA-security challenger returned an encryption of 0, then $\mathcal{A}$'s view will be the same as in $H_{i,j}$.

Note that if $p_j[1..i]$ is a matching prefix, so $\mathcal{B}$ does not produce a challenge, then $H_{i-1,j}$ and $H_{i,j}$ are identical, so $\mathcal{A}$'s view is the same as in both $H_{i-1,j}$ and $H_{i,j}$. Thus, $H_{i-1,j}$ and $H_{i,j}$ are indistinguishable by the CPA security of $\mathcal{E}_{\text{SKE}}$.

We can show by a very similar reduction that games $H_{m_{max},j}$ and $H_{1,j+1}$ are indistinguishable. Since there are a polynomial number of hybrid games, we conclude then that games $H_{0,1} = G_{14}$ and $H_{m_{max},q_{max}} = G_{15}$ are indistinguishable. □

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 $T_1, \ldots, T_m$. 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 $\hat{\rho}(u)$ strings used in D are unique, so $\mathcal{S}$ does not need to ensure consistency between any of the random values. Then, 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 $\text{QP}(s, p_1, \ldots, p_j)$. 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. However, both simulators produce identical views.

□