Università degli Studi di Padova

Dipartimento Matematica

Master Thesis in Cybersecurity

Fuzzy Searchable Symmetric Encryption: Design and Implementation of a Novel Scheme Toward Real-World Applications

Master Candidate

Simone Ragusa

Supervisor

Prof. Nicola Laurenti

Academic Year
2024/2025

Thinking doesn't guarantee that we won't make mistakes.
But not thinking guarantees that we will.

— Leslie Lamport

Abstract

Most services and applications employ cloud-based solutions to store user data. To provide users with the ability to perform search queries, these services typically encrypt the data with keys known to the server, allowing search results to be resolved directly on the server, which reduces the load on user devices. However, this approach is clearly not suitable when the application server is untrusted. Searchable symmetric encryption (SSE) aims at enabling searches over an encrypted database stored on an untrusted server while preserving the privacy of both queries and data. Unfortunately, security concerns, poor usability and flexibility, and lack of easy-to-use implementations are preventing the adoption of SSE schemes by real-world applications.

This work presents Emys, a novel dynamic searchable symmetric encryption (DSSE) scheme offering full-text fuzzy search capabilities with tunable typo tolerance, making it adaptable to diverse contexts. The threat model considered includes malicious servers that attempt not only to learn information, but also to tamper with it. Therefore, the scheme provides verifiability of search results, allowing the detection of misbehaving servers and the immediate rejection of altered responses in such cases. Additionally, Emys provides both forward privacy and Type-I_B backward privacy. The latter is a new type of backward privacy introduced in this work, formulated by generalizing and refining similar notions from the literature. To complement the DSSE scheme, this work also describes a file handling scheme as a drop-in replaceable mechanism to manage the actual data.

To encourage consistent and accessible SSE implementations, we present a general programming interface that is flexible enough to accommodate a wide variety of schemes. Moreover, the interface promotes the handling of subtle details and complexities within implementations themselves, exposing only essential functionality to enable straightforward adoption by software developers. Finally, we implement the Emys DSSE scheme, conforming to the interface. Despite lacking several optimizations, the benchmark results suggest that the current implementation is already usable for small to medium-sized systems.

Acknowledgments

I want to thank my supervisor, Prof. Nicola Laurenti, for his trust and support throughout the work on this thesis. I sincerely appreciate the independence he granted me, which made the whole process both rewarding and enjoyable.

I am deeply grateful to Filippo Valsorda, whose contributions to the cryptography community and online writings sparked my passion for cryptography a few years ago and continue to inspire me. I am also thankful to Mike Rosulek for authoring *The Joy of Cryptography*, which has been an invaluable resource for me to learn the fundamentals of provable security.

My warmest thanks to all my friends who have always supported and encouraged me. It wouldn't be the same without you. Special thanks to Elsa — I would never have written this thesis if you hadn't convinced me I was capable of taking this degree path.

Finally, I'd like to thank my family for always being there and taking care of me. I notice all the little things.

Contents

List of Acronyms and Initialisms

AAD	Additional Authenticated Data
AD	Associated Data
AEAD	Authenticated Encryption with Associated Data
ASE	Asymmetric Searchable Encryption
CKA2	Adaptive Dynamic Chosen-Keyword Attack
CPU	Central Processing Unit
CQA2	Adaptive Dynamic Chosen-Query Attack
DBMS	Database Management System
DSSE	Dynamic Searchable Symmetric Encryption
FHE	Fully Homomorphic Encryption
FIPS	Federal Information Processing Standard
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
KDF	Key Derivation Function
MAC	Message Authentication Code
MIT	Massachusetts Institute of Technology
MPC	Multi-Party Computation
OPE	Order-Preserving Encryption
ORAM	Oblivious RAM
ORE	Order-Revealing Encryption
OTP	One-Time Pad
PAE	Pre-Authentication Encoding
PASETO	Platform-Agnostic Security Tokens
PPE	Property-preserving encryption

PRF	Pseudorandom Function
PRO	Programmable Random Oracle
RAM	Random-Access Memory
RO	Random Oracle
ROM	Random Oracle Model
SE	Searchable Encryption
SKE	Symmetric-Key Encryption
SQL	Structured Query Language
SSE	Searchable Symmetric Encryption
TLS	Transport Layer Security
UHF	Universal Hash Function
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
XOF	Extendable-Output Function
XOR	Exclusive Or

1 Introduction

Cloud services continue to gain popularity due to the economic benefits they offer, along with the powerful capabilities of cloud computing and the convenience of cloud storage solutions. This is a widely acknowledged situation, as the drawbacks of local storage and computing are evident, and maintaining powerful hardware can be challenging for both individuals and small to medium-sized businesses. However, outsourcing data to the cloud results in a loss of control over the data for users, raising reasonable concerns about associated privacy and confidentiality issues. The classic solution to this problem is encrypting the data before outsourcing it. While this prevents cloud service providers from accessing sensitive information, it also reduces the usability of the data to the point that it is often impossible to work with it. A naive solution would be to authorize the cloud server with the secret key, let it decrypt the data, perform the necessary computations, and then re-encrypt the data once finished. Clearly, this destroys every chance of data privacy. Another solution would be to have the server return all the data and let the user decrypt and operate on it, though this defeats most of the benefits of using cloud services.

File hosting and email services are among the most commonly used by individual users, think about Google Drive, Dropbox, and Gmail. To allow for searching capabilities within these services, providers only encrypt data in transit and at rest [Dro25, Goo21]. This approach usually protects data privacy from being broken following data breaches. However, the providers themselves do have full access to the plaintext data, which is unsatisfactory. In general, this corresponds to the naive solution outlined above. As a consequence, an approach that allows users to perform searches on encrypted data is desirable.

On one side of the spectrum we have fully homomorphic encryption (FHE), multi-party computation (MPC), oblivious RAM (ORAM), and oblivious data structures. Fully homomorphic encryption is a form of encryption that allows arbitrary computations over encrypted data [Gen09], including support for any type of search query. Multi-party computation protocols enable a group to jointly perform a computation while keeping any participant's inputs private [EKR22]. Oblivious RAM algorithms interface with a physical CPU and RAM in such a way that the information about the actual memory accesses is hidden

[GO96]. Similarly, oblivious data structures hide the pattern of operations applied to them and only return the final result of a procedure. All these techniques give almost perfect security guarantees, and they would be great candidates to implement search on encrypted data. Unfortunately, current schemes are inefficient and impose a significant overhead on computation costs, communication costs, or both [HS21, LN18].

On the other side of the spectrum we have deterministic encryption, order-preserving encryption (OPE), order-revealing encryption (ORE), and property-preserving encryption (PPE). These forms of encryption allow obtaining some limited information about the plaintext without fully revealing the ciphertext, with varying degree and type of leakage. Some of these techniques, such as deterministic encryption, allow for trivial implementations of encrypted search, and all of them have exhibit good performance in practice. Unfortunately, the security guarantees are not as good as those of well-known probabilistic encryption schemes.

A middle ground that stands in between is searchable encryption (SE), where cryptographers make trade-offs between efficiency and security. Searchable encryption is a cryptographic mechanism that enables users to search over encrypted data while preserving the privacy of the search queries and the data itself. As with traditional encryption, we can distinguish between symmetric mechanisms, under the domain of searchable symmetric encryption (SSE), and asymmetric mechanisms, denoted as asymmetric searchable encryption (ASE). This work focuses on searchable symmetric encryption.

1.1 Searchable Symmetric Encryption

The field of searchable symmetric encryption (SSE) begins with the seminal paper by Song, Wagner, and Perrig [SWP00], which is the first work to explicitly consider the problem of searching on encrypted data. Later, in 2006, Curtmola et al. gave the first formal definitions, introducing the notion of adaptive security with leakage, and proposed the first practical solutions [Cur+06].

SSE schemes can be categorized by their dynamism and by the query types they support. Static schemes parse all files of a dataset and build the necessary structures during an initial setup phase, after which it is not possible to add or remove any file. On the other hand, dynamic schemes allow adding new files, and removing and modifying existing ones [KPR12]. This also makes it possible to parse a large dataset of files gradually, which helps, for example, to deal with network bandwidth restrictions.

As for query type support, most existing SSE schemes found in the literature focus on single-keyword search [Bos16, BMO17, Zha+25]. This query type allows for exact matching of a single keyword within the content of the outsourced files, thus being fairly restrictive. Other query types are:

- Conjunctive queries, which allow for exact matching of multiple keywords in a single query [Sun+15]. All keywords must be present for the file to be in the result set.
- Boolean queries, allowing for any combination of conjunctions, disjunctions, and negations of keywords in the same query [MS13].
- Range queries, useful for sortable data (e.g., calendar dates, age of people), allows for retrieving data within a certain interval [Zuo+18].
- Ranked search, which allows ranking the search results according to a certain relevance criteria (e.g., frequency of keywords) [Wan+12].
- Substring queries, allowing for partial matching of a single keyword [Moa+18].
- Fuzzy queries, performing approximate string matching, which can search for keywords while tolerating minor typos [Ton+23].

Semantic search is also possible, although there is not much research on it, and it is a challenging problem to create such schemes with adequate space and time complexities.

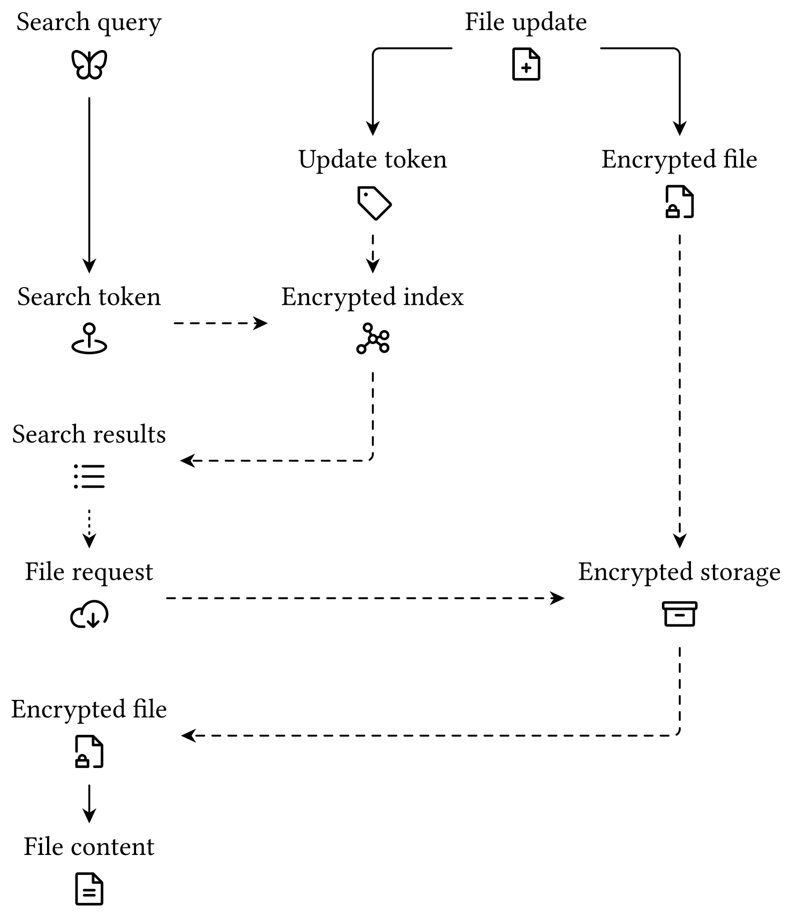

Figure 1.1: *System model of dynamic SSE. Solid edges indicate operations executed locally on the client, while dashed edges indicate communications to and from the server. The dotted edge signifies that the client may submit a file request regardless of search results.*

Figure 1.1 shows the high-level system model of dynamic searchable symmetric encryption. The server stores an encrypted index, used for performing searches, and the actual file contents in encrypted form. The client generates update tokens and search tokens, then sends them to the server to update the encrypted index and request the list of files satisfying a search query, respectively.

The notion of security for SSE schemes considers the amount of leakage of the scheme itself. The two most common leakage types are the search pattern and the access pattern. The former captures the ability of an adversary to recognize repeated search queries, while the latter reveals to the adversary the results of a specific query (usually the file identifiers of the matching files). Moreover, dynamic schemes also require two additional security notions, termed forward privacy and backward privacy.

Most SSE schemes consider an honest-but-curious cloud server as the adversary in their threat model. This means that the server will correctly follow the protocol, though it might try to gather some information about the file contents or the search queries. Stronger assumptions, such as a malicious server, require schemes to have verifiable search results [BFP16].

1.2 Current Issues and Research Objectives

Despite over 20 years of research on searchable encryption, only a limited number of real-world applications currently exist [GPT23]. In 2011, a group from the MIT built CryptDB for applications backed by SQL databases [Pop+11]. Later, in 2014, a paper presented Blind Seer, a database management system (DBMS) supporting rich yet private search queries [Pap+14]. More recently, MongoDB, a document-oriented database, introduced Queryable Encryption to their product [Mon23]. These systems provide encrypted search capabilities for products that will be used by developers or system administrators.

In 2018, Seny Kamara lead the development of Pixek, an end-to-end encrypted camera application that allows searching for photos using keywords [Wir18]. After 7 years, the project seems to be dead. Therefore, to the best of our knowledge, no real-world application with a direct impact on individual users exists today.

We have identified three main issues that contribute to the current state of the situation.

1. There is a general lack of confidence in the security of SSE schemes. In particular, the security notion allows for some amount of well-defined information to be leaked by the schemes, which seems to consistently lead to serious security breaches when applying cryptanalysis techniques [Dam+25]. Moreover, the honest-but-curious threat model may not be suitable for some applications (e.g., legal document storage, personal finance, healthcare records).
2. Most SSE schemes provide poor usability by either being static or allowing only single-keyword searches. Further, almost no design takes into account

the actual management of files, treating them only as unique identifiers, and leaving the handling of their content unaddressed.

3. There are no software libraries implementing SSE that developers can immediately use without extensive knowledge of the topic.

Contributions. The goal of this work is to narrow the gap between searchable symmetric encryption research and real-world applications for individual users. To attain such an objective, we design a novel SSE scheme that is both flexible, supporting full-text fuzzy search across multiple textual files, and provably secure, limiting the amount of leakage and offering search result verification. Additionally, we propose a general programming interface for SSE schemes that implementors can use. Finally, we implement our design, adhering to the proposed interface, to pave the way for the adoption of SSE schemes in applications and encouraging more usable implementations.

Organization. The rest of this thesis is structured as follows. In Chapter 2, we introduce the notation, provide background and formal definitions for searchable symmetric encryption schemes, and define some useful cryptographic primitives. Then, we also introduce two new cryptographic constructions. In Chapter 3, we present our novel SSE scheme, an accompanying file handling scheme, and the key hierarchy. Section 3.3 details the formal security proofs of the SSE scheme. In Chapter 4, we present a programming interface suitable for a wide variety of scheme designs. Then, we implement and benchmark our scheme. Finally, in Chapter 5, we discuss the results of this work, its limitations, and future improvements.

2 Preliminaries

2.1 Notation

Mathematical Symbols. We write the set of integers as

$$\mathbb{Z} \overset{\text{def}}{=} \{\dots, -2, -1, 0, 1, 2, \dots\},$$

the set of natural numbers (i.e., nonnegative integers) as

$$\mathbb{N} \overset{\text{def}}{=} \{0, 1, 2, \dots\},$$

and the set of positive natural numbers as

$$\mathbb{N}^+ \overset{\text{def}}{=} \{1, 2, \dots\}.$$

For $a, b \in \mathbb{Z}$, we say that a divides b, and write $a \mid b$, if $\exists k \in \mathbb{Z}$ such that $b = ka$. For $n \in \mathbb{N}^+$, we denote the set of integers modulo n as

$$\mathbb{Z}_n \overset{\text{def}}{=} \{0, \dots, n-1\},$$

and we say that two integers a and b are congruent modulo n, and write $a \equiv_n b$, if $n \mid (a - b)$. Moreover, we define the modulo operation written $a \bmod n$ as the unique $r \in \mathbb{Z}_n$ such that $n \mid (a - r)$.

For a non-integer number x, we define the floor of x and the ceiling of x, respectively, as the integers

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\},$$
$$\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}.$$

For any two numbers a, b, we write $a \approx b$ to denote approximate equality between a and b.

If S is a set, we write $|S|$ to denote its cardinality, i.e., the number of elements it contains. Also, we write S^n to denote the set of n-tuples of elements from the set S, namely,

$$S^n = \underbrace{S \times S \times \cdots \times S}_{n \text{ times}},$$

and we write S^* to denote the set of all finite-length tuples of elements from the set S. The expression $\mathcal{P}(S)$ denotes the power set of a set S, i.e., the set of all subsets of S, including the empty set $\emptyset$ and S itself. For a finite set S of cardinality n, we define the order statistics of S as

$$s_{(1)} \leq s_{(2)} \leq \cdots \leq s_{(n)},$$

where each $s_{(i)}$, for $1 \leq i \leq n$, is an element of the set S. In particular, we have $s_{(1)} = \min S$, $s_{(n)} = \max S$, and $s_{(i)}$ is the i-th smallest element overall.

If $f : \mathcal{X} \to \mathcal{Y}$ is a function, we write $f(x) = \bot$ to denote the undefined nature of f, suggesting that evaluating f is meaningless. We say that a function f is negligible if, for every polynomial p, we have

$$\lim_{x \to \infty} p(x) f(x) = 0.$$

If $f, g : \mathcal{X} \to \mathcal{Y}$ are two functions, we write $f \approx g$ to mean that $|f(x) - g(x)|$ is a negligible function. As long as it is applied a polynomial number of times, this use of the symbol $\approx$ is transitive.

Probability. We say that a probability distribution over a set of outcomes X assigns a probability $\Pr[x]$ to each outcome x via the probability function Pr. For each outcome, it must be that $0 \leq \Pr[x] \leq 1$. Moreover, the sum of all probabilities must satisfy

$$\sum_{x \in X} \Pr[x] = 1.$$

We call uniform distribution the special probability distribution in which every outcome $x \in X$ is assigned probability

$$\Pr[x] = \frac{1}{|X|}.$$

We refer to a collection of outcomes as an event, and we extend the notation of probability to events. Specifically, for an event A, we define

$$\Pr[A] = \sum_{x \in A} \Pr[x].$$

Strings. We use a red colored monospaced typeface to indicate strings of bits and special values, such as error values. We write $\{\texttt{0}, \texttt{1}\}^n$ to denote the set of n-bit binary strings, and $\{\texttt{0}, \texttt{1}\}^*$ to denote the set of all finite-length binary strings. We write $\texttt{0}^n$ and $\texttt{1}^n$ to denote strings of n zeros and n ones, respectively.

If b is a binary string, we write $|b|$ to denote its length in bits, we write b_i to refer to its i-th bit, and we write $b[i : j]$ to denote the substring of b from bit i included and bit j excluded.

We write $x \| y$ to denote the result of concatenating two strings x and y. When x and y are strings of the same length, we write $x \oplus y$ to denote the bitwise exclusive-or of the two strings.

Pseudocode. When $\mathcal{D}$ is a probability distribution, we write $x \twoheadleftarrow \mathcal{D}$ to denote sampling x according to the distribution $\mathcal{D}$. When S is a finite set, we write $x \twoheadleftarrow S$ to denote sampling x from the uniform distribution over S.

We write $x \coloneqq y$ to denote the assignment of the value of expression y to variable x. Note that the result of expression y must be deterministic, otherwise the $\twoheadleftarrow$ symbol should be used.

We write comparisons as $==$ (as in many programming languages). The expression $x == y$ returns either `true` or `false`, and modifies neither of the two variables (or expressions). Other boolean operators we use are $\neq$, $\in$, and $\notin$.

If T is an array or an associative array (i.e., a key-value store), we write $|T|$ to denote the number of elements in T and $T[i]$ to denote the i-th element or the element corresponding to key i, respectively. When T is an associative array, the expression "$T[x]$ undefined" returns either `true` or `false`, indicating whether the element corresponding to key x exists in the array or not.

We use the $-$ symbol to indicate an ignored value on the left-hand side of an assignment or sampling expression, and we use the $\perp$ symbol to indicate an empty value on the right-hand side of a similar expression.

Security games. We write security definitions and proofs using code-based games in the style of *The Joy Of Cryptography* by Mike Rosulek (draft of January 3, 2021, at the time of writing) [Ros21], with minor notation changes. We reproduce here the most relevant definitions.

Definition 2.1 (Libraries) *A **library** $\mathcal{L}$ is a collection of subroutines and private, static variables. A library's **interface** consists of the names, argument types, and output types of all its subroutines. If a program $\mathcal{A}$ includes calls to subroutines in the interface of $\mathcal{L}$, then we write $\mathcal{A} \diamond \mathcal{L}$ to denote the result of **linking** $\mathcal{A}$ to $\mathcal{L}$ in the natural way (i.e., answering those subroutine calls using the implementation specified in $\mathcal{L}$). We write $\mathcal{A} \diamond \mathcal{L} \Rightarrow z$ to denote the event that program $\mathcal{A} \diamond \mathcal{L}$ outputs the value z.*

Definition 2.2 (Interchangeable) *Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be two libraries that have the same interface. We say that $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are **interchangeable**, and write $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$, if for all programs $\mathcal{A}$ that output a boolean value,*

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \texttt{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \texttt{true}].$$

Definition 2.3 (Indistinguishable) *Let $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ be two libraries that have the same interface. We say that $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ are **indistinguishable**, and write $\mathcal{L}_{\text{left}} \approxeq \mathcal{L}_{\text{right}}$, if for all polynomial-time programs $\mathcal{A}$ that output a boolean value,*

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \texttt{true}] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \texttt{true}].$$

We call the quantity

$$\text{Adv}_{\mathcal{A}} = \left|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \texttt{true}] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \texttt{true}]\right|$$

*the **advantage** of $\mathcal{A}$ in distinguishing $\mathcal{L}_{\text{left}}$ from $\mathcal{L}_{\text{right}}$. Two libraries are therefore indistinguishable if all polynomial-time calling programs have a negligible advantage in distinguishing them.*

Lemma 2.4 ($\approxeq$ properties) *If $\mathcal{L}_1 \equiv \mathcal{L}_2$, then $\mathcal{L}_1 \approxeq \mathcal{L}_2$. Also, from the transitivity of the $\approx$ symbol for functions, if $\mathcal{L}_1 \approxeq \mathcal{L}_2 \approxeq \mathcal{L}_3$, then $\mathcal{L}_1 \approxeq \mathcal{L}_3$.*

Lemma 2.5 (Chaining) *If $\mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\text{right}}$, then, for any library $\mathcal{L}_{\star}$, we have $\mathcal{L}_{\star} \diamond \mathcal{L}_{\text{left}} \equiv \mathcal{L}_{\star} \diamond \mathcal{L}_{\text{right}}$. Similarly, if $\mathcal{L}_{\text{left}} \approxeq \mathcal{L}_{\text{right}}$, then, for any polynomial-time library $\mathcal{L}_{\star}$, we have $\mathcal{L}_{\star} \diamond \mathcal{L}_{\text{left}} \approxeq \mathcal{L}_{\star} \diamond \mathcal{L}_{\text{right}}$.*

2.2 Searchable Symmetric Encryption Definitions

From this point forward, this work exclusively considers dynamic schemes. Accordingly, the following definitions pertain to dynamic searchable symmetric encryption.

Definition 2.6 (DSSE scheme) *For a key space $\mathcal{K}$, a file identifier space $\mathcal{I}$, a symbol space Ω, a query space $\mathcal{Q} \subseteq \{q \mid q : \mathcal{P}(\Omega) \to \{\mathtt{true}, \mathtt{false}\}\}$ and the update operation space $\mathcal{O} = \{\mathtt{add}, \mathtt{del}\}$, a **dynamic searchable symmetric encryption (DSSE) scheme** consists of the following algorithms:*

- KeyGen*: a randomized algorithm run by the client that outputs a key $k \in \mathcal{K}$.*
- SearchToken*: a (possibly randomized) algorithm run by the client that takes a key $k \in \mathcal{K}$ and a query $q \in \mathcal{Q}$ as input, and outputs a search token $\tau_s \in \{0,1\}^{\ell_s}$.*
- Search*: a deterministic algorithm run by the server that takes a search token $\tau_s \in \{0,1\}^{\ell_s}$ as input, and outputs a search result $r \in \{0,1\}^{\ell_r}$.*
- SearchResult*: a deterministic algorithm run by the client that takes a key $k \in \mathcal{K}$, a query $q \in \mathcal{Q}$ and a search result $r \in \{0,1\}^{\ell_r}$ as input, and outputs a finite set of file identifiers $R \subseteq \mathcal{I}$ or $\mathtt{err}$ if search result verification failed.*
- UpdateToken*: a (possibly randomized) algorithm run by the client that takes a key $k \in \mathcal{K}$, a file identifier $id \in \mathcal{I}$, a symbol $\omega \in \Omega$ and an update operation $op \in \mathcal{O}$ as input, and outputs an update token $\tau_u \in \{0,1\}^{\ell_u}$.*
- Update*: a deterministic algorithm run by the server that takes an update token $\tau_u \in \{0,1\}^{\ell_u}$ as input, and outputs nothing.*

In terms of splitting functionalities into different algorithms, the above definition of dynamic searchable symmetric encryption (DSSE) sits in between the widely used protocol-like definition [Bos16, Li+24, Zuo+21] and the original definition given by Kamara et al. [KPR12]. Recently, Le et al. took a similar approach [LH25].

There are several reasons why we opted to arrange the definition in this way, introducing some modifications:

- A greater separation of concerns makes it easier to write security proofs and translate schemes from design to implementation.
- Unifying all update operations (unlike in [KPR12], where there are separate AddToken and DelToken algorithms) forces designers to think about strategies that avoid leaking the update operation.

- The SearchResult algorithm allows schemes to optionally support search result verification without introducing an additional algorithm, unlike in the work of Zhao et al. [Zha+25] and many others.
- Abstracting away the actual management of files allows designers to focus on building secure index structures, dealing almost exclusively with file identifiers. Nonetheless, real-world applications deal with actual files. Therefore, bridging the gap between SSE and secure storage is crucial, and we provide a solution in Section 3.2.2.
- Single updates generalize naturally to batch updates, sending multiple update tokens to the server at a time, which will then execute the Update algorithm repeatedly.

Note that users of the schemes are responsible for properly handling file identifiers and ensuring that they uniquely correspond to different files.

We denote by $\mathsf{FS} = \{f_0, \dots, f_{\ell-1}\}$ the collection of files handled by a DSSE scheme Σ, and $\mathsf{FS}(q) \subseteq \mathsf{FS}$ the subset of files in the collection that satisfy the search query q. Similarly, we call $\mathsf{FI} \in \Sigma.\mathcal{I}^{\ell}$ the set of file identifiers in the collection, and $\mathsf{FI}(q) \subseteq \mathsf{FI}$ the subset of file identifiers corresponding to files that satisfy the search query q.

Intuitively, a DSSE scheme is correct if the result set R is correct for every search query q, namely $R = \mathsf{FI}(q)$. In the formal definition below, we require the probability that the equality holds to be almost always equal to 1. In particular, the definition is written in terms of a probability because SearchToken is allowed to be a randomized algorithm. Moreover, we require the result to be correct only with an overwhelming probability to allow for search pattern protection techniques that may negatively affect the result with a negligible probability.

Definition 2.7 (DSSE correctness) *A DSSE scheme Σ satisfies **correctness** if for all $k \in \Sigma.\mathcal{K}$ and all $q \in \Sigma.\mathcal{Q}$,*

$$\Pr[\mathsf{SearchResult}(k, q, \mathsf{Search}(\mathsf{SearchToken}(k, q))) = \mathsf{FI}(q)] \approx 1$$

after any number of calls to Σ.UpdateToken and Σ.Update. In presence of active malicious actions, the scheme satisfies correctness if,

$$\Pr[\mathsf{SearchResult}(k, q, \mathsf{Search}(\mathsf{SearchToken}(k, q))) = \mathtt{err}] \approx 1.$$

In the following, we provide the definitions of adaptive security and verifiability for a DSSE scheme. We renamed the security notion from adaptive dynamic chosen-keyword attack (CKA2), as originally defined by Kamara et al. [KPR12], to adaptive dynamic chosen-query attack (CQA2) to emphasize the generalization to any type of search query. As in Definition 2.6 there is no Setup algorithm, the definition below does not consider a setup leakage function $\mathcal{L}_{\mathsf{Setup}}$ as found in the literature. The reason is that a setup can be performed lazily via updates.

Definition 2.8 (CQA2 security) *Let Σ be a DSSE scheme. We say that Σ has $(\mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}})$-**security against adaptive dynamic chosen-query attacks (CQA2 security)**, for leakage func-*

tions $\mathcal{L}_{\text{Search}}$ and $\mathcal{L}_{\text{Update}}$ and a stateful simulator Π, if $\mathcal{L}^{\Sigma}_{\text{cqa2-real}} \approxeq \mathcal{L}^{\Sigma,\Pi}_{\text{cqa2-rand}}$, where:

```
$\mathcal{L}^{\Sigma}_{\text{cqa2-real}}$
$k \twoheadleftarrow \Sigma.\text{KeyGen}$

CQA.SEARCHTOKEN($q \in \Sigma.\mathcal{Q}$):
    $\tau_s \twoheadleftarrow \Sigma.\text{SearchToken}(k, q)$
    return $\tau_s$

CQA.UPDATETOKEN($id \in \Sigma.\mathcal{I}, \omega \in \Sigma.\Omega, op \in \Sigma.\mathcal{O}$):
    $\tau_u \twoheadleftarrow \Sigma.\text{UpdateToken}(k, id, \omega, op)$
    return $\tau_u$
```

```
$\mathcal{L}^{\Sigma,\Pi}_{\text{cqa2-rand}}$
CQA.SEARCHTOKEN($q \in \Sigma.\mathcal{Q}$):
    $\tau_s \twoheadleftarrow \Pi.\text{SimulateSearchToken}(\mathcal{L}_{\text{Search}}(q))$
    return $\tau_s$

CQA.UPDATETOKEN($id \in \Sigma.\mathcal{I}, \omega \in \Sigma.\Omega, op \in \Sigma.\mathcal{O}$):
    $\tau_u \twoheadleftarrow \Pi.\text{SimulateUpdateToken}(\mathcal{L}_{\text{Update}}(id, \omega, op))$
    return $\tau_u$
```

The definition of verifiability provided below tests a client's ability to detect whether a server is not following the protocol correctly. In particular, in the ideal world, the client keeps a local copy of the server state and performs honest operations on it.

Definition 2.9 (Verifiability) *Let Σ be a DSSE scheme. We say that Σ has **verifiability** if $\mathcal{L}^{\Sigma}_{\text{verif-real}} \approxeq \mathcal{L}^{\Sigma}_{\text{verif-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{verif-real}}$
$k \twoheadleftarrow \Sigma.\text{KeyGen}$

VERIF.SEARCHTOKEN($q \in \Sigma.\mathcal{Q}$):
    $\tau_s \twoheadleftarrow \Sigma.\text{SearchToken}(k, q)$
    return $\tau_s$

VERIF.UPDATETOKEN($id \in \Sigma.\mathcal{I}, \omega \in \Sigma.\Omega, op \in \Sigma.\mathcal{O}$):
    $\tau_u \twoheadleftarrow \Sigma.\text{UpdateToken}(k, id, \omega, op)$
    return $\tau_u$

VERIF.VERIFY($q \in \mathcal{Q}, r \in \{0, 1\}^{\ell_r}$):
    return $\Sigma.\text{SearchResult}(k, q, r) \neq \texttt{err}$
```

```
$\mathcal{L}^{\Sigma}_{\text{verif-rand}}$
$k \twoheadleftarrow \Sigma.\mathsf{KeyGen}$

VERIF.SEARCHTOKEN($q \in \Sigma.\mathcal{Q}$):
    $\tau_s \twoheadleftarrow \Sigma.\mathsf{SearchToken}(k, q)$
    return $\tau_s$

VERIF.UPDATETOKEN($id \in \Sigma.\mathcal{I}, \omega \in \Sigma.\Omega, op \in \Sigma.\mathcal{O}$):
    $\tau_u \twoheadleftarrow \Sigma.\mathsf{UpdateToken}(k, id, \omega, op)$
    $\Sigma.\mathsf{Update}(\tau_u)$
    return $\tau_u$

VERIF.VERIFY($q \in \mathcal{Q}, r \in \{0, 1\}^{\ell_r}$):
    $\tau_s \twoheadleftarrow \Sigma.\mathsf{SearchToken}(k, q)$
    $\hat{r} := \Sigma.\mathsf{Search}(\tau_s)$
    return $r == \hat{r}$
```

2.2.1 Leakage Patterns

In this section, we introduce the two most common leakage patterns. Note that information leaked by these patterns is often the starting point of attacks to searchable symmetric encryption schemes [DHP21, Liu+13, OK21, Xu+23]. On the other hand, the vast majority of the schemes do not protect these patterns since it is difficult to do so without incurring performance penalties [Du+23, Sha+21].

First, we define two lists that record all search and update queries.

Definition 2.10 (Q_s and Q_u) *Let Σ be a DSSE scheme. We define the* ***recording of all search queries*** *performed with Σ as*

$$Q_s = \{(t, q) \mid t \in \mathbb{N}^+, q \in \Sigma.\mathcal{Q}\},$$

where t denotes a time instant. Similarly, we define the ***recording of all update queries*** *performed with Σ as*

$$Q_u = \{(t, id, \omega, op) \mid t \in \mathbb{N}^+, id \in \Sigma.\mathcal{I}, \omega \in \Sigma.\Omega, op \in \Sigma.\mathcal{O}\}.$$

Since we defined DSSE in great generality, to include schemes implementing any type of search query, we first need to define a function that tells whether a given symbol relates to a search query or not. Intuitively, the following function returns `true` if a symbol ω appears in the search query q, in any possible form, and otherwise it returns `false`.

Definition 2.11 (Update effect) *Let Σ be a DSSE scheme, let $q \in \Sigma.\mathcal{Q}$ be a search query, and let $\omega \in \Sigma.\Omega$ be any valid symbol. We define the* ***update effect function****, indicating whether adding or removing the symbol ω to any file would influence the result of q, as*

$$\psi : \Sigma.\Omega \times \Sigma.\mathcal{Q} \rightarrow \{\texttt{true}, \texttt{false}\}.$$

The pattern defined next reveals the timestamps of searches involving a given symbol ω.

Definition 2.12 ($\mathsf{sp}(\omega)$) *Let Σ be a DSSE scheme, and let $\omega \in \Sigma.\Omega$ be any valid symbol. We define the **symbol search pattern** of ω as*

$$\mathsf{sp}(\omega) = \{t \mid (t, q) \in Q_s \text{ and } \psi(\omega, q)\},$$

where t denotes a time instant.

Using the above, we define a pattern that lists the sets of timestamps for searched symbols ω contained in the search query q.

Definition 2.13 (Search pattern) *Let Σ be a DSSE scheme, and let $q \in \Sigma.\mathcal{Q}$ be a search query. We define the **search pattern** of q as*

$$\mathsf{sp}(q) = \{\mathsf{sp}(\omega) \mid \psi(\omega, q)\}.$$

Informally, we say that the search pattern reveals which searches are for exactly the same search query q. Note that we could define the search pattern in a stronger form.

$$\mathsf{sp}_2(q) = \{t \mid (t, q) \in Q_s\}$$

Next, the access pattern reveals the identifiers of all files *currently* matching the search query q, while the access volume pattern only tells how many files match.

Definition 2.14 (Access pattern) *Let Σ be a DSSE scheme, and let $q \in \Sigma.\mathcal{Q}$ be a search query. We define the **access pattern** of q as*

$$\mathsf{ap}(q) = \{\mathsf{g}(id) \mid id \in \mathsf{FI}(q)\},$$

*where $\mathsf{g}(\cdot)$ is an arbitrary bijective function (e.g., identity function). Moreover, we define the **access volume pattern** of q as*

$$\mathsf{avp}(q) = |\mathsf{FI}(q)|.$$

2.2.2 Forward Privacy

The notion of forward privacy, first introduced by Stefanov et al. [SPS14] and subsequently formalized by Bost [Bos16], captures the ability of a DSSE scheme to hide the relationship between updates and previous searches. In particular, the server should not learn that a file being updated matches a symbol that was involved in a previous search query.

Forward privacy is particularly important for dynamic schemes due to its ability to thwart both adaptive and non-adaptive file-injection attacks [ZKP16].

Definition 2.15 (Forward privacy) *Let Σ be a CQA2-secure DSSE scheme. We say that Σ has **forward privacy** if the update leakage function $\mathcal{L}_{\mathsf{Update}}$ can be written as*

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \mathcal{L}'(id, op),$$

where $\mathcal{L}'$ is a stateless function.

In the definition above, notice that $\mathcal{L}'(id, op)$ captures the identifier id of the updated file and the update operation op, but fails to have knowledge about which symbol the update is for, thus hiding a potential relationship with previous search tokens.

2.2.3 Backward Privacy

The notion of backward privacy, first mentioned by Stefanov et al. [SPS14] and later formalized by Bost et al. [BMO17], captures the ability of a DSSE scheme to avoid leakage about files added and deleted in between two search queries that would cause a match for those files.

The following pattern, introduced by Bost et al. [BMO17], represents an augmented access pattern that, in addition to the identifiers of the currently matching files, also reveals the time each of those files were added. Note that we renamed it from TimeDB to TimeFI, following the previous definitions. Moreover, we use the ψ function defined above to keep the definition general.

Definition 2.16 ($\mathsf{TimeFI}(q)$) *Let Σ be a DSSE scheme, and let $q \in \Sigma.\mathcal{Q}$ be a search query. We define the **access pattern with insertion time** of q as*

$$\begin{aligned}\mathsf{TimeFI}(q) = \{(t, \mathsf{g}(id)) \mid\ & (t, id, \omega, \texttt{add}) \in Q_u \\ & \text{and } \forall t' \geq t\ (t', id, \omega, \texttt{del}) \notin Q_u \\ & \text{and } \psi(\omega, q)\},\end{aligned}$$

where t denotes a time instant and $\mathsf{g}(\cdot)$ is an arbitrary bijective function (e.g., identity function).

Note that we can now write a formal definition for $\mathsf{FI}(q)$ using the one above as follows:

$$\mathsf{FI}(q) = \{\mathsf{g}(id) \mid \exists t\ (t, \mathsf{g}(id)) \in \mathsf{TimeFI}(q)\}.$$

The pattern defined next, also introduced by Bost et al. [BMO17], reveals the timestamps of updates for a given symbol ω.

Definition 2.17 ($\mathsf{Updates}(\omega)$) *Let Σ be a DSSE scheme, and let $\omega \in \Sigma.\Omega$ be any valid symbol. We define the **symbol update time pattern** of ω as*

$$\mathsf{Updates}(\omega) = \{t \mid (t, id, \omega, op) \in Q_u\}.$$

Using the above, we can define the pattern that reveals the timestamps at which files were updated that contain symbols affecting the result of the search query.

Definition 2.18 (Updates(q)) *Let Σ be a DSSE scheme, and let $q \in \Sigma.\Omega$ be a search query. We define the* ***update time pattern*** *of q as*

$$\mathsf{Updates}(q) = \{\mathsf{Updates}(\omega) \mid \psi(\omega, q)\}.$$

Notice that the update time pattern is a set of timestamp sets. A stronger notion, in which the pattern is a flattened set of timestamps, is also possible.

$$\begin{aligned}\mathsf{Updates}_2(q) &= \bigcup_{T \in \mathsf{Updates}(q)} T \\ &= \{t \mid t \in \mathsf{Updates}(\omega) \text{ and } \psi(\omega, q)\}\end{aligned}$$

However, we couldn't find any scheme in the literature that explicitly satisfies or even considers this stronger alternative notion. For this reason, we stick to the previous definition.

There is one last pattern we need to introduce before giving the definition of backward privacy. This pattern, given by Bost et al. [BMO17], reveals exactly which symbol deletion cancels which addition, for symbols related to the search query.

Definition 2.19 (DelHist(q)) *Let Σ be a DSSE scheme, and let $q \in \Sigma.\Omega$ be a search query. We define the* ***deletion history*** *of q as*

$$\begin{aligned}\mathsf{DelHist}(q) = \{(t_{\mathrm{add}}, t_{\mathrm{del}}) \mid \exists\, id\ (t_{\mathrm{add}}, id, \omega, \mathtt{add}) \in Q_u \\ \text{and } (t_{\mathrm{del}}, id, \omega, \mathtt{del}) \in Q_u \\ \text{and } \psi(\omega, q)\}.\end{aligned}$$

The definition of backward privacy given below is more general than the one proposed by Bost at el. and also introduces a new type of backward privacy [BMO17]. Type-$\mathrm{I_A}$ corresponds to Type-I in the definition by Bost et al., while Type-$\mathrm{I_B}$ is a generalization of both Type-$\mathrm{I^-}$ and Type-C introduced by Zuo et al. [Zuo+20, Zuo+19]. Type-$\mathrm{I_A}$ and Type-$\mathrm{I_B}$ are the strongest levels of backward privacy, while Type-III is the weakest. In fact, the following relations holds:

$$\begin{aligned}\text{Type-I}_\mathrm{A} &\Rightarrow \text{Type-II} \\ \text{Type-I}_\mathrm{B} &\Rightarrow \text{Type-II} \\ \text{Type-II} &\Rightarrow \text{Type-III}.\end{aligned}$$

Note that we cannot say whether Type-$\mathrm{I_A}$ is stronger than Type-$\mathrm{I_B}$ or vice versa, as one does not imply the other. Indeed, the claim by Zuo et al. in [Zuo+19] that Type-$\mathrm{I^-}$ is stronger than Type-I is false in general.

Definition 2.20 (Backward privacy) *Let Σ be a CQA2-secure DSSE scheme.*

- *We say that Σ has* ***backward privacy with insertion pattern (Type-$\mathrm{I_A}$)*** *if the search and update leakage functions $\mathcal{L}_{\mathsf{Search}}$, $\mathcal{L}_{\mathsf{Update}}$ can be written as*

$$\begin{aligned}\mathcal{L}_{\mathsf{Update}}(id, \omega, op) &= \mathcal{L}'(op) \\ \mathcal{L}_{\mathsf{Search}}(q) &= \mathcal{L}''(\mathsf{TimeFI}(q), |\mathsf{Updates}(q)|),\end{aligned}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

- *We say that Σ has* ***backward privacy with update pattern (Type-I$_B$)*** *if the search and update leakage functions $\mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}}$ can be written as*

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \mathcal{L}'(op)$$
$$\mathcal{L}_{\mathsf{Search}}(q) = \mathcal{L}''(\mathsf{ap}(q), \mathsf{Updates}(q)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

- *We say that Σ has* ***backward privacy with insertion pattern and update pattern (Type-II)*** *if the search and update leakage functions $\mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}}$ can be written as*

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \mathcal{L}'(op, \omega)$$
$$\mathcal{L}_{\mathsf{Search}}(q) = \mathcal{L}''(\mathsf{TimeFI}(q), \mathsf{Updates}(q)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

- *We say that Σ has* ***weak backward privacy (Type-III)*** *if the search and update leakage functions $\mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}}$ can be written as*

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \mathcal{L}'(op, \omega)$$
$$\mathcal{L}_{\mathsf{Search}}(q) = \mathcal{L}''(\mathsf{TimeFI}(q), \mathsf{DelHist}(q)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

2.3 Cryptographic Primitives

In the following, we introduce some basic definitions of security and cryptographic primitives. Then, in the next two sections, we describe some homomorphic constructions, which are directly used by the DSSE scheme presented in this work in Section 3.2.1.

Definition 2.21 (SKE scheme) *For a key space $\mathcal{K}$, a message space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$, a* ***symmetric-key encryption (SKE) scheme*** *consists of the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a key $k \in \mathcal{K}$.*
- Enc*: a (possibly randomized) algorithm that takes a key $k \in \mathcal{K}$ and a plaintext message $m \in \mathcal{M}$ as input, and outputs a ciphertext $c \in \mathcal{C}$.*
- Dec*: a deterministic algorithm that takes a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ as input, and outputs a plaintext message $m \in \mathcal{M}$.*

Definition 2.22 (SKE correctness) *A SKE scheme Σ satisfies* ***correctness*** *if for all $k \in \Sigma.\mathcal{K}$ and all $m \in \Sigma.\mathcal{M}$,*

$$\Pr[\Sigma.\mathsf{Dec}(k, \Sigma.\mathsf{Enc}(k, m)) = m] = 1.$$

Definition 2.23 (One-time secrecy) *Let Σ be a SKE scheme. We say that Σ has **one-time secrecy** if $\mathcal{L}^{\Sigma}_{\text{ots-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ots-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{ots-real}}$

OTS.ENC($m \in \Sigma.\mathcal{M}$):
  $k \twoheadleftarrow \Sigma.\text{KeyGen}$
  $c \twoheadleftarrow \Sigma.\text{Enc}(k, m)$
  return $c$
```

```
$\mathcal{L}^{\Sigma}_{\text{ots-rand}}$

OTS.ENC($m \in \Sigma.\mathcal{M}$):
  $c \twoheadleftarrow \Sigma.\mathcal{C}$
  return $c$
```

Definition 2.24 (MAC scheme) *For a key space $\mathcal{K}$, a message space $\mathcal{M}$ and a tag space $\mathcal{T}$, a **message authentication code (MAC) scheme** consists of the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a pair of keys $(k_i, k_a) \in \mathcal{K}^2$.*
- MAC*: a deterministic algorithm that takes a pair of keys $(k_i, k_a) \in \mathcal{K}^2$ and a message $m \in \mathcal{M}$, and outputs a tag $t \in \mathcal{T}$.*

Definition 2.25 (One-time auth) *Let Σ be a MAC scheme. We say that Σ has **one-time authentication** if $\mathcal{L}^{\Sigma}_{\text{ota-real}} \equiv \mathcal{L}^{\Sigma}_{\text{ota-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{ota-real}}$

$k_i \twoheadleftarrow \Sigma.\mathcal{K}$

OTA.MAC($m \in \Sigma.\mathcal{M}$):
  $k_a \twoheadleftarrow \Sigma.\mathcal{K}$
  return $\Sigma.\text{MAC}((k_i, k_a), m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{ota-rand}}$

OTA.MAC($m \in \Sigma.\mathcal{M}$):
  $t \twoheadleftarrow \Sigma.\mathcal{T}$
  return $t$
```

Definition 2.26 (MAC security) *Let Σ be a MAC scheme. We say that Σ is a **secure MAC** if $\mathcal{L}^{\Sigma}_{\text{mac-real}} \approxeq \mathcal{L}^{\Sigma}_{\text{mac-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{mac-real}}$

$k_i \twoheadleftarrow \Sigma.\mathcal{K}$
$K_a$ := empty assoc. array

MAC.GUESS($m \in \Sigma.\mathcal{M}, t \in \Sigma.\mathcal{T}$):
  if $K_a[m]$ undefined:
    $K_a[m] \twoheadleftarrow \Sigma.\mathcal{K}$
  $k_a := K_a[m]$
  return $t == \Sigma.\text{MAC}((k_i, k_a), m)$

MAC.REVEAL($m \in \Sigma.\mathcal{M}$):
  if $K_a[m]$ undefined:
    $K_a[m] \twoheadleftarrow \Sigma.\mathcal{K}$
  $k_a := K_a[m]$
  return $\Sigma.\text{MAC}((k_i, k_a), m)$
```

```
$\mathcal{L}^{\Sigma}_{\text{mac-rand}}$

$L$ := empty assoc. array

MAC.GUESS($m \in \Sigma.\mathcal{M}, t \in \Sigma.\mathcal{T}$):
  if $L[m]$ undefined:
    return false
  return $t == L[m]$

MAC.REVEAL($m \in \Sigma.\mathcal{M}$):
  if $L[m]$ undefined:
    $L[m] \twoheadleftarrow \Sigma.\mathcal{T}$
  return $L[m]$
```

Definition 2.27 (AEAD scheme) *For a key space $\mathcal{K}$, an associated data space $\mathcal{A}$, a message space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$, an* ***authenticated encryption with associated data (AEAD) scheme*** *consists of the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a key $k \in \mathcal{K}$.*
- Enc*: a (possibly randomized) algorithm that takes a key $k \in \mathcal{K}$, some associated data $a \in \mathcal{A}$ and a plaintext message $m \in \mathcal{M}$ as input, and outputs a ciphertext $c \in \mathcal{C}$.*
- Dec*: a deterministic algorithm that takes a key $k \in \mathcal{K}$, some associated data $a \in \mathcal{A}$ and a ciphertext $c \in \mathcal{C}$ as input, and outputs either a plaintext message $m \in \mathcal{M}$ or an error* `err`.

Definition 2.28 (AEAD security) *Let Σ be an AEAD scheme. We say that Σ has* ***authenticated encryption with associated data (AEAD) security*** *if $\mathcal{L}^{\Sigma}_{\text{aead-real}} \approx \mathcal{L}^{\Sigma}_{\text{aead-rand}}$, where:*

```
$\mathcal{L}^{\Sigma}_{\text{aead-real}}$
$k \twoheadleftarrow \Sigma.\text{KeyGen}$
AEAD.ENC($a \in \Sigma.\mathcal{A}, m \in \Sigma.\mathcal{M}$):
    return $\Sigma.\text{Enc}(k, a, m)$
AEAD.DEC($a \in \Sigma.\mathcal{A}, c \in \Sigma.\mathcal{C}$):
    return $\Sigma.\text{Dec}(k, a, c)$
```

```
$\mathcal{L}^{\Sigma}_{\text{aead-rand}}$
$D :=$ empty assoc. array
AEAD.ENC($a \in \Sigma.\mathcal{A}, m \in \Sigma.\mathcal{M}$):
    $c \twoheadleftarrow \Sigma.\mathcal{C}(|m|)$
    $D[(a,c)] := m$
    return $c$
AEAD.DEC($a \in \Sigma.\mathcal{A}, c \in \Sigma.\mathcal{C}$):
    if $D[(a,c)]$ undefined:
        return err
    return $D[(a,c)]$
```

Definition 2.29 (RO model) *We say that a security proof is in the* ***random oracle model (ROM)*** *if every cryptographic hash function $H : \mathcal{X} \to \mathcal{Y}$ (and, accordingly, every extendable-output function) gets treated as a public, random function, which we call a* ***random oracle (RO)****. In particular, we say that H gets treated as a random oracle if $\mathcal{L}^{H}_{\text{ro-real}} \equiv \mathcal{L}^{H}_{\text{ro-ideal}}$, where:*

```
$\mathcal{L}^{H}_{\text{ro-real}}$
RO.QUERY($x \in \mathcal{X}$):
    return $H(x)$
```

```
$\mathcal{L}^{H}_{\text{ro-ideal}}$
$B :=$ empty assoc. array
RO.QUERY($x \in \mathcal{X}$):
    if $B[x]$ undefined:
        $B[x] \twoheadleftarrow \mathcal{Y}$
    return $B[x]$
```

Additionally, we define random oracles with adaptive programmability. We refer to such a function as a ***programmable random oracle (PRO)****. We say that H gets treated as a programmable random oracle if $\mathcal{L}^{H}_{\text{pro-real}} \equiv \mathcal{L}^{H}_{\text{pro-ideal}}$, where:*

```
$\mathcal{L}^{H}_{\text{pro-real}}$
$B :=$ empty assoc. array

PRO.QUERY($x \in \mathcal{X}$):
  if $B[x]$ undefined:
    $B[x] := H(x)$
  return $B[x]$

PRO.PROGRAM($x \in \mathcal{X}, y \in \mathcal{Y}$):
  if $B[x]$ undefined:
    $B[x] := y$
```

```
$\mathcal{L}^{H}_{\text{pro-ideal}}$
$B :=$ empty assoc. array

PRO.QUERY($x \in \mathcal{X}$):
  if $B[x]$ undefined:
    $B[x] \twoheadleftarrow \mathcal{Y}$
  return $B[x]$

PRO.PROGRAM($x \in \mathcal{X}, y \in \mathcal{Y}$):
  if $B[x]$ undefined:
    $B[x] := y$
```

Calls to PRO.PROGRAM *must preserve the output distribution, i.e., uniform on $\mathcal{Y}$.*

Definition 2.30 (PRF security) *Let $F : \{0,1\}^{\lambda} \times \{0,1\}^{in} \to \{0,1\}^{out}$ be a deterministic function. We say that F is a secure* ***pseudorandom function (PRF)*** *if $\mathcal{L}^{F}_{\text{prf-real}} \approx \mathcal{L}^{F}_{\text{prf-rand}}$, where:*

```
$\mathcal{L}^{F}_{\text{prf-real}}$
$k \twoheadleftarrow \{0,1\}^{\lambda}$

PRF.QUERY($x \in \{0,1\}^{in}$):
  return $F(k,x)$
```

```
$\mathcal{L}^{F}_{\text{prf-rand}}$
$T :=$ empty assoc. array

PRF.QUERY($x \in \{0,1\}^{in}$):
  if $T[x]$ undefined:
    $T[x] \twoheadleftarrow \{0,1\}^{out}$
  return $T[x]$
```

Note that, in the above definition, the library $\mathcal{L}^{F}_{\text{prf-rand}}$ is basically a random oracle over fixed-length binary domain and range.

We now introduce a generalization of the classic one-time pad (OTP) symmetric-key encryption scheme, whose security property will be useful later. In particular, classic OTP performs addition (and subtraction) modulo 2, while the following construction performs addition (and subtraction) modulo n. Thus, the key space $\mathcal{K}$, the message space $\mathcal{M}$, and the ciphertext space $\mathcal{C}$ all correspond to the set of integers modulo n.

Construction 2.31 (OTP)

```
KeyGen:                 Enc($k, m \in \mathbb{Z}_n$):       Dec($k, c \in \mathbb{Z}_n$):
  $k \twoheadleftarrow \mathbb{Z}_n$              $c := m + k \bmod n$            $m := c - k \bmod n$
  return $k$              return $c$                  return $m$
```

Claim 2.32 *Construction 2.31 has one-time secrecy (Definition 2.23).*

Proof We must show that:

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$
$\textsc{ots.enc}(m \in \mathbb{Z}_n)$:
    $k \twoheadleftarrow \mathbb{Z}_n$
    $c := m + k \bmod n$
    return $c$
```

$\equiv$

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-rand}}$
$\textsc{ots.enc}(m \in \mathbb{Z}_n)$:
    $c \twoheadleftarrow \mathbb{Z}_n$
    return $c$
```

Arbitrarily fix $m, c \in \mathbb{Z}_n$. By the cancellation property of the ring $\mathbb{Z}/n\mathbb{Z}$, we have that

$$c \equiv_n m + k \iff k \equiv_n c - m.$$

We compute the probability that $\textsc{ots.enc}(m)$ from $\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$ produces output c. That is,

$$\Pr[\textsc{ots.enc}(m) = c] = \Pr[k \equiv_n c - m],$$

where the probability is over the uniform choice of $k \twoheadleftarrow \mathbb{Z}_n$.

Since we are considering a specific choice for m and c, there is only one value of $k \in \mathbb{Z}_n$ that makes $k \equiv_n c - m$ true. Therefore, because of the uniform choice of k, the probability of choosing that particular value is $1/n$.

In conclusion, the output of $\textsc{ots.enc}(m)$ from $\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$ follows the uniform distribution over $\mathbb{Z}_n$, for every input m. Since, the output of $\textsc{ots.enc}(m)$ from $\mathcal{L}^{\text{OTP}}_{\text{ots-rand}}$ is exactly a sample from the uniform distribution over $\mathbb{Z}_n$, the two libraries are interchangeable. ∎

2.3.1 Symmetric Encryption with Homomorphic Addition

The following construction is a symmetric-key encryption (SKE) scheme with an additional algorithm Add that performs addition on any of the three scheme spaces.

Note that this scheme is simply a blockwise version of Construction 2.31. In particular, this construction performs addition (and subtraction) modulo b for each of the d blocks of a message.

Construction 2.33 (AHE)

$\mathcal{K} = \mathbb{Z}_b^d$

$\mathcal{M} = \mathbb{Z}_b^d$

$\mathcal{C} = \mathbb{Z}_b^d$

```
KeyGen:
    $k \twoheadleftarrow \mathcal{K}$
    return $k$

Enc($k \in \mathcal{K}, m \in \mathcal{M}$):
    for $i = 1$ to $d$:
        $c_i := m_i + k_i \bmod b$
    return $(c_1, \ldots, c_d)$

Dec($k \in \mathcal{K}, c \in \mathcal{C}$):
    for $i = 1$ to $d$:
        $m_i := c_i - k_i \bmod b$
    return $(m_1, \ldots, m_d)$

Add$\left(x, y \in \mathbb{Z}_b^d\right)$:
    for $i = 1$ to $d$:
        $z_i := x_i + y_i \bmod b$
    return $(z_1, \ldots, z_d)$
```

To take advantage of the additive homomorphism of AHE, it is sufficient to apply the Add algorithm pairwise to the ciphertexts of interest $c_1, \ldots, c_n$, obtaining an aggregate ciphertext $\hat{c}$. Computing the corresponding plaintext is a matter of getting the aggregate key $\hat{k}$ from the appropriate keys $k_1, \ldots, k_n$ using again Add, and decrypting with the Dec algorithm as usual.

Example 2.34 *Let $b = 10$ and $d = 2$ in AHE. Let us generate three one-time keys.*

$$\begin{aligned} k_1 &\twoheadleftarrow \mathsf{AHE.KeyGen} \\ k_2 &\twoheadleftarrow \mathsf{AHE.KeyGen} \\ k_3 &\twoheadleftarrow \mathsf{AHE.KeyGen} \end{aligned}$$

We encrypt some plaintext messages, $m_1 = (3, 4)$, $m_2 = (1, 5)$, and $m_3 = (8, 2)$.

$$\begin{aligned} c_1 &= \mathsf{AHE.Enc}(k_1, (3, 4)) \\ c_2 &= \mathsf{AHE.Enc}(k_2, (1, 5)) \\ c_3 &= \mathsf{AHE.Enc}(k_3, (8, 2)) \end{aligned}$$

Now, we obtain the aggregate ciphertext $\hat{c}$ and the aggregate key $\hat{k}$.

$$\begin{aligned} c' &= \mathsf{AHE.Add}(c_1, c_2) \\ \hat{c} &= \mathsf{AHE.Add}(c', c_3) \end{aligned}$$

$$\begin{aligned} k' &= \mathsf{AHE.Add}(k_1, k_2) \\ \hat{k} &= \mathsf{AHE.Add}(k', k_3) \end{aligned}$$

Finally, we decrypt $\hat{c}$, using $\hat{k}$, into the aggregate plaintext message $\widehat{m}$.

$$\widehat{m} = \mathsf{AHE.Dec}\big(\hat{k}, \hat{c}\big)$$

It is easy to see that $\widehat{m} = (2, 1)$, independently of the values of the generated keys.

Claim 2.35 *Construction 2.33 satisfies SKE correctness (Definition 2.22).*

Proof Immediate by the fact that subtraction modulo b, which is performed by AHE.Dec, is the inverse operation of addition modulo b, performed by AHE.Enc. Therefore, for all $k \in \mathsf{AHE}.\mathcal{K}$ and all $m \in \mathsf{AHE}.\mathcal{M}$,

$$\Pr[\mathsf{AHE.Dec}(k, \mathsf{AHE.Enc}(k, m)) = m] = 1.$$

∎

Claim 2.36 *Construction 2.33 has one-time secrecy (Definition 2.23).*

Proof We must show that $\mathcal{L}^{\mathsf{AHE}}_{\text{ots-real}} \equiv \mathcal{L}^{\mathsf{AHE}}_{\text{ots-rand}}$. We prove the claim using a sequence of hybrids.

$\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$:

```
$\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$
OTS.ENC($m \in \mathbb{Z}_b^d$):
  $k \twoheadleftarrow \mathbb{Z}_b^d$
  for $i = 1$ to $d$:
    $c_i := m_i + k_i \bmod b$
  return $(c_1, \ldots, c_d)$
```

The starting point is the $\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$ library, with the details of AHE filled in.

We factor out all additions modulo b inside the for-loop in terms of the $\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$ library, where we set $n = b$. In fact, the body of the for-loop is precisely a one-time pad modulo b.

$\mathcal{L}_{\text{hyb-1}}$:

```
OTS.ENC($m \in \mathbb{Z}_b^d$):
  for $i = 1$ to $d$:
    $c_i \twoheadleftarrow$ OTS.ENC'($m_i$)
  return $(c_1, \ldots, c_d)$
```

$\diamond$

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$
OTS.ENC'($m \in \mathbb{Z}_b$):
  $k \twoheadleftarrow \mathbb{Z}_b$
  $c := m + k \bmod b$
  return $c$
```

We apply the (perfect) one-time secrecy of OTP (Claim 2.32), replacing $\mathcal{L}^{\text{OTP}}_{\text{ots-real}}$ with $\mathcal{L}^{\text{OTP}}_{\text{ots-rand}}$. The resulting hybrid is interchangeable with the previous one.

$\mathcal{L}_{\text{hyb-2}}$:

```
OTS.ENC($m \in \mathbb{Z}_b^d$):
  for $i = 1$ to $d$:
    $c_i \twoheadleftarrow$ OTS.ENC'($m_i$)
  return $(c_1, \ldots, c_d)$
```

$\diamond$

```
$\mathcal{L}^{\text{OTP}}_{\text{ots-rand}}$
OTS.ENC'($m \in \mathbb{Z}_b$):
  $c \twoheadleftarrow \mathbb{Z}_b$
  return $c$
```

$\mathcal{L}_{\text{hyb-3}}$:

```
OTS.ENC($m \in \mathbb{Z}_b^d$):
  for $i = 1$ to $d$:
    $c_i \twoheadleftarrow \mathbb{Z}_b$
  return $(c_1, \ldots, c_d)$
```

We have inlined the call to the OTS.ENC' subroutine.

$\mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$:

```
$\mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$
OTS.ENC($m \in \mathbb{Z}_b^d$):
  $c \twoheadleftarrow \mathbb{Z}_b^d$
  return $c$
```

We have simplified the for-loop by sampling all the c_i in one go. The resulting library is exactly $\mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$.

Therefore, we have proved that $\mathcal{L}^{\text{AHE}}_{\text{ots-real}} \equiv \mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$, which completes the proof. ∎

2.3.2 Message Authentication Code with Homomorphic Addition

The following construction is a classic polynomial MAC with a Carter–Wegman structure [WC81]. An additional Add algorithm can be used to additively aggregate tags, given that keys are properly handled.

Construction 2.37 (AHMAC)

$$p = \text{a prime} > 2^\lambda \qquad \mathcal{K} = \mathbb{Z}_p \qquad \mathcal{M} = \mathbb{Z}_p^d \qquad \mathcal{T} = \mathbb{Z}_p$$

```
MAC((k_i, k_a) ∈ 𝒦^2, m ∈ ℳ):
    $y := 0$
    for $i = d$ down to 1:
        $y := (y + m_i) \cdot k_\mathrm{i} \bmod p$
    $t := y + k_\mathrm{a} \bmod p$
    return $t$

KeyGen:
    $(k_\mathrm{i}, k_\mathrm{a}) \twoheadleftarrow \mathcal{K}^2$
    return $(k_\mathrm{i}, k_\mathrm{a})$

Add($u_0, u_1 \in \mathbb{Z}_p$):
    $\hat{u} := u_0 + u_1 \bmod p$
    return $\hat{u}$
```

In order to take advantage of the additive homomorphism, it is necessary to reuse the same integrity key k_i for all messages and derive different authentication keys k_a for each message, independently and uniformly at random.

Example 2.38 *Consider two messages $m_1, m_2 \in AHMAC.\mathcal{M}$. Let*

$$k_\mathrm{i} \twoheadleftarrow \mathbb{Z}_p$$
$$k_{\mathrm{a},1} \twoheadleftarrow \mathbb{Z}_p$$
$$k_{\mathrm{a},2} \twoheadleftarrow \mathbb{Z}_p.$$

The tags for the two messages are computed as follows.

$$t_1 = \mathsf{AHMAC.MAC}\big((k_\mathrm{i}, k_{\mathrm{a},1}), m_1\big) = \sum_{j=1}^{d} m_{1,j} k_\mathrm{i}^j + k_{\mathrm{a},1} \bmod p$$

$$t_2 = \mathsf{AHMAC.MAC}\big((k_\mathrm{i}, k_{\mathrm{a},2}), m_2\big) = \sum_{j=1}^{d} m_{2,j} k_\mathrm{i}^j + k_{\mathrm{a},2} \bmod p$$

Now, we can obtain the combined tag $\hat{t}$ using AHMAC.Add.

$$\begin{aligned}\hat{t} &= \mathsf{AHMAC.Add}(t_1, t_2)\\ &= t_1 + t_2 \bmod p\\ &= \sum_{j=1}^{d} m_{1,j} k_\mathrm{i}^j + k_{\mathrm{a},1} + \sum_{j=1}^{d} m_{2,j} k_\mathrm{i}^j + k_{\mathrm{a},2} \bmod p\\ &= \sum_{j=1}^{d} \Big(m_{1,j} k_\mathrm{i}^j + m_{2,j} k_\mathrm{i}^j\Big) + \big(k_{\mathrm{a},1} + k_{\mathrm{a},2}\big) \bmod p\end{aligned}$$

$$= \sum_{j=1}^{d} (m_{1,j} + m_{2,j}) k_{\mathsf{i}}^{j} + (k_{\mathsf{a},1} + k_{\mathsf{a},2}) \bmod p$$

It is easy to see that $\hat{t}$ is a valid tag for the blockwise additively combined message

$$\hat{m} = ((m_{1,d} + m_{2,d} \bmod p), (m_{1,d-1} + m_{2,d-1} \bmod p), \ldots, (m_{1,1} + m_{2,1} \bmod p)),$$

using integrity key k_i and authentication key

$$k_{\mathsf{a}} = \mathsf{AHMAC.Add}(k_{\mathsf{a},1}, k_{\mathsf{a},2}).$$

Claim 2.39 *Construction 2.37 has one-time authentication (Definition 2.25).*

Proof We must show that $\mathcal{L}^{\mathsf{AHMAC}}_{\mathsf{ota\text{-}real}} \equiv \mathcal{L}^{\mathsf{AHMAC}}_{\mathsf{ota\text{-}rand}}$. We prove the claim using a sequence of hybrids. Notice that the proof is very similar to that of Claim 2.36.

$\mathcal{L}^{\mathsf{AHMAC}}_{\mathsf{ota\text{-}real}}$:

```
$\mathcal{L}^{\mathsf{AHMAC}}_{\mathsf{ota\text{-}real}}$
$k_{\mathsf{i}} \twoheadleftarrow \mathbb{Z}_p$
OTA.MAC($m \in \mathbb{Z}_p^d$):
    $k_{\mathsf{a}} \twoheadleftarrow \mathbb{Z}_p$
    $y := 0$
    for $i = d$ down to 1:
        $y := (y + m_i) \cdot k_{\mathsf{i}} \bmod p$
    $t := y + k_{\mathsf{a}} \bmod p$
    return $t$
```

The starting point is the $\mathcal{L}^{\mathsf{AHMAC}}_{\mathsf{ota\text{-}real}}$ library, with the details of AHMAC filled in.

We factor out the final addition modulo p in terms of the $\mathcal{L}^{\mathsf{OTP}}_{\mathsf{ots\text{-}real}}$ library, where we set $n = p$. In fact, the last line, right before the return statement, is precisely a one-time pad modulo p.

$\mathcal{L}_{\mathsf{hyb\text{-}1}}$:

```
$k_{\mathsf{i}} \twoheadleftarrow \mathbb{Z}_p$
OTA.MAC($m \in \mathbb{Z}_p^d$):
    $y := 0$
    for $i = d$ down to 1:
        $y := (y + m_i) \cdot k_{\mathsf{i}} \bmod p$
    $t \twoheadleftarrow$ OTS.ENC($y$)
    return $t$
```

$\diamond$

```
$\mathcal{L}^{\mathsf{OTP}}_{\mathsf{ots\text{-}real}}$
OTS.ENC($m \in \mathbb{Z}_p$):
    $k \twoheadleftarrow \mathbb{Z}_p$
    $c := m + k \bmod p$
    return $c$
```

We apply the (perfect) one-time secrecy of OTP (Claim 2.32), replacing $\mathcal{L}^{\mathsf{OTP}}_{\text{ots-real}}$ with $\mathcal{L}^{\mathsf{OTP}}_{\text{ots-rand}}$. The resulting hybrid is interchangeable with the previous one.

$\mathcal{L}_{\text{hyb-2}}$:

```
$k_i \twoheadleftarrow \mathbb{Z}_p$
$\textsc{ota.mac}(m \in \mathbb{Z}_p^d)$:
  $y := 0$
  for $i = d$ down to 1:
    $y := (y + m_i) \cdot k_i \bmod p$
  $t \twoheadleftarrow \textsc{ots.enc}(y)$
  return $t$
```

$\diamond$

```
$\mathcal{L}^{\mathsf{OTP}}_{\text{ots-rand}}$
$\textsc{ots.enc}(m \in \mathbb{Z}_p)$:
  $c \twoheadleftarrow \mathbb{Z}_p$
  return $c$
```

$\mathcal{L}_{\text{hyb-3}}$:

```
$k_i \twoheadleftarrow \mathbb{Z}_p$
$\textsc{ota.mac}(m \in \mathbb{Z}_p^d)$:
  $y := 0$
  for $i = d$ down to 1:
    $y := (y + m_i) \cdot k_i \bmod p$
  $t \twoheadleftarrow \mathbb{Z}_p$
  return $t$
```

We have inlined the call to the OTS.ENC subroutine.

$\mathcal{L}^{\mathsf{AHMAC}}_{\text{ota-rand}}$:

```
$\mathcal{L}^{\mathsf{AHMAC}}_{\text{ota-rand}}$
$\textsc{ota.mac}(m \in \mathbb{Z}_p^d)$:
  $t \twoheadleftarrow \mathbb{Z}_p$
  return $t$
```

We have removed all the lines containing unused variables. The resulting library is exactly $\mathcal{L}^{\mathsf{AHMAC}}_{\text{ota-rand}}$.

Therefore, we have proved that $\mathcal{L}^{\mathsf{AHMAC}}_{\text{ota-real}} \equiv \mathcal{L}^{\mathsf{AHMAC}}_{\text{ota-rand}}$, which completes the proof. ∎

Claim 2.40 *Construction 2.37 is a secure MAC (Definition 2.26).*

Proof Immediate by the fact that AHMAC is a textbook Carter–Wegman MAC construction [WC81] using a polynomial universal hash function (UHF) [CW79, Die+92, Tho15]. ∎

3 The Emys Construction

3.1 Overview

Many works on searchable symmetric encryption focus on single-keyword, multi-keyword (i.e., conjunctive), or boolean search [Li+24]. Although these query types are useful in specific contexts, such as relational databases, they may not be adequate for applications used directly by individual users, such as document cloud storage, email services, or note-taking applications. In these situations, typo-resistant searches are preferable. Consequently, designing schemes that support fuzzy multi-keyword search is essential.

Emys is a novel dynamic searchable symmetric encryption (DSSE) scheme that implements fuzzy matching using trigrams and a threshold for typo tolerance. An algorithm parses each file as a sequence of character-based trigrams (where valid characters are from a subset of the Unicode character set) and builds a bit array for each trigram. Every bit in the array represents the presence or absence of the corresponding trigram in a specific file. Below, Section 3.1.1 explains how the bitmap index works in detail. The search query is also broken down into a sequence of trigrams, which are then compared against those in the index. If a file has a sufficient number of trigrams matching, the identifier of that file goes in the result set.

Furthermore, Emys supports search result verification by applying a homomorphic message authentication code (MAC) (see Section 2.3.2) to withstand malicious servers, consistent with the threat model, explained next.

Threat Model. Unlike most schemes found in the literature, we assume the cloud server to be malicious. A traditional honest-but-curious server operates as an honest entity that follows the protocol and always returns the correct result for each query. At the same time, such a curious server will try to learn as much information as possible using the scheme leakage. In real-world scenarios, this is not always the case. A malicious server may only execute a fraction of the search operations or try to forge some results. This behavior may be due to various reasons that benefit the cloud provider, such as reducing computation costs or storage resources. Note that we won't try to prevent a server from deleting files or parts of the index, though we will be able to detect such actions.

On the other hand, we always consider the client to be honest and its underlying hardware and software to be trusted.

Network eavesdroppers are within the threat model. However, we won't address them explicitly, given that using TLS provides sufficient protection.

We assume that an adversary has no way of knowing the exact contents of the files, except for approximate file sizes, or the search queries that users submit. However, the preferences and habits of users and their network of relationships are within the threat model, since we allow the adversary to use social engineering tactics.

File Handling. The scheme handles actual files separately from the index. The DSSE scheme performs index updates and search operations. A dedicated construction handles file metadata and content. The main advantage of this approach is the clear separation of concerns, which enables a modular setup. In fact, since the searchable encryption scheme only handles file identifiers, any file handling mechanism that takes unique identifiers as input can substitute the one proposed here. A downside is that the separation can lead to additional leakage. For instance, if the relationship between file identifiers and actual files is deterministic, the overall scheme will leak some form of access pattern.

Related Work. The EMYS DSSE scheme is based on the `FBDSSE-CQ` scheme for conjunctive queries developed by Zuo et al. [Zuo+21]. We extended `FBDSSE-CQ` by adding support for fuzzy queries and search result verification, reducing the leakage, and providing an explicit construction for handling files.

3.1.1 Extended Bitmap Index

The index is structured as a collection of bit arrays where each array is associated with a specific trigram. The size is the same for all arrays and is equal to $\eta\ell$ (plus some padding to allow encryption and MAC computation to work correctly, though we will ignore it in this section), where ℓ is the maximum number of files that the scheme can handle. So, η bits are used to represent the presence of a certain trigram in a file. In particular, it suffices to set the $i\eta$-th bit to 1 to signify the presence of the i-th file.

Example 3.1 *Let $\ell = 5$ and $\eta = 3$. Consider a dataset with two files, each containing some trigrams.*

$$\mathsf{FS} = \{f_0, f_1 \mid f_0 = \{\omega_1\}, f_1 = \{\omega_2, \omega_3\}\}$$

Then, the corresponding bitmap index is as follows.

$$\omega_1 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{001}}_{f_0}$$

$$
\begin{aligned}
\omega_2 &\mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{001}}_{f_1}\underbrace{\texttt{000}}_{f_0} \\
\omega_3 &\mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{001}}_{f_1}\underbrace{\texttt{000}}_{f_0}
\end{aligned}
$$

To update the index, we use addition and subtraction modulo $2^{\eta\ell}$ (in practice, we will use the Add functions from the homomorphic schemes instead). To represent the addition of the trigram ω_j to file f_i, we add to the bit array for trigram ω_j the bit string with a 1 in position $i\eta$. Similarly, to express the removal of a trigram ω_s from file f_t, we subtract the bit string with only the $t\eta$-th bit set to 1 from the bit array for trigram ω_s.

Example 3.2 *Continuing from Example 3.1, we now want to add ω_2 to file f_0.*

$$
\begin{aligned}
\delta_0 &= \overbrace{\texttt{000}}^{f_4}\overbrace{\texttt{000}}^{f_3}\overbrace{\texttt{000}}^{f_2}\overbrace{\texttt{000}}^{f_1}\overbrace{\texttt{001}}^{f_0} \\
\omega_2 &\mapsto \texttt{000000000001000} + \delta_0 \bmod 2^{15} \\
&= \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{001}}_{f_1}\underbrace{\texttt{001}}_{f_0}
\end{aligned}
$$

Finally, we remove trigram ω_3 from f_1.

$$
\begin{aligned}
\delta_1 &= \overbrace{\texttt{000}}^{f_4}\overbrace{\texttt{000}}^{f_3}\overbrace{\texttt{000}}^{f_2}\overbrace{\texttt{001}}^{f_1}\overbrace{\texttt{000}}^{f_0} \\
\omega_3 &\mapsto \texttt{000000000001000} - \delta_1 \bmod 2^{15} \\
&= \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0}
\end{aligned}
$$

Note that the final state of our file dataset reflected by the index is as follows.

$$\mathsf{FS} = \{f_0, f_1 \mid f_0 = \{\omega_1, \omega_2\}, f_1 = \{\omega_2\}\}$$

To perform a search query $q = \{\ldots, \omega_i, \omega_{i+1}, \ldots\}$, we take the bit arrays corresponding to each trigram in the query and add them together. Then, the result set of file identifiers depends on the threshold value γ: every substring of η bits denoting a value $\geq \gamma$ selects the corresponding file identifier for inclusion in the result set. Notice that the query q can contain a maximum of $2^\eta - 1$ trigrams, otherwise addition overflows within the η-bit substrings.

Example 3.3 *Let $\ell = 5$ and $\eta = 3$. Consider a dataset with four files, each containing some trigrams.*

$$
\begin{aligned}
\mathsf{FS} = \{f_0, f_1, f_2, f_3 \mid f_0 = \{\omega_1, \omega_2\}, f_1 = \{\omega_2, \omega_3\},& \\
f_2 = \{\omega_4, \omega_5, \omega_6\}, f_3 = \{\omega_6, \omega_7, \omega_8\}\}&
\end{aligned}
$$

Then, the corresponding bitmap index is as follows.

$$\begin{array}{ll}
\omega_1 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{001}}_{f_0} & \omega_2 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{001}}_{f_1}\underbrace{\texttt{001}}_{f_0} \\
\omega_3 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{001}}_{f_1}\underbrace{\texttt{000}}_{f_0} & \omega_4 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{001}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0} \\
\omega_5 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{001}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0} & \omega_6 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{001}}_{f_3}\underbrace{\texttt{001}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0} \\
\omega_7 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{001}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0} & \omega_8 \mapsto \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{001}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0}
\end{array}$$

We now perform search query $q = \{\omega_2, \omega_3\}$ with threshold $\gamma = 1$, meaning that we allow for up to $|q| - \gamma = 1$ wrong trigrams.

$$\begin{aligned}
r &= \texttt{000000000001001} + \texttt{000000000001000} \bmod 2^{15} \\
&= \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{000}}_{f_3}\underbrace{\texttt{000}}_{f_2}\underbrace{\texttt{010}}_{f_1}\underbrace{\texttt{001}}_{f_0} \\
&\Rightarrow \mathsf{FS}(q) = \{f_0, f_1\}
\end{aligned}$$

Finally, we perform search query $q' = \{\omega_4, \omega_5, \omega_7\}$ with threshold $\gamma' = 2$, meaning that we allow for up to $|q'| - \gamma' = 1$ wrong trigrams.

$$\begin{aligned}
r' &= \texttt{000000001000000} + \texttt{000000001000000} + \texttt{000001000000000} \bmod 2^{15} \\
&= \underbrace{\texttt{000}}_{f_4}\underbrace{\texttt{001}}_{f_3}\underbrace{\texttt{010}}_{f_2}\underbrace{\texttt{000}}_{f_1}\underbrace{\texttt{000}}_{f_0} \\
&\Rightarrow \mathsf{FS}(q') = \{f_2\}
\end{aligned}$$

3.1.2 Search Tokens and Update Tokens

Internally to Emys, search tokens and update tokens as defined in Definition 2.6 consist of multiple components. In both cases, we encode the resulting object as a bit string for transmission to the server. However, in this section, we are interested in their components before encoding.

Every trigram ω is associated with a counter $\varkappa$ that keeps track of the number of updates performed on that trigram. For a given trigram, its corresponding bit array is at a different location on the server for different values of the counter $\varkappa$. For example, when $\varkappa = i$ we designate ut_i the location on the server. We call this ut_i an *internal update token*. When we want to update a file adding trigram ω to it, we perform the following steps:

1. Build the bit array b representing the update (in the same way as δ_0 and δ_1 in Example 3.2).
2. Encrypt and tag b as (c, t) using Construction 2.33 and Construction 2.37.
3. Increment the counter $\varkappa$ for ω.
4. Generate a new internal update token $ut_\varkappa$.

Finally, we can send $ut_\varkappa$ and (c, t) to the server for storage.

In practice, update tokens τ_u contain one other element in addition to $ut_\varkappa$ and (c, t). However, to appreciate its importance, we first need to go through and understand the search process.

During search for a trigram ω, the server should be able to recompute all the locations that store updates for ω. In other words, the server needs to compute the internal update tokens $ut_\varkappa, ut_{\varkappa-1}, \ldots, ut_0$. To make this possible, we generate each ut_i from a corresponding *internal search token* st_i. In general, we want internal update tokens to be unlinkable until we issue an internal search token to reconstruct the chain. To this end, we compute ut_i from st_i using a keyed hash function. Also, we don't want the server to be able to compute future internal search tokens, thus we generate them by sampling uniformly at random. To avoid sending all internal search tokens $st_\varkappa, st_{\varkappa-1}, \ldots, st_0$, we link the current internal update token $ut_\varkappa$ with the previous internal search token $st_{\varkappa-1}$ by sending a masked previous internal search token $\xi_{\varkappa-1}$ to the server, computed as $\xi_{\varkappa-1} = st_{\varkappa-1} \oplus H_2(k_{\text{upd}}, st_\varkappa)$. In this way, we only need to send st_k upon search. Figure 3.1 illustrates the chain of internal tokens.

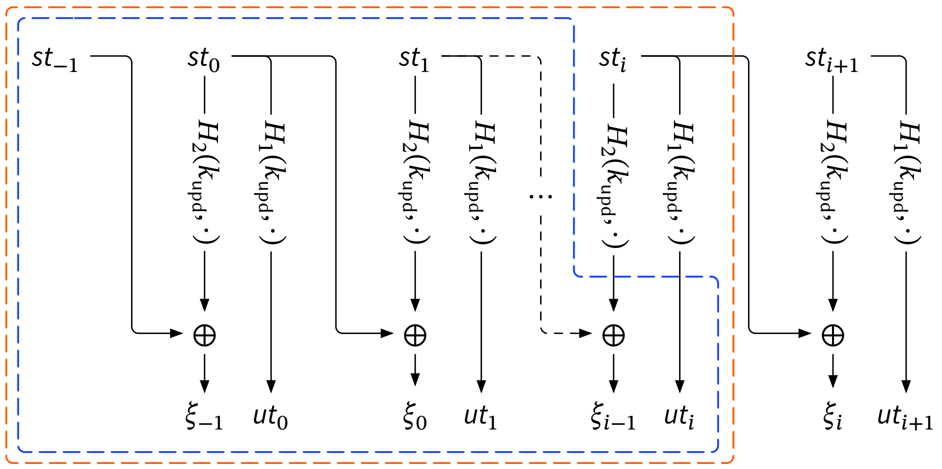

Figure 3.1: *Internal search tokens st_i are sampled uniformly at random. Internal update tokens ut_i are derived using a keyed hash function H_1. Masked internal search tokens ξ_i are computed using another keyed hash function H_2, as $\xi_i = st_i \oplus H_2(k_{upd}, st_{i+1})$. When the counter $\varkappa$ has value i, the client knows all the values inside the orange box, while the server only knows the values inside the blue box.*

At this point, it should be evident that update tokens have three components, that is $\tau_u = (ut_\varkappa, \xi_{\varkappa-1}, (c, t))$, to allow the server to reconstruct the token chain at a later time without the need to store all internal search tokens at the client. Further, search tokens are a set of triples,

$$\tau_s = \{(\varkappa, st_\varkappa, k_{\text{upd}})\}_{\omega \in q},$$

since the server needs to know not only the internal search token $st_\varkappa$, but also the current counter value $\varkappa$ and the key k_{upd} to use with H_1 to move up the chain from the given $st_\varkappa$ to st_0.

3.2 Formal Definitions of the Scheme

In this section, we give the formal definitions for the dynamic searchable symmetric encryption scheme and the file handling scheme of Emys, along with informal descriptions.

3.2.1 DSSE Scheme

The purpose of the DSSE scheme is to operate an encrypted index that enables searches on the encrypted data outsourced by a user.

Practical instantiations of the scheme must specify the values and algorithms to adopt for all the following parameters and functions, respectively, providing a convincing rationale for the choices.

- λ, the security parameter of the scheme, expressed in bits, which defines the security level of the master key k, all derived subkeys, internal search tokens st_i, and internal update tokens ut_i.
- ℓ, the maximum number of files that the scheme can handle.
- η, the number of bits used to denote a file, which determines the maximum number of unique trigrams that the scheme can handle in a single search query as $\mu = 2^\eta - 1$. It must divide λ, i.e., $\exists k \in \mathbb{N}^+$ such that $\lambda = k\eta$.
- $p > 2^\lambda$, the prime modulus for the AHMAC message authentication code scheme, and also the block modulus (i.e., the value of b) for the AHE symmetric-key encryption scheme.
- $d = \lceil \eta\ell/\lambda \rceil$, the number of blocks in the messages used with the AHE symmetric-key encryption scheme. Implicit from the values of λ, ℓ, and η.
- γ, the threshold specifying the minimum number of matching trigrams required to include a file identifier in the result set.
- $H_1 : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, a keyed hash function, used to derive the internal update tokens ut_i from the internal search tokens st_i.
- $H_2 : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, a keyed hash function, used in the computation of the masked internal search tokens ξ_i.
- $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$, a pseudorandom function, used to derive all subkeys. Note that this same PRF is also used to derive all subkeys for the file handling scheme (see Section 3.2.3).
- $E : \{0,1\}^\lambda \to \mathbb{Z}_p^d$, an extendable-output function, used to expand the index encryption keys (see Section 3.2.3).

Section 4.2.1 provides recommendations for sensible parameter values and cryptographic algorithms that implementors can use, together with explanations about the ones selected for the current implementation in this work.

To define the DSSE scheme we follow Definition 2.6. To start, we establish concrete spaces for the scheme.

Definition 3.4 (Emys spaces)

$$\mathcal{K} = \{\texttt{0}, \texttt{1}\}^\lambda \qquad \mathcal{Q} = \bigcup_{k=1}^{2^\eta - 1} \{\omega \mid \omega \in \Omega\}^k$$
$$\mathcal{I} = \{0, \ldots, \ell - 1\} \subseteq \mathbb{N}$$
$$\Omega = \{\text{Unicode codepoints}\}^3 \qquad \mathcal{O} = \{\texttt{add}, \texttt{del}\}$$

Notice that the definition of the query space $\mathcal{Q}$ does not fully align with Definition 2.6. This is intentional, as the complete specification of $\mathcal{Q}$, which is reported below for the sake of completeness, is rather inconvenient to use. The definition given above improves readability and simplifies the algorithms.

$$\mathcal{Q} = \left\{ q_{S,\gamma} \;\middle|\; \begin{array}{l} S \subseteq \Omega, \quad 1 \leq |S| \leq 2^\eta - 1, \\ \gamma \in \{1, \ldots, |S|\}, \\ q_{S,\gamma}(f) = (|S \cap f| \geq \gamma) \end{array} \right\}$$

In particular, Definition 3.4 treats search queries $q \in \mathcal{Q}$ as tuples of symbols ω_i from the symbol space Ω, where their size is between 1 and $2^\eta - 1$, that is $1 \leq |q| \leq 2^\eta - 1$.

Implementations may decide to limit the symbol space Ω to cover only a subset of the Unicode codepoints. This allows to put an upper bound to the size of the index, since it depends on the total number of possible trigrams for a given symbol space. For example, if we consider only the Basic Latin Unicode block, which contains 128 characters [The24], the total number of possible trigrams would be 2^{21}.

Next is the key generation algorithm. This one is straightforward and does not need much comment. However, it should be noticed that practical applications may not use this function and securely derive the master key k using a suitable key derivation function (KDF) on a user-provided passphrase.

Algorithm 3.5 (Emys.KeyGen)

```
KeyGen:
  $k \twoheadleftarrow \mathcal{K}$
  return $k$
```

We assume that both the client and the server keep some internal state in the form of associative arrays. Viewing an associative array T as a key-value store, we denote with T.keys the set of keys stored in T. Formally, $T.\text{keys} = \{x \mid \exists x : T[x] \text{ is defined}\}$.

```
$C$ := empty assoc. array  // client-side state
$S$ := empty assoc. array  // server-side state
```

The algorithm for producing search tokens, which is run by the client, starts by sorting the search query tuple. Any SORT subroutine is valid, as long as the sorting algorithm inside does not change over time. Moreover, the sorting algorithm is allowed to be unstable, as we ignore duplicated symbols in the query anyway. The reason we sort the query is to limit the amount of symbol cooccurrence information an adversary can get. In particular, this prevents leaking information about the frequency of adjacent symbols in user queries.

Then, SearchToken continues symbol by symbol: it skips duplicated trigrams and unknown trigrams, it computes the update token keys k_{upd}, and bundles the search token.

The actual search token is a set of triples. Each triple correspond to one of the trigrams ω_i from the query q, and it contains the update counter $\varkappa$ for ω_i, the internal search token $st_\varkappa$, and the update key k_{upd}.

Algorithm 3.6 (Emys.SearchToken)

```
SearchToken($k \in \mathcal{K}, q \in \mathcal{Q}$):
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $q := \textsc{sort}(q)$
  for each $\omega \in q$:
    if $\omega \in \mathcal{S}$:
      continue  // skip duplicate trigrams
    $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
    if $C[\omega]$ undefined:
      continue  // skip unknown trigrams
    $(\varkappa, st_\varkappa) := C[\omega]$
    $(k_{\text{upd}}, -, -, -) := \textsc{derivekeys}_{\textsc{dsse}}((\omega, \varkappa))$
    $T := T \cup \{(\varkappa, st_\varkappa, k_{\text{upd}})\}$
  $\tau_s := \textsc{encode}(T)$
  return $\tau_s$
```

We abstract away the generation of subkeys via the DERIVEKEYS$_{\text{DSSE}}$ subroutine to simplify the algorithms. Section 3.2.3 explains the internals of DERIVEKEYS$_{\text{DSSE}}$ in detail. Any ENCODE subroutine that serializes objects is acceptable, as long as it is consistent across all algorithms that utilize it.

The Search algorithm run by the server starts by decoding the search token τ_s using DECODE. This subroutine must consistently perform the inverse operation of ENCODE. For each triple in the search token set, the algorithm reconstructs the chain of internal update and search tokens. Corresponding to each ω_i, the process computes the bit array comprising all updates to ω_i in c', and its cumulative MAC tag in t'. Once the inner for-loop terminates, (c', t') contains all updates to ω_i and we can save it at location $ut_\varkappa$. This effectively compresses all updates into one and allows the server to free some memory by deleting old updates. Final bit arrays (c', t') of each ω_i are then accumulated together into $\left(\hat{c}, \hat{t}\right)$ to produce the search result bit array.

Algorithm 3.7 (Emys.Search)

```
Search($\tau_s \in \{0,1\}^{\ell_s}$):
  $T := \textsc{decode}(\tau_s)$
  $\hat{c} := 0$
  $\hat{t} := 0$
  for each $(\varkappa, st_\varkappa, k_{\text{upd}}) \in T$:
    $c' := 0$
    $t' := 0$
    for $i = \varkappa$ down to 0:
      $ut_i := H_1(k_{\text{upd}}, st_i)$
      $(\xi_{i-1}, (c_i, t_i)) := S[ut_i]$
      delete $S[ut_i]$  // free server storage
      $c' :=$ AHE.Add$(c', c_i)$
      $t' :=$ AHMAC.Add$(t', t_i)$
      if $\xi_{i-1} == \perp$:
        break
      $st_{i-1} := \xi_{i-1} \oplus H_2(k_{\text{upd}}, st_i)$
    $S[ut_\varkappa] := (\perp, (c', t'))$  // save accumulated search result
    $\hat{c} :=$ AHE.Add$(\hat{c}, c')$
    $\hat{t} :=$ AHMAC.Add$(\hat{t}, t')$
  $r := \textsc{encode}((\hat{c}, \hat{t}))$
  return $r$
```

Finally, at the client, the SearchResult algorithm extracts the result set R containing file identifiers and simultaneously verifies the integrity and authenticity of the result, effectively providing verifiability of the search results. The algorithm starts by computing the accumulated encryption key $\hat{k_\text{e}}$ and authentication key $\hat{k_\text{a}}$, for each trigram $\omega \in q$ and each corresponding counter from $\varkappa$ to 0. Note that the integrity key k_i does not get accumulated, and the last derivation is implicitly used in the pseudocode, since the key is always the same (see Section 3.2.3). Then, we have the MAC verification step. Since an adversary cannot compute the authentication key $\hat{k_\text{a}}$, if the MAC verification fails, either one or more bit arrays were maliciously modified, one or more malicious bit arrays were added to the result, or one or more valid bit arrays were removed from the result. If the verification is successful, the algorithm proceeds to decrypt the result bit array $\hat{c}$. Finally, we parse the result bit array, η bits at a time. At each iteration i, we check whether the number of trigrams in q that matched for the i-th file is at least equal to γ, and we add i to the result set R if that is the case. The BIN2DEC subroutine performs a simple conversion from binary to decimal integers.

Algorithm 3.8
(Emys.SearchResult)

```
SearchResult$\big(k \in \mathcal{K}, q \in \mathcal{Q}, r \in \{0,1\}^{\ell_r}\big)$:
  $(\hat{c}, \hat{t}) := \textsc{decode}(r)$
  $\hat{k}_e := 0$
  $\hat{k}_a := 0$
  $\mathcal{S} := \emptyset$
  for each $\omega \in q$:
    if $\omega \in \mathcal{S}$:
      continue  // skip duplicate trigrams
    $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
    if $C[\omega]$ undefined:
      continue  // skip unknown trigrams
    $(\varkappa, st_\varkappa) := C[\omega]$
    for $i = \varkappa$ down to 0:
      $(-, k_e, k_a, k_i) := \textsc{deriveKeys}_{\textsc{dsse}}((\omega, i))$
      $\hat{k}_e := \text{AHE.Add}(\hat{k}_e, k_e)$
      $\hat{k}_a := \text{AHMAC.Add}(\hat{k}_a, k_a)$
  if $\hat{t} \neq \text{AHMAC.MAC}((k_i, \hat{k}_a), \hat{c})$:
     // integrity and/or authentication failed
     // server used modified or unexpected values
    return err
  $b := \text{AHE.Dec}(\hat{k}_e, \hat{c})$
  $R := \emptyset$
  for $i = 0$ to $\ell - 1$:
    $m := \textsc{bin2dec}(b[i\eta : i\eta + \eta])$  // padding ignored for brevity
    if $m \geq \gamma$:
      $R := R \cup \{i\}$
  return $R$
```

Note that the MAC-based verifiability technique of Emys is robust against malicious empty search results. This property is not obvious and is often missing from other verifiable schemes [Zhu+25].

The algorithm for generating the update tokens, run by the client, first checks whether the trigram ω to be updated is already known. If not, it initializes a new counter $\varkappa$ to -1 and samples the internal search token st_{-1}, otherwise, the last counter value and internal search token are taken from the client associative array. Then, it generates a new internal search token $st_{\varkappa+1}$, derives the corresponding internal update token $ut_{\varkappa+1}$, and computes the masked previous internal search token $\xi_\varkappa$. The bit array representing the update is built by computing $2^{id \cdot \eta}$ which sets to 1 only the bit in position $id \cdot \eta$. At this point, if the update operation op is the deletion operation, meaning that we want to remove the trigram ω from the file, we negate each of the d blocks of b modulo p. This allows us to restrict the server to performing only addition operations, effectively hiding the update operation type from it, thus promoting backward privacy. Finally, the algorithm encrypts the bit array, computes its MAC tag, and bundles the update token.

Algorithm 3.9 (Emys.UpdateToken)

```
UpdateToken($k \in \mathcal{K}, id \in \mathcal{I}, \omega \in \Omega, op \in \mathcal{O}$):
  if $C[\omega]$ undefined:
    $\varkappa := -1$
    $st_\varkappa \twoheadleftarrow \{0,1\}^\lambda$
  else:
    $(\varkappa, st_\varkappa) := C[\omega]$
  $st_{\varkappa+1} \twoheadleftarrow \{0,1\}^\lambda$
  $C[\omega] := (\varkappa+1, st_{\varkappa+1})$
  $(k_{\mathrm{upd}}, k_{\mathrm{e}}, k_{\mathrm{a}}, k_{\mathrm{i}}) :=$ DERIVEKEYS$_{\mathrm{DSSE}}((\omega, \varkappa+1))$
  $ut_{\varkappa+1} := H_1(k_{\mathrm{upd}}, st_{\varkappa+1})$
  $\xi_\varkappa := st_\varkappa \oplus H_2(k_{\mathrm{upd}}, st_{\varkappa+1})$
  $b := 2^{id \cdot \eta}$  // build the bit array
  interpret $b$ as $d$ elements of $\mathbb{Z}_p$
  if $op$ == del:
     // negate the bit array by blocks on the client
     // since the server performs only additions
    for $i = 1$ to $d$:
      $b_i := -b_i \bmod p$
  $c :=$ AHE.Enc$(k_{\mathrm{e}}, b)$
  $t :=$ AHMAC.MAC$((k_{\mathrm{i}}, k_{\mathrm{a}}), c)$
  $\tau_u :=$ ENCODE$((ut_{\varkappa+1}, \xi_\varkappa, (c, t)))$
  return $\tau_u$
```

The Update algorithm run by the server is quite straightforward. First, it decodes the update token, and then it stores the masked previous internal search token $\xi_\varkappa$ and the encrypted bit array with tag (c, t) at position $ut_{\varkappa+1}$.

Algorithm 3.10 (Emys.Update)

```
Update$\left(\tau_u \in \{0,1\}^{\ell_u}\right)$:
  $(ut_{\varkappa+1}, \xi_\varkappa, (c, t)) :=$ DECODE$(\tau_u)$
  $S[ut_{\varkappa+1}] := (\xi_\varkappa, (c, t))$
```

Collectively, all the above algorithms constitute the Emys DSSE scheme.

Construction 3.11 (Emys DSSE)

Let H_1 and H_2 denote two keyed hash functions, F denote a secure pseudorandom function, and E denote a secure extendable-output function. Then the ***Emys DSSE scheme*** *is defined by:*

- *Spaces as in Definition 3.4.*
- KeyGen *as in Algorithm 3.5.*
- SearchToken *as in Algorithm 3.6.*
- Search *as in Algorithm 3.7.*
- SearchResult *as in Algorithm 3.8.*

- UpdateToken *as in Algorithm 3.9.*
- Update *as in Algorithm 3.10.*

3.2.2 File Handling Scheme

The file handling scheme is responsible for securely managing the content and metadata of files outsourced by a user.

We give each file a unique file path name so that we can treat the files as part of a hierarchical file system. File paths are unrooted, slash-separated sequences of path elements, and we call *base name* the last (i.e., rightmost) path element. We reserve the special "." path name for the root directory. Path elements must be non-empty and cannot be "." or "..", except for the special case of the root directory. If an application is uninterested in the file system structure, it may assign all files to the root directory, giving them unique names that do not include any forward slashes.

The server utilizes two associative arrays to manage file information: one associates file identifiers with file metadata, and the other one relates file identifiers with file contents.

```
$M :=$ empty assoc. array  // file metadata (server-side)
$D :=$ empty assoc. array  // file contents (server-side)
```

File metadata consist of four elements:
- *id*, the unique file identifier, which should match the one used within the searchable symmetric encryption scheme.
- *name*, the unique file path name.
- *size*, the actual file size in bytes.
- *modtime*, the last file modification time in UTC.

Note that, although each file should have a unique *name*, the scheme itself does not enforce this. In other words, applications are responsible for making sure that file path names are unique. An easy way to enforce unique path names would be to use them as keys, possibly obfuscated, for the file contents associative array. However, doing so would complicate the file renaming procedure, hence we avoid it.

Records in the metadata associative array M use an obfuscated variant of the file identifier as keys. Specifically, we obtain the array key h_m using a keyed hash function H_3,

$$h_m = H_3(k_{\mathrm{mk}}, id).$$

The metadata blob $m = (id, name, size, modtime)$ that we store in M is encrypted with freshly generated keys using an authenticated encryption with associated data (AEAD) scheme Γ. We wrap encryption keys and store them together with the encrypted blob. Thus, the value we store in M is a pair.

$$(\Gamma.\mathsf{Enc}(k_{\mathrm{mw}}, (\texttt{meta key}, h_m), k_{\mathrm{m}}), \Gamma.\mathsf{Enc}(k_{\mathrm{m}}, (\texttt{meta data}, h_m), m))$$

Notice how we use the corresponding key h_m as part of the additional authenticated data (AAD) to bind m to the specific file identifier, i.e., the intended context. The metadata blob should have a fixed size to avoid leaking information about file path names. However, this may be problematic: considerable memory usage or limitations on file path names might be needed. In these situations, padding the metadata blob, similarly to file contents, can be an acceptable compromise.

The file contents associative array D uses similar keys. In particular, only the key applied to the hash function H_3 is different.

$$h_d = H_3(k_{\mathrm{dk}}, id)$$

The reason for using different keys is to disconnect the two arrays from the point of view of a stateless adversary and to enable some access pattern protections to function.

We encrypt file contents with fresh keys, wrap these keys, and store them alongside the encrypted files. Similarly to M, the value we store in D is a pair.

$$(\Gamma.\mathrm{Enc}(k_{\mathrm{dw}}, (\texttt{file key}, h_d), k_{\mathrm{d}}), \Gamma.\mathrm{Enc}(k_{\mathrm{d}}, (\texttt{file data}, h_d), \textsc{pad}(f)))$$

As before, we use the key h_d as AAD to bind the contents of the file to its identifier. Doing this guarantees that an adversary cannot swap the contents of two files. The PAD subroutine pads the file contents to reduce the leakage of file sizes. An appropriate UNPAD subroutine removes the padding.

Except for file encryption keys k_{d} and metadata encryption keys k_{m}, all other keys mentioned above, namely k_{mk}, k_{mw}, k_{dk} and k_{dw}, are derived from a master key k using the DERIVEKEYS$_{\mathrm{FH}}$ subroutine as specified in Section 3.2.3.

The following algorithms illustrate how a client interacts with outsourced files. Pseudocode lines highlighted in blue refer to operations that in reality happen on the server, and thus require a network round trip. Indeed, those lines perform read or write operations on either M or D, which reside on the server.

The algorithm for downloading file metadata starts by computing the key h_m for the metadata array M and retrieving the wrapped metadata encryption key and the encrypted metadata blob c_m. Then, it unwraps the metadata encryption key k_{m} and finally uses it to decrypt the metadata blob m.

Algorithm 3.12 (DownloadMeta)

```
DownloadMeta($k, id$):
    $(k_{\mathrm{mk}}, k_{\mathrm{mw}}, -, -) := \textsc{derivekeys}_{\mathrm{FH}}$
    $h_m := H_3(k_{\mathrm{mk}}, id)$
    $(c_k, c_m) := M[h_m]$
    $k_{\mathrm{m}} := \Gamma.\mathrm{Dec}(k_{\mathrm{mw}}, (\texttt{meta key}, h_m), c_k)$
    $m := \Gamma.\mathrm{Dec}(k_{\mathrm{m}}, (\texttt{meta data}, h_m), c_m)$
    return $(k_{\mathrm{m}}, m)$
```

The procedure for downloading file contents is similar. First, we need to decrypt the file key k_{d} using the key wrapping key k_{dw}, and only then can we decrypt the file and remove the padding. The algorithm also returns the file encryption key k_{d}, which we can use to re-encrypt the file after updating it (see Algorithm 3.15 below).

Algorithm 3.13 (DownloadContent)

```
$\underline{\text{DownloadContent}(k, id):}$
    $(-, -, k_{\mathrm{dk}}, k_{\mathrm{dw}}) := \textsc{DeriveKeys}_{\textsc{fh}}$
    $h_d := H_3(k_{\mathrm{dk}}, id)$
    $(c_k, c_f) := D[h_d]$
    $k_{\mathrm{d}} := \Gamma.\mathrm{Dec}(k_{\mathrm{dw}}, (\texttt{file key}, h_d), c_k)$
    $d := \Gamma.\mathrm{Dec}(k_{\mathrm{d}}, (\texttt{file data}, h_d), c_f)$
    $f := \textsc{unpad}(d)$
    return $(k_{\mathrm{d}}, f)$
```

To upload metadata for either a new or an existing file, we basically need to perform the same operations as in Algorithm 3.12 in reverse order. UploadMeta accepts a metadata encryption key k_{m} as input, which can be the same one obtained from Algorithm 3.12 or one freshly sampled uniformly at random as $k_{\mathrm{m}} \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$. Note that, to rename a file, it is sufficient to use this algorithm and no re-encryption of the file contents is required. Moreover, file renaming allows for virtually moving the file to a different directory.

Algorithm 3.14 (UploadMeta)

```
$\underline{\text{UploadMeta}(k, k_{\mathrm{m}}, m):}$
    $(k_{\mathrm{mk}}, k_{\mathrm{mw}}, -, -) := \textsc{DeriveKeys}_{\textsc{fh}}$
    $(id, -, -, -) := m$
    $h_m := H_3(k_{\mathrm{mk}}, id)$
    $c_k \twoheadleftarrow \Gamma.\mathrm{Enc}(k_{\mathrm{mw}}, (\texttt{meta key}, h_m), k_{\mathrm{m}})$
    $c_m \twoheadleftarrow \Gamma.\mathrm{Enc}(k_{\mathrm{m}}, (\texttt{meta data}, h_m), m)$
    $M[h_m] := (c_k, c_m)$
```

Finally, to either upload a new file or an updated existing file, we can use the following algorithm. Similar to UploadMeta, UploadContent also accepts a file encryption key k_{d} as input, which can be the one returned by Algorithm 3.13 or a newly generated key.

Algorithm 3.15 (UploadContent)

```
$\underline{\text{UploadContent}(k, id, k_{\mathrm{d}}, f):}$
    $(-, -, k_{\mathrm{dk}}, k_{\mathrm{dw}}) := \textsc{DeriveKeys}_{\textsc{fh}}$
    $h_d := H_3(k_{\mathrm{dk}}, id)$
    $c_k \twoheadleftarrow \Gamma.\mathrm{Enc}(k_{\mathrm{dw}}, (\texttt{file key}, h_d), k_{\mathrm{d}})$
    $d := \textsc{pad}(f)$
    $c_f \twoheadleftarrow \Gamma.\mathrm{Enc}(k_{\mathrm{d}}, (\texttt{file data}, h_d), d)$
    $D[h_d] := (c_k, c_f)$
```

To remove a file, we set all of its metadata to their zero value, except for the file identifier, and set its contents to ρ bytes of zeros. Application developers should select the value of ρ based on the statistics of the files their application deals with. Note that we intentionally set the file path name to the invalid empty string, so that deleted files can get ignored when building the file system tree from M.

Algorithm 3.16 (RemoveFile)

```
RemoveFile($k, id$):
    $m := (id, "", 0, 0)$
    $f := 0^{8\rho}$  // all-zeros file of $\rho$ bytes
    $k_m \twoheadleftarrow \{0, 1\}^\lambda$
    $k_d \twoheadleftarrow \{0, 1\}^\lambda$
    UploadMeta($k, k_m, m$)
    UploadContent($k, id, k_d, f$)
```

We can define the following helper algorithm to check whether a specific file identifier is available, i.e., whether the user deleted the file that holds that identifier. Notice that checking the metadata is sufficient, as the only reason we leave the zeroed file there is to limit the amount of leaked information during file removal.

Algorithm 3.17 (IsFreeIdentifier)

```
IsFreeIdentifier($k, id$):
    $(-, m) :=$ DownloadMeta($k, id$)
    $(-, name, size, modtime) := m$
    if $name \neq ""$: return false
    if $size \neq 0$: return false
    if $modtime \neq 0$: return false
    return true
```

3.2.3 Key Hierarchy

The two schemes of Emys derive all keys from a long-term master key k. As a result, only one key needs to be stored securely, and it is possible to conveniently re-derive any subkey as required. However, it's important to note that all keys should be regarded as long-term. That is, no key should be considered ephemeral, and exposure will compromise security. Somewhat of an exception are update token derivation keys k_{upd}: these are deliberately made public to the server, though only at the right time according to the DSSE protocol.

Figure 3.2 illustrates the complete key hierarchy. The master key k, all file encryption keys k_d, and all metadata encryption keys k_m get sampled uniformly at random from $\{0, 1\}^\lambda$, where λ is the security parameter. The master key k is only used to derive other keys by using the pseudorandom function F. All derived keys are also λ bits long, but index encryption keys k_e need to be expanded to $d\lambda$ bits from the λ-bit pre-keys $\tilde{k}_e$. An extendable-output function (XOF) is used for key expansion.

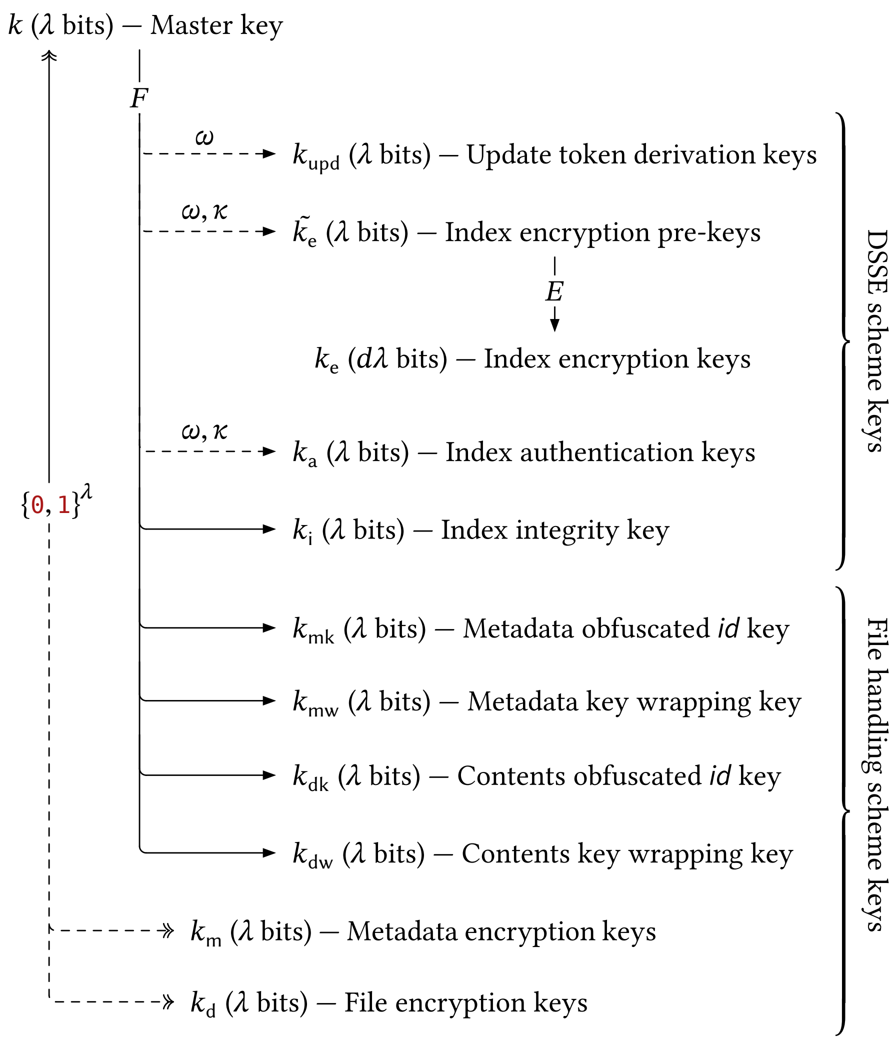
k (λ bits) — Master key
F
ω
k_{upd} (λ bits) — Update token derivation keys
$\omega, \varkappa$
$\tilde{k}_{\mathrm{e}}$ (λ bits) — Index encryption pre-keys
E
k_{e} ($d\lambda$ bits) — Index encryption keys
$\omega, \varkappa$
k_{a} (λ bits) — Index authentication keys
$\{0, 1\}^{\lambda}$
k_{i} (λ bits) — Index integrity key
k_{mk} (λ bits) — Metadata obfuscated *id* key
k_{mw} (λ bits) — Metadata key wrapping key
k_{dk} (λ bits) — Contents obfuscated *id* key
k_{dw} (λ bits) — Contents key wrapping key
k_{m} (λ bits) — Metadata encryption keys
k_{d} (λ bits) — File encryption keys
DSSE scheme keys
File handling scheme keys

Figure 3.2: *Flow of generation and derivation of keys. Two-headed arrows indicate generation by uniform random sampling. Triangle-headed arrows indicate key derivation or key expansion. Solid edges indicate generation or derivation of a single key. Dashed edges indicate generation or derivation of multiple keys, possibly using the variables above as part of domain separation.*

We use domain separation on all derived keys. When the context string used for domain separation has variable length or includes a user-controlled value, such as DSSE symbols ω, we canonicalize the whole context string using the following subroutine. Canonicalization is crucial to avoid attacks such as length extension attacks.

Algorithm 3.18 (CANONICALIZE)

```
CANONICALIZE$((e_1, ..., e_L) : \{0,1\}^* \times \cdots \times \{0,1\}^*)$:
    // assume non-binary inputs are automatically
    // encoded as binary in a consistent manner
    $out := \text{DEC2BIN}(L)$
    for $i = 1$ to $L$:
        $out := out \| \text{DEC2BIN}(|e_i|)$
        $out := out \| e_i$
    return $out$
```

The DEC2BIN subroutine converts decimal integers to binary strings. Integer numbers should be treatead as 64-bit unsigned integers and bit strings should use little-endian bit ordering. The subroutine must behave consistently across platforms and implementations.

Implementations may apply canonicalization to any context string regardedless of their nature to rule out interpretation mistakes.

The following subroutine derives all keys necessary to the DSSE scheme. Note that implementations will probably want to perform key derivation more granularly, precisely when needed. By including both the symbol ω and the update counter $\varkappa$ in deriving index encryption keys and index authentication keys, we ensure that those keys are never use more than once. This is mandatory to provide the security properties of Construction 2.33 and Construction 2.37. The subroutine takes the master key k as an implicit input parameter.

Algorithm 3.19 (DERIVEKEYS$_{\text{DSSE}}$)

```
DERIVEKEYS$_{\text{DSSE}}((\omega, \varkappa) : \{0,1\}^* \times \{0,1\}^*)$:
    // assume non-binary inputs are automatically
    // encoded as binary in a consistent manner
    $k_{\text{upd}} := F(k, \text{CANONICALIZE}((\texttt{update token derivation}, \omega)))$
    $\tilde{k}_{\text{e}} := F(k, \text{CANONICALIZE}((\texttt{index encryption}, \omega, \varkappa)))$
    $k_{\text{e}} := E(\tilde{k}_{\text{e}})$
    $k_{\text{a}} := F(k, \text{CANONICALIZE}((\texttt{index authentication}, \omega, \varkappa)))$
    $k_{\text{i}} := F(k, \texttt{index integrity})$
    return $(k_{\text{upd}}, k_{\text{e}}, k_{\text{a}}, k_{\text{i}})$
```

Correspondingly, the following subroutine derives the keys used by the file handling scheme. Again, we consider the master key k as an implicit input parameter.

Algorithm 3.20 (DERIVEKEYS$_{\text{FH}}$)

```
DERIVEKEYS$_{\text{FH}}$:
    $k_{\text{mk}} := F(k, \texttt{metadata obfuscated id})$
    $k_{\text{mw}} := F(k, \texttt{metadata key wrapping})$
    $k_{\text{dk}} := F(k, \texttt{contents obfuscated id})$
    $k_{\text{dw}} := F(k, \texttt{contents key wrapping})$
    return $(k_{\text{mk}}, k_{\text{mw}}, k_{\text{dk}}, k_{\text{dw}})$
```

3.3 Security Proofs

Claim 3.21 *Let $\mathcal{L}_{\text{Search}}(q) = (\text{sp}(q), \text{Updates}(q))$ and $\mathcal{L}_{\text{Update}}(id, \omega, op) = \bot$. If F is a secure PRF, then the Emys dynamic searchable symmetric encryption scheme (Construction 3.11) has CQA2 security (Definition 2.8) in the random oracle model (Definition 2.29).*

Proof We must show that $\mathcal{L}^{\text{Emys}}_{\text{cqa2-real}} \approxeq \mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}}$. We prove the claim using a sequence of hybrids. First, let us recall the definitions of the two libraries.

```
$\mathcal{L}^{\text{Emys}}_{\text{cqa2-real}}$
$k \twoheadleftarrow \text{Emys.KeyGen}$
CQA.SEARCHTOKEN($q$):
    $\tau_s \twoheadleftarrow \text{Emys.SearchToken}(k, q)$
    return $\tau_s$
CQA.UPDATETOKEN($id, \omega, op$):
    $\tau_u \twoheadleftarrow \text{Emys.UpdateToken}(k, id, \omega, op)$
    return $\tau_u$
```

```
$\mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}}$
CQA.SEARCHTOKEN($q$):
    $\tau_s \twoheadleftarrow \Pi.\text{SimulateSearchToken}(\mathcal{L}_{\text{Search}}(q))$
    return $\tau_s$
CQA.UPDATETOKEN($id, \omega, op$):
    $\tau_u \twoheadleftarrow \Pi.\text{SimulateUpdateToken}(\mathcal{L}_{\text{Update}}(id, \omega, op))$
    return $\tau_u$
```

To keep the pseudocode of hybrids tidy, we apply some simplifications:

- Shorten the CANONICALIZE subroutine name to CANON.
- Shorten the context strings using their initials (e.g., `update token derivation` becomes `utd`)
- Omit the types of the arguments for most of the subroutines.
- Trim out unmodified pseudocode lines, leaving at least one line before and one line after the changes.

By convention, we consider all associative arrays indicated by the letter B to be global tables for their respective random oracles.[1]

Now, we define the simulator Π, which takes the leakage functions $\mathcal{L}_{\text{Search}}(q)$ and $\mathcal{L}_{\text{Update}}(id, \omega, op)$ as inputs.

[1]We can describe a random oracle by saying that "There is a black box. In the box lives a gnome, with a big book and some dice" [Por11]. Thus, we use B as in "book".

Π:

```
$B_1 :=$ empty assoc. array
$B_2 :=$ empty assoc. array
$K_{\text{upd}} :=$ empty assoc. array
$T_{ut} :=$ empty assoc. array
$T_{\xi} :=$ empty assoc. array
$j := 1$

SimulateUpdateToken:
  $ut_{\kappa+1} \twoheadleftarrow \{0,1\}^{\lambda}$
  $\xi_{\kappa} \twoheadleftarrow \{0,1\}^{\lambda}$
  $T_{ut}[j] := ut_{\kappa+1}$
  $T_{\xi}[j] := \xi_{\kappa}$
  $c \twoheadleftarrow \mathbb{Z}_p^d$
  $t \twoheadleftarrow \mathbb{Z}_p$
  $j := j + 1$
  $\tau_u := \textsc{encode}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
  return $\tau_u$

SimulateSearchToken(sp($q$), Updates($q$)):
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $\hat{q} := \textsc{sort}(\text{Updates}(q))$
  for each $V \in \hat{q}$:
    $\hat{\omega} := \min V$
    if $C[\hat{\omega}]$ undefined:
      $st_{-1} \twoheadleftarrow \{0,1\}^{\lambda}$
      $C[\hat{\omega}] := (st_{-1})$
    if $\hat{\omega} \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\hat{\omega}\}$
    if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
    $\kappa := |V| - 1$
    $(st_{-1}, \ldots, st_{|C[\hat{\omega}]|-2}) := C[\hat{\omega}]$
    for $i = |C[\hat{\omega}]| - 1$ to $\kappa$:
      $st_i \twoheadleftarrow \{0,1\}^{\lambda}$
      $C[\hat{\omega}] := (st_{-1}, \ldots, st_i)$
      $B_1[k_{\text{upd}} \| st_i] := T_{ut}[V_{(i)}]$
      $B_2[k_{\text{upd}} \| st_i] := st_{i-1} \oplus T_{\xi}[V_{(i-1)}]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \textsc{encode}(T)$
  return $\tau_s$
```

Notice how the simulator does not use the search pattern sp(q). This suggests that an alternative formulation of the scheme that does not leak the search pattern at all might be possible.

We can now proceed with the sequence of hybrids.

$\mathcal{L}^{\text{EMYS}}_{\text{cqa2-real}}$:

```
$\mathcal{L}^{\text{EMYS}}_{\text{cqa2-real}}$
$k \twoheadleftarrow \{0,1\}^\lambda$

CQA.SEARCHTOKEN($q$):
    $T := \emptyset$
    $\mathcal{S} := \emptyset$
    $q := \text{SORT}(q)$
    for each $\omega \in q$:
        if $\omega \in \mathcal{S}$:
            continue
        $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
        if $C[\omega]$ undefined:
            continue
        $(\kappa, st_\kappa) := C[\omega]$
        $(k_{\text{upd}}, -, -, -) := \text{DERIVEKEYS}_{\text{DSSE}}((\omega, \kappa))$
        $T := T \cup \{(\kappa, st_\kappa, k_{\text{upd}})\}$
    $\tau_s := \text{ENCODE}(T)$
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    if $C[\omega]$ undefined:
        $\kappa := -1$
        $st_\kappa \twoheadleftarrow \{0,1\}^\lambda$
    else:
        $(\kappa, st_\kappa) := C[\omega]$
    $st_{\kappa+1} \twoheadleftarrow \{0,1\}^\lambda$
    $C[\omega] := (\kappa+1, st_{\kappa+1})$
    $(k_{\text{upd}}, k_{\text{e}}, k_{\text{a}}, k_{\text{i}}) := \text{DERIVEKEYS}_{\text{DSSE}}((\omega, \kappa+1))$
    $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
    $\xi_\kappa := st_\kappa \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
    $b := 2^{id \cdot \eta}$
    interpret $b$ as $d$ elements of $\mathbb{Z}_p$
    if $op == \text{del}$:
        for $i = 1$ to $d$:
            $b_i := -b_i \bmod p$
    $c := \text{AHE.Enc}(k_{\text{e}}, b)$
    $t := \text{AHMAC.MAC}((k_{\text{i}}, k_{\text{a}}), c)$
    $\tau_u := \text{ENCODE}((ut_{\kappa+1}, \xi_\kappa, (c, t)))$
    return $\tau_u$
```

The starting point is $\mathcal{L}^{\text{EMYS}}_{\text{cqa2-real}}$. The details of EMYS have been filled in and highlighted.

$\mathcal{L}_{\text{hyb-1}}$:

```
$k \twoheadleftarrow \{0,1\}^\lambda$

CQA.SEARCHTOKEN($q$):
  // ⋯⋯✂——
    $(\kappa, st_\kappa) := C[\omega]$
    $k_{\text{upd}} := F(k, \text{CANON}((\text{utd}, \omega)))$
    $T := T \cup \{(\kappa, st_\kappa, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ⋯⋯✂——
  $C[\omega] := (\kappa + 1, st_{\kappa+1})$
  $k_{\text{upd}} := F(k, \text{CANON}((\text{utd}, \omega)))$
  $\tilde{k}_{\text{e}} := F(k, \text{CANON}((\text{ie}, \omega, \kappa + 1)))$
  $k_{\text{e}} := E(\tilde{k}_{\text{e}})$
  $k_{\text{a}} := F(k, \text{CANON}((\text{ia}, \omega, \kappa + 1)))$
  $k_{\text{i}} := F(k, \text{ii})$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ⋯⋯✂——
  return $\tau_u$
```

We have inlined all the calls to the DERIVEKEYS$_{\text{DSSE}}$ subroutine.

We factor out the statements pertaining to the PRF F in terms of $\mathcal{L}^F_{\text{prf-real}}$.

$\mathcal{L}_{\text{hyb-2}}$:

```
CQA.SEARCHTOKEN($q$):
  // ⋯⋯✂——
    $(\kappa, st_\kappa) := C[\omega]$
    $k_{\text{upd}} := \text{PRF.QUERY}(\text{CANON}((\text{utd}, \omega)))$
    $T := T \cup \{(\kappa, st_\kappa, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ⋯⋯✂——
  $C[\omega] := (\kappa + 1, st_{\kappa+1})$
  $k_{\text{upd}} := \text{PRF.QUERY}(\text{CANON}((\text{utd}, \omega)))$
  $\tilde{k}_{\text{e}} := \text{PRF.QUERY}(\text{CANON}((\text{ie}, \omega, \kappa + 1)))$
  $k_{\text{e}} := E(\tilde{k}_{\text{e}})$
  $k_{\text{a}} := \text{PRF.QUERY}(\text{CANON}((\text{ia}, \omega, \kappa + 1)))$
  $k_{\text{i}} := \text{PRF.QUERY}(\text{ii})$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ⋯⋯✂——
  return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-real}}$
$k \twoheadleftarrow \{0,1\}^\lambda$

PRF.QUERY($x \in \{0,1\}^*$):
  return $F(k, x)$
```

We apply the PRF security (Definition 2.30) of F, replacing $\mathcal{L}^F_{\text{prf-real}}$ with $\mathcal{L}^F_{\text{prf-rand}}$. The resulting hybrid is indistinguishable from the previous one.

$\mathcal{L}_{\text{hyb-3}}$:

```
CQA.SEARCHTOKEN($q$):
    // ······✂———
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^F_{\text{prf-rand}}$
$T :=$ empty assoc. array

PRF.QUERY($x \in \{0,1\}^*$):
    if $T[x]$ undefined:
        $T[x] \twoheadleftarrow \{0,1\}^\lambda$
    return $T[x]$
```

$\mathcal{L}_{\text{hyb-4}}$:

```
$K_{\text{upd}} :=$ empty assoc. array
$K_{\text{e-pre}} :=$ empty assoc. array
$K_{\text{a}} :=$ empty assoc. array
$K_{\text{i}} :=$ empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ······✂———
        $(\kappa, st_\kappa) := C[\omega]$
        if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$ undefined:
            $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^\lambda$
        $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$
        $T := T \cup \{(\kappa, st_\kappa, k_{\text{upd}})\}$
    // ······✂———

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    $C[\omega] := (\kappa + 1, st_{\kappa+1})$
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$ undefined:
        $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^\lambda$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$
    if $K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \kappa + 1))]$ undefined:
        $K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \kappa + 1))] \twoheadleftarrow \{0,1\}^\lambda$
    $\tilde{k}_{\text{e}} := K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \kappa + 1))]$
    $k_{\text{e}} := E(\tilde{k}_{\text{e}})$
    if $K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \kappa + 1))]$ undefined:
        $K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \kappa + 1))] \twoheadleftarrow \{0,1\}^\lambda$
    $k_{\text{a}} := K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \kappa + 1))]$
    if $K_{\text{i}}[\texttt{ii}]$ undefined:
        $K_{\text{i}}[\texttt{ii}] \twoheadleftarrow \{0,1\}^\lambda$
    $k_{\text{i}} := K_{\text{i}}[\texttt{ii}]$
    $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
    // ······✂———
```

We have inlined all the calls to the PRF.QUERY subroutine. Note that each call gets a separate table for storing the evaluation results. Indeed, when we compute the overall advantage at the end, we must take into account 4 PRF evaluations.

We factor out the call to E in terms of $\mathcal{L}^{E}_{\text{ro-real}}$.

$\mathcal{L}_{\text{hyb-5}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$K_{\text{e-pre}}$ := empty assoc. array
$K_{\text{a}}$ := empty assoc. array
$K_{\text{i}}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ······✂───
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂───
    if $K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \varkappa + 1))]$ undefined:
        $K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \varkappa + 1))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
    $\tilde{k}_{\text{e}} := K_{\text{e-pre}}[\text{CANON}((\texttt{ie}, \omega, \varkappa + 1))]$
    $k_{\text{e}} :=$ RO.HASH($\tilde{k}_{\text{e}}$)
    if $K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \varkappa + 1))]$ undefined:
        $K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \varkappa + 1))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
    $k_{\text{a}} := K_{\text{a}}[\text{CANON}((\texttt{ia}, \omega, \varkappa + 1))]$
    // ······✂───
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{E}_{\text{ro-real}}$

RO.HASH($x \in \{\texttt{0}, \texttt{1}\}^{\lambda}$):
    return $E(x)$
```

Since we are in the random oracle model, we can replace $\mathcal{L}^{E}_{\text{ro-real}}$ with $\mathcal{L}^{E}_{\text{ro-ideal}}$. Moreover, $\mathcal{L}_{\text{hyb-5}}$ and $\mathcal{L}_{\text{hyb-6}}$ are interchangeable, i.e., the adversary has zero advantage.

$\mathcal{L}_{\text{hyb-6}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$K_{\text{e-pre}}$ := empty assoc. array
$K_{\text{a}}$ := empty assoc. array
$K_{\text{i}}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ······✂───
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂───
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{E}_{\text{ro-ideal}}$

$B$ := empty assoc. array

RO.HASH($x \in \{\texttt{0}, \texttt{1}\}^{\lambda}$):
    if $B[x]$ undefined:
        $B[x] \twoheadleftarrow \mathbb{Z}_p^d$
    return $B[x]$
```

$\mathcal{L}_{\text{hyb-7}}$:

```
$K_{\mathsf{upd}}$ := empty assoc. array
$K_{\mathsf{e\text{-}pre}}$ := empty assoc. array
$K_{\mathsf{a}}$ := empty assoc. array
$K_{\mathsf{i}}$ := empty assoc. array
$B_{\mathsf{e}}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ······✂——
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂——
    if $K_{\mathsf{e\text{-}pre}}[\text{CANON}((\mathtt{ie}, \omega, \varkappa + 1))]$ undefined:
        $K_{\mathsf{e\text{-}pre}}[\text{CANON}((\mathtt{ie}, \omega, \varkappa + 1))] \twoheadleftarrow \{\mathtt{0}, \mathtt{1}\}^\lambda$
    $\tilde{k}_{\mathsf{e}} := K_{\mathsf{e\text{-}pre}}[\text{CANON}((\mathtt{ie}, \omega, \varkappa + 1))]$
    if $B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}]$ undefined:
        $B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
    $k_{\mathsf{e}} := B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}]$
    if $K_{\mathsf{a}}[\text{CANON}((\mathtt{ia}, \omega, \varkappa + 1))]$ undefined:
    // ······✂——
    return $\tau_u$
```

We have inlined the call to the RO.HASH subroutine. Remember that, since we are in the ROM, table B_{e} is global, i.e., any call to E will use that same table under the hood.

$\mathcal{L}_{\text{hyb-8}}$:

```
$K_{\mathsf{upd}}$ := empty assoc. array
$K_{\mathsf{i}}$ := empty assoc. array
$B_{\mathsf{e}}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ······✂——
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂——
    $k_{\mathsf{upd}} := K_{\mathsf{upd}}[\text{CANON}((\mathtt{utd}, \omega))]$
    $\tilde{k}_{\mathsf{e}} \twoheadleftarrow \{\mathtt{0}, \mathtt{1}\}^\lambda$
    if $B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}]$ undefined:
        $B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
    $k_{\mathsf{e}} := B_{\mathsf{e}}[\tilde{k}_{\mathsf{e}}]$
    $k_{\mathsf{a}} \twoheadleftarrow \{\mathtt{0}, \mathtt{1}\}^\lambda$
    if $K_{\mathsf{i}}[\mathtt{ii}]$ undefined:
        $K_{\mathsf{i}}[\mathtt{ii}] \twoheadleftarrow \{\mathtt{0}, \mathtt{1}\}^\lambda$
    $k_{\mathsf{i}} := K_{\mathsf{i}}[\mathtt{ii}]$
    // ······✂——
    return $\tau_u$
```

For a given value of ω, the related value of $\varkappa$ gets incremented on every call to the CQA.UPDATETOKEN subroutine. Therefore, the if-branch on $K_{\mathsf{e\text{-}pre}}$ and K_{a} are always taken. Thus, simplifying the library has no effect on the calling program. We have also removed the two unused associative arrays.

$\mathcal{L}_{\text{hyb-9}}$:

```
$K_{\text{upd}} :=$ empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$
$B_{\text{e}} :=$ empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $\tilde{k}_{\text{e}} \twoheadleftarrow \{0,1\}^{\lambda}$
  if $B_{\text{e}}[\tilde{k}_{\text{e}}]$ undefined:
    $B_{\text{e}}[\tilde{k}_{\text{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
  $k_{\text{e}} := B_{\text{e}}[\tilde{k}_{\text{e}}]$
  $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

The index integrity key k_{i} is lazily sampled the first time it is needed. Afterward, it is never resampled again, and its value remains unchanged. Therefore, simplifying the code by sampling k_{i} eagerly has no effect on the library's behavior. We have also removed the unused table K_{i}.

$\mathcal{L}_{\text{hyb-10}}$:

```
$K_{\text{upd}} :=$ empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$
$B_{\text{e}} :=$ empty assoc. array
bad := false

CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $\tilde{k}_{\text{e}} \twoheadleftarrow \{0,1\}^{\lambda}$
  if $B_{\text{e}}[\tilde{k}_{\text{e}}]$ undefined:
    $B_{\text{e}}[\tilde{k}_{\text{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
  else:
    bad := true
  $k_{\text{e}} := B_{\text{e}}[\tilde{k}_{\text{e}}]$
  $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

We have introduced a bad variable to set up an identical-until-bad case and later apply the fundamental lemma of game-playing [BR04]. Since the library never actually reads from this variable, the change has no effect on the calling program.

$\mathcal{L}_{\text{hyb-11}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$
$B_{\text{e}}$ := empty assoc. array
bad := false

CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $\tilde{k}_{\text{e}} \twoheadleftarrow \{0,1\}^{\lambda}$
  if $B_{\text{e}}[\tilde{k}_{\text{e}}]$ undefined:
    $B_{\text{e}}[\tilde{k}_{\text{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
  else:
    bad := true
    $B_{\text{e}}[\tilde{k}_{\text{e}}] \twoheadleftarrow \mathbb{Z}_p^d$
  $k_{\text{e}} := B_{\text{e}}[\tilde{k}_{\text{e}}]$
  $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

$\mathcal{L}_{\text{hyb-10}}$ and $\mathcal{L}_{\text{hyb-11}}$ differ only in the highlighted line, which can only be reached when bad = true. Therefore, from the fundamental lemma of game-playing, the advantage is upper bounded by the probability that bad gets set to true, in either hybrid. This probability is the birthday probability, which is upper bounded by

$$\frac{p_e(\lambda)(p_e(\lambda)-1)}{2 \cdot 2^{\lambda}},$$

where the adversary makes a polynomial number $p_e(\lambda)$ of update queries overall. When the calling program runs in polynomial time, the above probability is negligible in the security parameter λ. Hence $\mathcal{L}_{\text{hyb-10}} \approxeq \mathcal{L}_{\text{hyb-11}}$.

$\mathcal{L}_{\text{hyb-12}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$

CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \omega))]$ undefined:
    $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \omega))]$
  $k_{\text{e}} \twoheadleftarrow \mathbb{Z}_p^d$
  $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

In $\mathcal{L}_{\text{hyb-11}}$, the CQA.UPDATETOKEN subroutine is going to assign a fresh value to $B_{\text{e}}[\tilde{k}_{\text{e}}]$ no matter what. We have removed the if-else-statement and simplified the code by sampling directly into k_{e}. We have also removed the unused table B_{e} and the bad variable. This library and the previous one have identical behavior.

We factor out the call to AHE.Enc in terms of $\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$.

$\mathcal{L}_{\text{hyb-13}}$:

```
$K_{\text{upd}} := $ empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$

CQA.SEARCHTOKEN($q$):
    // ······✂———
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$ undefined:
        $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$
    $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
    $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
    $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
    $b := 2^{id \cdot \eta}$
    interpret $b$ as $d$ elements of $\mathbb{Z}_p$
    if $op == \texttt{del}$:
        for $i = 1$ to $d$:
            $b_i := -b_i \bmod p$
    $c \twoheadleftarrow \text{OTS.ENC}(b)$
    $t := \text{AHMAC.MAC}((k_{\text{i}}, k_{\text{a}}), c)$
    $\tau_u := \text{ENCODE}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$

OTS.ENC($m \in \mathbb{Z}_p^d$):
    $k_{\text{e}} \twoheadleftarrow \mathbb{Z}_p^d$
    $c := \text{AHE.Enc}(k_{\text{e}}, m)$
    return $c$
```

We apply the (perfect) one-time secrecy of AHE (Claim 2.36), replacing $\mathcal{L}^{\text{AHE}}_{\text{ots-real}}$ with $\mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$. The resulting hybrid is interchangeable with the previous one.

$\mathcal{L}_{\text{hyb-14}}$:

```
$K_{\text{upd}} := $ empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$

CQA.SEARCHTOKEN($q$):
    // ······✂———
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{\text{AHE}}_{\text{ots-rand}}$

OTS.ENC($m \in \mathbb{Z}_p^d$):
    $c \twoheadleftarrow \mathbb{Z}_p^d$
    return $c$
```

Note that the hop to the above hybrid does not give the adversary any advantage. That may seem odd, since in practice the encryption key is the result of an expansion, and thus cannot be uniform in $\mathbb{Z}_p^d$. However, since we are in the random oracle model, the previous call to E *does* give us a uniform key in $\mathbb{Z}_p^d$.

$\mathcal{L}_{\text{hyb-15}}$:

```
$K_{\text{upd}} := $ empty assoc. array
$k_{\text{i}} \twoheadleftarrow \{0,1\}^{\lambda}$

CQA.SEARCHTOKEN($q$):
    // ......✂——
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ......✂——
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$ undefined:
        $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$
    $k_{\text{a}} \twoheadleftarrow \{0,1\}^{\lambda}$
    $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
    $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
    $c \twoheadleftarrow \mathbb{Z}_p^d$
    $t := \text{AHMAC.MAC}((k_{\text{i}}, k_{\text{a}}), c)$
    $\tau_u := \text{ENCODE}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
    return $\tau_u$
```

We have inlined the call to the OTS.ENC subroutine. We have also removed the lines involving the unused variable b, which in turn eliminates the need for *id* and *op*.

We factor out the call to AHMAC.MAC in terms of $\mathcal{L}^{\text{AHMAC}}_{\text{ota-real}}$. Note how the keys, k_{i} and k_{a}, maintain the correct scope when factoring out. Furthermore, we now sample the keys from $\mathbb{Z}_p$ instead of $\{0,1\}^{\lambda}$, which is possible because $p > 2^{\lambda}$ by definition (Construction 2.37).

$\mathcal{L}_{\text{hyb-16}}$:

```
$K_{\text{upd}} := $ empty assoc. array

CQA.SEARCHTOKEN($q$):
    // ......✂——
    return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
    // ......✂——
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$ undefined:
        $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \omega))]$
    $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
    $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
    $c \twoheadleftarrow \mathbb{Z}_p^d$
    $t \twoheadleftarrow \text{OTA.MAC}(c)$
    $\tau_u := \text{ENCODE}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
    return $\tau_u$
```

$\diamond$

$\mathcal{L}^{\text{AHMAC}}_{\text{ota-real}}$

```
$k_{\text{i}} \twoheadleftarrow \mathbb{Z}_p$

OTA.MAC($m \in \mathbb{Z}_p^d$):
    $k_{\text{a}} \twoheadleftarrow \mathbb{Z}_p$
    return $\text{AHMAC.MAC}((k_{\text{i}}, k_{\text{a}}), m)$
```

We apply the (perfect) one-time authentication of AHMAC (Claim 2.39), replacing $\mathcal{L}^{\text{AHMAC}}_{\text{ota-real}}$ with $\mathcal{L}^{\text{AHMAC}}_{\text{ota-rand}}$. The resulting hybrid is interchangeable with the previous one.

$\mathcal{L}_{\text{hyb-17}}$:

```
$K_{\text{upd}}$ := empty assoc. array
CQA.SEARCHTOKEN($q$):
    // ······✂———
    return $\tau_s$
CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{\text{AHMAC}}_{\text{ota-rand}}$
OTA.MAC($m \in \mathbb{Z}_p^d$):
    $t \twoheadleftarrow \mathbb{Z}_p$
    return $t$
```

$\mathcal{L}_{\text{hyb-18}}$:

```
$K_{\text{upd}}$ := empty assoc. array
CQA.SEARCHTOKEN($q$):
    // ······✂———
    return $\tau_s$
CQA.UPDATETOKEN($id, \omega, op$):
    // ······✂———
    $c \twoheadleftarrow \mathbb{Z}_p^d$
    $t \twoheadleftarrow \mathbb{Z}_p$
    $\tau_u$ := ENCODE(($ut_{\kappa+1}, \xi_\kappa, (c, t)$))
    return $\tau_u$
```

We have inlined the call to the OTA.MAC subroutine.

Fix an arbitrary calling program $\mathcal{A}$. Until this point, the distinguishing advantage of $\mathcal{A}$ is

$$\mathrm{Adv}_{\mathcal{A}}(\lambda) \leq 4 \cdot \mathrm{Adv}^{\mathrm{prf}}_{F,\mathcal{B}_1}(\lambda) + \frac{p_e(\lambda)(p_e(\lambda)-1)}{2 \cdot 2^{\lambda}},$$

where $\mathcal{B}_1$ is an adversary for the PRF security of F.

We now continue with the second part of the sequence of hybrids.

$\mathcal{L}_{\text{hyb-19}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$

CQA.SEARCHTOKEN($q$):
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $q :=$ SORT($q$)
  $U[q] := \emptyset$
  for each $\omega \in q$:
    $U[q] := U[q] \cup V[\omega]$
  for each $\omega \in q$:
    if $\omega \in \mathcal{S}$:
      continue
    // ······✂———
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  if $V[\omega]$ undefined:
    $V[\omega] := \emptyset$
  $V[\omega] := V[\omega] \cup \{j\}$
  if $C[\omega]$ undefined:
    $\kappa := -1$
    $st_\kappa \twoheadleftarrow \{0, 1\}^\lambda$
  else:
    $(\kappa, st_{-1}, \ldots, st_\kappa) := C[\omega]$
  $st_{\kappa+1} \twoheadleftarrow \{0, 1\}^\lambda$
  $C[\omega] := (\kappa + 1, st_{-1}, \ldots, st_{\kappa+1})$
  if $K_{\text{upd}}$[CANON((utd, $\omega$))] undefined:
    $K_{\text{upd}}$[CANON((utd, $\omega$))] $\twoheadleftarrow \{0, 1\}^\lambda$
  $k_{\text{upd}} := K_{\text{upd}}$[CANON((utd, $\omega$))]
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  $\xi_\kappa := st_\kappa \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  $T_{ut}[j] := ut_{\kappa+1}$
  $T_{\xi}[j] := \xi_\kappa$
  $c \twoheadleftarrow \mathbb{Z}_p^d$
  $t \twoheadleftarrow \mathbb{Z}_p$
  $j := j + 1$
  $\tau_u :=$ ENCODE(($ut_{\kappa+1}, \xi_\kappa, (c, t)$))
  return $\tau_u$
```

We have introduced several bookkeeping operations.

- The variable j counts the total number of update queries.
- The tables T_{ut} and T_{ξ} store the internal update tokens and masked internal search tokens, respectively.
- The client state table C now stores all search tokens, rather than just the last one.
- The table V accumulates the update times of each update query for every given symbol ω. Note that, since we do not increment the counter j on search queries, these times will not have a precise match with those counted globally by a calling program. However, this will not negatively impact the reasoning of the next hybrids.
- The table U stores the sets of update times for each symbol ω in a given query q.

These operations do not affect the library's behavior because they are not reflected externally.

We define $\hat{\omega}$ as the first update time of a given symbol ω. Since j is always increasing, symbols ω uniquely map to $\hat{\omega}$. Therefore, we replace each use of ω with $\hat{\omega}$ with no effect on the calling program. Also, we change the if-statement that skips unknown symbols from checking $C[\omega]$ to checking $V[\omega]$.

$\mathcal{L}_{\text{hyb-20}}$:

```
$K_{\text{upd}} := $ empty assoc. array
$V := $ empty assoc. array
$U := $ empty assoc. array
$T_{ut} := $ empty assoc. array
$T_{\xi} := $ empty assoc. array
$j := 1$

CQA.UPDATETOKEN$(id, \omega, op)$:
  if $V[\omega]$ undefined:
    $V[\omega] := \emptyset$
  $V[\omega] := V[\omega] \cup \{j\}$
  $\hat{\omega} := \min V[\omega]$
  if $C[\hat{\omega}]$ undefined:
    $\kappa := -1$
    $st_{\kappa} \twoheadleftarrow \{0,1\}^{\lambda}$
  else:
    $(\kappa, st_{-1}, \ldots, st_{\kappa}) := C[\hat{\omega}]$
  $st_{\kappa+1} \twoheadleftarrow \{0,1\}^{\lambda}$
  $C[\hat{\omega}] := (\kappa+1, st_{-1}, \ldots, st_{\kappa+1})$
  if $K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))]$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  $T_{ut}[j] := ut_{\kappa+1}$
  $T_{\xi}[j] := \xi_{\kappa}$
  $c \twoheadleftarrow \mathbb{Z}_p^d$
  $t \twoheadleftarrow \mathbb{Z}_p$
  $j := j + 1$
  $\tau_u := \text{ENCODE}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
  return $\tau_u$

CQA.SEARCHTOKEN$(q)$:
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $q := \text{SORT}(q)$
  $U[q] := \emptyset$
  for each $\omega \in q$:
    $U[q] := U[q] \cup V[\omega]$
  for each $\omega \in q$:
    if $V[\omega]$ undefined:
      continue
    $\hat{\omega} := \min V[\omega]$
    if $\hat{\omega} \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\hat{\omega}\}$
    $(\kappa, st_{-1}, \ldots, st_{\kappa}) := C[\hat{\omega}]$
    if $K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\text{utd}, \hat{\omega}))]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$
```

We factor out the call to the keyed hash function H_1 in terms of $\mathcal{L}^{H_1}_{\text{pro-real}}$. Moreover, we add a call to PRO1.PROGRAM which programs the random oracle on st_κ, where κ is the last update number for a given symbol ω. Notice that this call is redundant, since the call to PRO1.QUERY already sets the corresponding value. Therefore, this change has no effect on the library's behavior.

$\mathcal{L}_{\text{hyb-21}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$

CQA.SEARCHTOKEN($q$):
  // ······✂———
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$
    $\text{PRO1.PROGRAM}\Big(k_{\text{upd}} \| st_\kappa, T_{ut}\big[V[\omega]_{(\kappa)}\big]\Big)$
    $T := T \cup \{(\kappa, st_\kappa, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂———
  if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$
  $ut_{\kappa+1} := \text{PRO1.QUERY}(k_{\text{upd}} \| st_{\kappa+1})$
  $\xi_\kappa := st_\kappa \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂———
  return $\tau_u$
```

$\diamond$

$\mathcal{L}^{H_1}_{\text{pro-real}}$

```
$B$ := empty assoc. array

PRO1.QUERY($x$):
  if $B[x]$ undefined:
    $B[x] := H_1(x)$
  return $B[x]$

PRO1.PROGRAM($x, y$):
  if $B[x]$ undefined:
    $B[x] := y$
```

Note that we adjusted the usage of H_1 from taking two inputs from $\{\texttt{0}, \texttt{1}\}^{\lambda}$ to a single input in $\{\texttt{0}, \texttt{1}\}^{2\lambda}$. This change preserves the semantic meaning of H_1 and enables us to leverage the random oracle programmability. Nevertheless, we could have defined PRF programmability as well and used that instead, invoking the PRF security definition.

We replace $\mathcal{L}^{H_1}_{\text{pro-real}}$ with $\mathcal{L}^{H_1}_{\text{pro-ideal}}$ by the fact that we are in the random oracle model. The resulting hybrid is interchangeable with the previous one.

$\mathcal{L}_{\text{hyb-22}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$

CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  return $\tau_u$
```

$\diamond$

```
$\mathcal{L}^{H_1}_{\text{pro-ideal}}$

$B$ := empty assoc. array

PRO1.QUERY($x$):
  if $B[x]$ undefined:
    $B[x] \twoheadleftarrow \{0,1\}^{\lambda}$
  return $B[x]$

PRO1.PROGRAM($x, y$):
  if $B[x]$ undefined:
    $B[x] := y$
```

$\mathcal{L}_{\text{hyb-23}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$
$B_1$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂——
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\mathtt{utd}, \hat{\omega}))]$
    if $B_1[k_{\text{upd}} \| st_{\kappa}]$ undefined:
      $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\Big[V[\omega]_{(\kappa)}\Big]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\mathtt{utd}, \hat{\omega}))]$
  if $B_1[k_{\text{upd}} \| st_{\kappa+1}]$ undefined:
    $B_1[k_{\text{upd}} \| st_{\kappa+1}] \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := B_1[k_{\text{upd}} \| st_{\kappa+1}]$
  $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

We have inlined the calls to the PRO1.QUERY and PRO1.PROGRAM subroutines. Since we are in the random oracle model, B_1 is global, and any call to H_1 will use that same table.

$\mathcal{L}_{\text{hyb-24}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$
$B_1$ := empty assoc. array
bad := false
$\mathcal{R}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂——
    if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
    if $B_1[k_{\text{upd}} \| st_{\kappa}]$ undefined:
      $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\Big[V[\omega]_{(\kappa)}\Big]$
    else if $st_{\kappa} \in \mathcal{R}$:
      bad := true
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \textsc{encode}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
  if $B_1[k_{\text{upd}} \| st_{\kappa+1}]$ undefined:
    $B_1[k_{\text{upd}} \| st_{\kappa+1}] \twoheadleftarrow \{0,1\}^{\lambda}$
    $\mathcal{R} := \mathcal{R} \cup \{st_{\kappa+1}\}$
  else:
    bad := true
  $ut_{\kappa+1} := B_1[k_{\text{upd}} \| st_{\kappa+1}]$
  $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

We have introduced a bad variable to set up an identical-until-bad case and an auxiliary set $\mathcal{R}$ to help us identify the bad event. In particular, the bad event happens when an adversary queries H_1 on an internal search token before either the CQA.UPDATETOKEN subroutine does it or before the CQA.SEARCHTOKEN subroutine programs it. This change has no effect on the calling program.

$\mathcal{L}_{\text{hyb-25}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$
$B_1$ := empty assoc. array
bad := false
$\mathcal{R}$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂———
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$
    if $B_1[k_{\text{upd}} \| st_{\kappa}]$ undefined:
      $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\left[V[\omega]_{(\kappa)}\right]$
    else if $st_{\kappa} \in \mathcal{R}$:
      bad := true
      $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\left[V[\omega]_{(\kappa)}\right]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂———
  if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$
  if $B_1[k_{\text{upd}} \| st_{\kappa+1}]$ undefined:
    $B_1[k_{\text{upd}} \| st_{\kappa+1}] \twoheadleftarrow \{0,1\}^{\lambda}$
    $\mathcal{R} := \mathcal{R} \cup \{st_{\kappa+1}\}$
  else:
    bad := true
    $B_1[k_{\text{upd}} \| st_{\kappa+1}] \twoheadleftarrow \{0,1\}^{\lambda}$
  $ut_{\kappa+1} := B_1[k_{\text{upd}} \| st_{\kappa+1}]$
  $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂———
  return $\tau_u$
```

$\mathcal{L}_{\text{hyb-24}}$ and $\mathcal{L}_{\text{hyb-25}}$ differ only in the two highlighted lines, which can only be reached when bad = true. Therefore, from the fundamental lemma of game-playing, the advantage is upper bounded by the probability that bad gets set to true, in either hybrid. As the input to H_1 is uniform in $\{0,1\}^{2\lambda}$, and the adversary can guess among all search tokens, such a probability is upper bounded by

$$p_e(\lambda) \cdot \frac{p_1(\lambda)}{2^{2\lambda}},$$

where the adversary makes a polynomial number $p_1(\lambda)$ of queries to the random oracle H_1. When the calling program runs in polynomial time, the above probability is negligible in the security parameter λ. Hence $\mathcal{L}_{\text{hyb-24}} \approxeq \mathcal{L}_{\text{hyb-25}}$.

$\mathcal{L}_{\text{hyb-26}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_{\xi}$ := empty assoc. array
$j := 1$
$B_1$ := empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂——
    if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
    $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\Big[V[\omega]_{(\kappa)}\Big]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \textsc{encode}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
  $ut_{\kappa+1} \twoheadleftarrow \{\texttt{0}, \texttt{1}\}^{\lambda}$
  $\xi_{\kappa} := st_{\kappa} \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  // ······✂——
  return $\tau_u$
```

In the previous hybrid, the CQA.SEARCHTOKEN and CQA.UPDATETOKEN subroutines are going to assign a specific internal update token and a fresh value, respectively, regardless. We have removed the if-else-statements, and simplified the code of the CQA.UPDATETOKEN subroutine by sampling directly into $ut_{\kappa+1}$. We have also removed the bad variable and the auxiliary variable $\mathcal{R}$, which are now unused. This library and the previous one have identical behavior.

$\mathcal{L}_{\text{hyb-27}}$:

```
$K_{\text{upd}} :=$ empty assoc. array
$V :=$ empty assoc. array
$U :=$ empty assoc. array
$T_{ut} :=$ empty assoc. array
$T_{\xi} :=$ empty assoc. array
$j := 1$
$B_1 :=$ empty assoc. array
$B_2 :=$ empty assoc. array

CQA.SEARCHTOKEN($q$):
  // ······✂——
    $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
    $B_1[k_{\text{upd}} \| st_{\kappa}] := T_{ut}\left[V[\omega]_{(\kappa)}\right]$
    $B_2[k_{\text{upd}} \| st_{\kappa}] := st_{\kappa-1} \oplus T_{\xi}\left[V[\omega]_{(\kappa-1)}\right]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s := \textsc{encode}(T)$
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $C[\hat{\omega}] := (\kappa + 1, st_{-1}, \ldots, st_{\kappa+1})$
  if $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$ undefined:
    $K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
  $k_{\text{upd}} := K_{\text{upd}}[\textsc{canon}((\texttt{utd}, \hat{\omega}))]$
  $ut_{\kappa+1} \twoheadleftarrow \{0,1\}^{\lambda}$
  $\xi_{\kappa} \twoheadleftarrow \{0,1\}^{\lambda}$
  $T_{ut}[j] := ut_{\kappa+1}$
  // ······✂——
  return $\tau_u$
```

We have performed the same operations as $\mathcal{L}_{\text{hyb-23}}$ through $\mathcal{L}_{\text{hyb-26}}$ on H_2, programming it accordingly. The hybrid is indistinguishable from the previous one, with advantage upper bounded by

$$p_e(\lambda) \cdot \frac{p_2(\lambda)}{2^{2\lambda}},$$

where the adversary makes a polynomial number $p_2(\lambda)$ of queries to the random oracle H_2.

$\mathcal{L}_{\text{hyb-28}}$:

```
// ······✂——
CQA.SEARCHTOKEN($q$):
  // ······✂——
  return $\tau_s$

CQA.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  $C[\hat{\omega}] := (\kappa + 1, st_{-1}, \ldots, st_{\kappa+1})$
  $ut_{\kappa+1} \twoheadleftarrow \{0,1\}^{\lambda}$
  // ······✂——
  $\tau_u := \textsc{encode}((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
  return $\tau_u$
```

We have removed the lines involving the unused update token derivation key k_{upd} in the CQA.UPDATETOKEN subroutine.

Since $ut_{\varkappa+1}$ and $\xi_\varkappa$ are now independent of $st_{\varkappa+1}$ and $st_\varkappa$, the only reason for generating them in the CQA.UPDATETOKEN subroutine is to store them in the client state array C. However, it is not necessary to do so that early. Hence, we lazily generate the internal search tokens in the CQA.SEARCHTOKEN subroutine. Moreover, we can recover the last update number $\varkappa$ for a given symbol ω from the size of the set $V[\omega]$; thus, we do not need to store it in C. Notice that we only generate the new tokens we need and never regenerate already issued tokens. We also program both H_1 and H_2 on each new internal search token we generate. These changes have no effect on the calling program.

$\mathcal{L}_{\text{hyb-29}}$:

```
$K_{\text{upd}}$ := empty assoc. array
$V$ := empty assoc. array
$U$ := empty assoc. array
$T_{ut}$ := empty assoc. array
$T_\xi$ := empty assoc. array
$j := 1$
$B_1$ := empty assoc. array
$B_2$ := empty assoc. array

CQA.UPDATETOKEN($id, \omega, op$):
  if $V[\omega]$ undefined:
    $V[\omega] := \emptyset$
  $V[\omega] := V[\omega] \cup \{j\}$
  $ut_{\varkappa+1} \twoheadleftarrow \{0,1\}^\lambda$
  $\xi_\varkappa \twoheadleftarrow \{0,1\}^\lambda$
  $T_{ut}[j] := ut_{\varkappa+1}$
  $T_\xi[j] := \xi_\varkappa$
  $c \twoheadleftarrow \mathbb{Z}_p^d$
  $t \twoheadleftarrow \mathbb{Z}_p$
  $j := j + 1$
  $\tau_u := \text{ENCODE}((ut_{\varkappa+1}, \xi_\varkappa, (c, t)))$
  return $\tau_u$

CQA.SEARCHTOKEN($q$):
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $q := \text{SORT}(q)$
  $U[q] := \emptyset$
  for each $\omega \in q$:
    $U[q] := U[q] \cup V[\omega]$
  for each $\omega \in q$:
    if $V[\omega]$ undefined:
      continue
    $\hat{\omega} := \min V[\omega]$
    if $C[\hat{\omega}]$ undefined:
      $st_{-1} \twoheadleftarrow \{0,1\}^\lambda$
      $C[\hat{\omega}] := (st_{-1})$
    if $\hat{\omega} \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\hat{\omega}\}$
    if $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^\lambda$
    $k_{\text{upd}} := K_{\text{upd}}[\text{CANON}((\texttt{utd}, \hat{\omega}))]$
    $\varkappa := |V[\omega]| - 1$
    $(st_{-1}, \ldots, st_{|C[\hat{\omega}]|-2}) := C[\hat{\omega}]$
    for $i = |C[\hat{\omega}]| - 1$ to $\varkappa$:
      $st_i \twoheadleftarrow \{0,1\}^\lambda$
      $C[\hat{\omega}] := (st_{-1}, \ldots, st_i)$
      $B_1[k_{\text{upd}} \| st_i] := T_{ut}\Big[V[\omega]_{(i)}\Big]$
      $B_2[k_{\text{upd}} \| st_i] := st_{i-1} \oplus T_\xi\Big[V[\omega]_{(i-1)}\Big]$
    $T := T \cup \{(\varkappa, st_\varkappa, k_{\text{upd}})\}$
  $\tau_s := \text{ENCODE}(T)$
  return $\tau_s$
```

Notice that $V[\omega]$ and $U[q]$ mirror exactly Updates(ω) and Updates(q), respectively. We replace $U[q]$ with Updates(q) and rename $V[\omega]$ to simply V. We remove the if-statement that skips unknown symbols since we assume that Updates(q) does not contain empty sets. Likewise, we remove V and U, since we no longer need to build them explicitly, as Updates(q) is an input to the CQA.SEARCHTOKEN subroutine for the simulator. The final result is $\mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}}$.

$\mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}}$:

```
$\mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}}$

$K_{\text{upd}} :=$ empty assoc. array
$T_{ut} :=$ empty assoc. array
$T_{\xi} :=$ empty assoc. array
$j := 1$
$B_1 :=$ empty assoc. array
$B_2 :=$ empty assoc. array

CQA.UPDATETOKEN$(id, \omega, op)$:
  $ut_{\kappa+1} \twoheadleftarrow \{0,1\}^{\lambda}$
  $\xi_{\kappa} \twoheadleftarrow \{0,1\}^{\lambda}$
  $T_{ut}[j] := ut_{\kappa+1}$
  $T_{\xi}[j] := \xi_{\kappa}$
  $c \twoheadleftarrow \mathbb{Z}_p^d$
  $t \twoheadleftarrow \mathbb{Z}_p$
  $j := j + 1$
  $\tau_u :=$ ENCODE$((ut_{\kappa+1}, \xi_{\kappa}, (c, t)))$
  return $\tau_u$

CQA.SEARCHTOKEN$(q)$:
  $T := \emptyset$
  $\mathcal{S} := \emptyset$
  $\hat{q} :=$ SORT(Updates$(q)$)
  for each $V \in \hat{q}$:
    $\hat{\omega} := \min V$
    if $C[\hat{\omega}]$ undefined:
      $st_{-1} \twoheadleftarrow \{0,1\}^{\lambda}$
      $C[\hat{\omega}] := (st_{-1})$
    if $\hat{\omega} \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\hat{\omega}\}$
    if $K_{\text{upd}}[$CANON$((\text{utd}, \hat{\omega}))]$ undefined:
      $K_{\text{upd}}[$CANON$((\text{utd}, \hat{\omega}))] \twoheadleftarrow \{0,1\}^{\lambda}$
    $k_{\text{upd}} := K_{\text{upd}}[$CANON$((\text{utd}, \hat{\omega}))]$
    $\kappa := |V| - 1$
    $(st_{-1}, \ldots, st_{|C[\hat{\omega}]|-2}) := C[\hat{\omega}]$
    for $i = |C[\hat{\omega}]| - 1$ to $\kappa$:
      $st_i \twoheadleftarrow \{0,1\}^{\lambda}$
      $C[\hat{\omega}] := (st_{-1}, \ldots, st_i)$
      $B_1[k_{\text{upd}} \| st_i] := T_{ut}[V_{(i)}]$
      $B_2[k_{\text{upd}} \| st_i] := st_{i-1} \oplus T_{\xi}[V_{(i-1)}]$
    $T := T \cup \{(\kappa, st_{\kappa}, k_{\text{upd}})\}$
  $\tau_s :=$ ENCODE$(T)$
  return $\tau_s$
```

The complete sequence of hybrids is

$$\mathcal{L}^{\text{Emys}}_{\text{cqa2-real}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}} \approxeq \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-10}}$$
$$\mathcal{L}_{\text{hyb-10}} \approxeq \mathcal{L}_{\text{hyb-11}} \equiv \mathcal{L}_{\text{hyb-12}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-24}} \approxeq \mathcal{L}_{\text{hyb-25}}$$
$$\mathcal{L}_{\text{hyb-25}} \equiv \mathcal{L}_{\text{hyb-26}} \approxeq \mathcal{L}_{\text{hyb-27}} \equiv \mathcal{L}_{\text{hyb-28}} \equiv \mathcal{L}_{\text{hyb-29}} \equiv \mathcal{L}^{\text{Emys},\Pi}_{\text{cqa2-rand}},$$

where we highlighted the hops between hybrids that are only indistinguishable, i.e., not interchangeable.

For an arbitrary calling program $\mathcal{A}$, the overall distinguishing advantage it has against the Emys DSSE scheme in the CQA2 security game is

$$\mathrm{Adv}^{\mathrm{cqa2}}_{\textsc{Emys},\mathcal{A}}(\lambda) \leq 4 \cdot \mathrm{Adv}^{\mathrm{prf}}_{F,\mathcal{B}_1}(\lambda) + \frac{p_e(\lambda)}{2^\lambda} \cdot \left(\frac{p_e(\lambda)-1}{2} + \frac{p_1(\lambda)}{2^\lambda} + \frac{p_2(\lambda)}{2^\lambda} \right),$$

where $\mathcal{B}_1$ is an adversary for the PRF security of F, and $p_e(\lambda)$, $p_1(\lambda)$ and $p_2(\lambda)$ are polynomial functions in the security parameter λ as described above.

We have shown that $\mathcal{L}^{\textsc{Emys}}_{\text{cqa2-real}} \approxeq \mathcal{L}^{\textsc{Emys},\Pi}_{\text{cqa2-rand}}$, so the scheme has CQA2 security in the random oracle model. ∎

Claim 3.22 *The Emys dynamic searchable symmetric encryption scheme (Construction 3.11) has forward privacy (Definition 2.15).*

Proof Forward privacy follows directly from Claim 3.21. In particular, since the update leakage function is

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \bot,$$

we have that Emys leaks even less than required to have forward privacy. ∎

Claim 3.23 *The Emys dynamic searchable symmetric encryption scheme (Construction 3.11) has backward privacy with update pattern (Type-I$_B$) (Definition 2.20).*

Proof Backward privacy follows directly from Claim 3.21. In particular, the search and update leakage functions

$$\mathcal{L}_{\mathsf{Update}}(id, \omega, op) = \bot,$$
$$\mathcal{L}_{\mathsf{Search}}(q) = (\mathrm{sp}(q), \mathsf{Updates}(q))$$

leak a subset of the information required to achieve Type-I$_B$. Note that leaking the search pattern $\mathrm{sp}(q)$ is always permitted in backward privacy [BMO17]. Nonetheless, proof of Claim 3.21 instantiates a simulator that does not require the use of $\mathrm{sp}(q)$. ∎

Claim 3.24 *The Emys dynamic searchable symmetric encryption scheme (Construction 3.11) has verifiability (Definition 2.9).*

Proof We must show that $\mathcal{L}^{\textsc{Emys}}_{\text{verif-real}} \approxeq \mathcal{L}^{\textsc{Emys}}_{\text{verif-rand}}$. We prove the claim using a sequence of hybrids.

Similarly to proof of Claim 3.21, we apply some simplifications to keep the pseudocode tidy. In particular, here we also shorten the DERIVEKEYS$_{\text{DSSE}}$ subroutine name to DK$_{\text{DSSE}}$.

$\mathcal{L}^{\text{Emys}}_{\text{verif-real}}$:

```
$\mathcal{L}^{\text{Emys}}_{\text{verif-real}}$
$k \twoheadleftarrow$ Emys.KeyGen

verif.searchtoken($q$):
    $\tau_s \twoheadleftarrow$ Emys.SearchToken($k, q$)
    return $\tau_s$

verif.updatetoken($id, \omega, op$):
    $\tau_u \twoheadleftarrow$ Emys.UpdateToken($k, id, \omega, op$)
    return $\tau_u$

verif.verify($q, r$):
    return Emys.SearchResult($k, q, r$) $\neq$ err
```

As usual, the starting point is $\mathcal{L}^{\text{Emys}}_{\text{verif-real}}$.

$\mathcal{L}_{\text{hyb-1}}$:

```
$k \twoheadleftarrow$ Emys.KeyGen

verif.searchtoken($q$):
    $\tau_s \twoheadleftarrow$ Emys.SearchToken($k, q$)
    return $\tau_s$

verif.updatetoken($id, \omega, op$):
    $\tau_u \twoheadleftarrow$ Emys.UpdateToken($k, id, \omega, op$)
    Emys.Update($\tau_u$)
    return $\tau_u$

verif.verify($q, r$):
    $\tau_s \twoheadleftarrow$ Emys.SearchToken($k, q$)
    $\hat{r} :=$ Emys.Search($\tau_s$)
    return Emys.SearchResult($k, q, r$) $\neq$ err
```

We have added the function calls necessary to maintain a local copy of the server state. These changes have no effect on the library's behavior from the perspective of a calling program.

$\mathcal{L}_{\text{hyb-2}}$:

```
$k \twoheadleftarrow$ EMYS.KeyGen

VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$

VERIF.UPDATETOKEN($id, \omega, op$):
  $\tau_u \twoheadleftarrow$ EMYS.UpdateToken($k, id, \omega, op$)
  EMYS.Update($\tau_u$)
  return $\tau_u$

VERIF.VERIFY($q, r$):
   $\mathcal{S} := \emptyset$
   $q :=$ SORT($q$)
   $\hat{c} := 0$
   $\hat{t} := 0$
   for each $\omega \in q$:
      if $\omega \in \mathcal{S}$:
         continue
      $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
      if $C[\omega]$ undefined:
         continue
      $(\kappa, st_\kappa) := C[\omega]$
      $(k_{\text{upd}}, -, -, -) :=$ DK$_{\text{DSSE}}((\omega, \kappa))$
      $c' := 0$
      $t' := 0$
      for $i = \kappa$ down to 0:
         $ut_i := H_1(k_{\text{upd}}, st_i)$
         $(\xi_{i-1}, (c_i, t_i)) := S[ut_i]$
         delete $S[ut_i]$
         $c' :=$ AHE.Add($c', c_i$)
         $t' :=$ AHMAC.Add($t', t_i$)
         if $\xi_{i-1} == \perp$:
            break
         $st_{i-1} := \xi_{i-1} \oplus H_2(k_{\text{upd}}, st_i)$
      $S[ut_\kappa] := (\perp, (c', t'))$
      $\hat{c} :=$ AHE.Add($\hat{c}, c'$)
      $\hat{t} :=$ AHMAC.Add($\hat{t}, t'$)
  $\hat{r} :=$ ENCODE($(\hat{c}, \hat{t})$)
  return EMYS.SearchResult($k, q, r$) $\neq$ err
```

We have inlined the calls to SearchToken and Search in VERIF.VERIFY, merging the operations in common and skipping the encoding-decoding steps to simplify the code. Indeed, we don't need to store the search tokens in a table like SearchToken does, since we can process them immediately. Notice that we renamed r to $\hat{r}$ to avoid shadowing the input parameter.

$\mathcal{L}_{\text{hyb-3}}$:

```
$k \twoheadleftarrow$ Emys.KeyGen

verif.searchtoken($q$):
  $\tau_s \twoheadleftarrow$ Emys.SearchToken($k, q$)
  return $\tau_s$

verif.updatetoken($id, \omega, op$):
  $\tau_u \twoheadleftarrow$ Emys.UpdateToken($k, id, \omega, op$)
  Emys.Update($\tau_u$)
  return $\tau_u$

verif.verify($q, r$):
  $\mathcal{S} := \emptyset$
  $q :=$ sort($q$)
  $\hat{c} := 0$
  $\hat{t} := 0$
  $\hat{k}_e := 0$
  $\hat{k}_a := 0$
  for each $\omega \in q$:
    if $\omega \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
    if $C[\omega]$ undefined:
      continue
    $(\kappa, st_\kappa) := C[\omega]$
    $c' := 0$
    $t' := 0$
    for $i = \kappa$ down to 0:
      $(k_{\text{upd}}, k_e, k_a, k_i) :=$ DK$_{\text{DSSE}}((\omega, i))$
      $\hat{k}_e :=$ AHE.Add$(\hat{k}_e, k_e)$
      $\hat{k}_a :=$ AHMAC.Add$(\hat{k}_a, k_a)$
      $ut_i := H_1(k_{\text{upd}}, st_i)$
      $(\xi_{i-1}, (c_i, t_i)) := S[ut_i]$
      delete $S[ut_i]$
      $c' :=$ AHE.Add$(c', c_i)$
      $t' :=$ AHMAC.Add$(t', t_i)$
      if $\xi_{i-1} == \perp$:
        break
      $st_{i-1} := \xi_{i-1} \oplus H_2(k_{\text{upd}}, st_i)$
    $S[ut_\kappa] := (\perp, (c', t'))$
    $\hat{c} :=$ AHE.Add$(\hat{c}, c')$
    $\hat{t} :=$ AHMAC.Add$(\hat{t}, t')$
  $\hat{r} :=$ encode$((\hat{c}, \hat{t}))$
  $(c, t) :=$ decode$(r)$
  return $t ==$ AHMAC.MAC$((k_i, \hat{k}_a), c)$
```

We have inlined the call to SearchResult, leveraging the already available code to avoid repetition. Moreover, we have not included the code after the MAC verification statement, since we are only interested in the result of the latter.

$\mathcal{L}_{\text{hyb-4}}$:

```
$k \twoheadleftarrow$ EMYS.KeyGen

VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$

VERIF.UPDATETOKEN($id, \omega, op$):
  if $C[\omega]$ undefined:
    $\kappa := -1$
    $st_\kappa \twoheadleftarrow \{0, 1\}^\lambda$
  else:
    $(\kappa, st_\kappa) := C[\omega]$
  $st_{\kappa+1} \twoheadleftarrow \{0, 1\}^\lambda$
  $C[\omega] := (\kappa + 1, st_{\kappa+1})$
  $(k_{\text{upd}}, k_{\text{e}}, k_{\text{a}}, k_{\text{i}}) := \text{DK}_{\text{DSSE}}((\omega, \kappa + 1))$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  $\xi_\kappa := st_\kappa \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  $b := 2^{id \cdot \eta}$
  interpret $b$ as $d$ elements of $\mathbb{Z}_p$
  if $op == \texttt{del}$:
    for $i = 1$ to $d$:
      $b_i := -b_i \bmod p$
  $c :=$ AHE.Enc($k_{\text{e}}, b$)
  $t :=$ AHMAC.MAC($(k_{\text{i}}, k_{\text{a}}), c$)
  $\tau_u :=$ ENCODE($(ut_{\kappa+1}, \xi_\kappa, (c, t))$)
  $S[ut_{\kappa+1}] := (\xi_\kappa, (c, t))$
  return $\tau_u$

VERIF.VERIFY($q, r$):
  // ······✂———
  $(c, t) :=$ DECODE($r$)
  return $t ==$ AHMAC.MAC($(k_{\text{i}}, \hat{k_{\text{a}}}), c$)
```

We have inlined the calls to UpdateToken and Update, skipping the redundant decoding step from the latter to simplify the code. Note that, even though we could omit this hybrid, we still include it because it helps to better understand subsequent reasoning.

$\mathcal{L}_{\text{hyb-5}}$:

```
$k \twoheadleftarrow$ EMYS.KeyGen
VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$
VERIF.UPDATETOKEN($id, \omega, op$):
  // ······✂———
  $b := 2^{id \cdot \eta}$
  interpret $b$ as $d$ elements of $\mathbb{Z}_p$
  if $op$ == del:
    for $i = 1$ to $d$:
      $b_i := -b_i \bmod p$
  $c :=$ AHE.Enc($k_e, b$)
  $t :=$ AHMAC.MAC(($k_i, k_a$), $c$)
  $\tau_u :=$ ENCODE(($ut_{\kappa+1}, \xi_\kappa, (c, t)$))
  $S[ut_{\kappa+1}] := (\xi_\kappa, (c, t))$
  return $\tau_u$
VERIF.VERIFY($q, r$):
  // ······✂———
    $(\kappa, st_\kappa) := C[\omega]$
    $c' := 0$
    // $t' := 0$
    for $i = \kappa$ down to 0:
      $(k_{upd}, k_e, k_a, k_i) :=$ DK$_{\text{DSSE}}$(($\omega, i$))
      $\hat{k}_e :=$ AHE.Add($\hat{k}_e, k_e$)
      $\hat{k}_a :=$ AHMAC.Add($\hat{k}_a, k_a$)
      $ut_i := H_1(k_{upd}, st_i)$
      $(\xi_{i-1}, (c_i, t_i)) := S[ut_i]$
      delete $S[ut_i]$
      $c' :=$ AHE.Add($c', c_i$)
      // $t' :=$ AHMAC.Add($t', t_i$)
      $st_{i-1} := \xi_{i-1} \oplus H_2(k_{upd}, st_i)$
    $S[ut_\kappa] := (\perp, (c', -))$
    $\hat{c} :=$ AHE.Add($\hat{c}, c'$)
    // $\hat{t} :=$ AHMAC.Add($\hat{t}, t'$)
  $\hat{t} :=$ AHMAC.MAC(($k_i, \hat{k}_a$), $\hat{c}$)
  $\hat{r} :=$ ENCODE(($\hat{c}, \hat{t}$))
  $(c, t) :=$ DECODE($r$)
  return $t$ == AHMAC.MAC(($k_i, \hat{k}_a$), $c$)
```

We have simplified the computation of $\hat{t}$ in VERIF.VERIFY. In particular, because of the line highlighted in blue in VERIF.UPDATETOKEN, the removed lines were computing an aggregated MAC as in Example 2.38.

Below is $\mathcal{L}_{\text{hyb-5}}$ again, shown in its entirety with no lines trimmed out.

$\mathcal{L}_{\text{hyb-5}}$:

```
$k \twoheadleftarrow$ EMYS.KeyGen

VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$

VERIF.UPDATETOKEN($id, \omega, op$):
  if $C[\omega]$ undefined:
    $\kappa := -1$
    $st_\kappa \twoheadleftarrow \{0,1\}^\lambda$
  else:
    $(\kappa, st_\kappa) := C[\omega]$
  $st_{\kappa+1} \twoheadleftarrow \{0,1\}^\lambda$
  $C[\omega] := (\kappa+1, st_{\kappa+1})$
  $(k_{\text{upd}}, k_{\text{e}}, k_{\text{a}}, k_{\text{i}}) :=$ DK$_{\text{DSSE}}((\omega, \kappa+1))$
  $ut_{\kappa+1} := H_1(k_{\text{upd}}, st_{\kappa+1})$
  $\xi_\kappa := st_\kappa \oplus H_2(k_{\text{upd}}, st_{\kappa+1})$
  $b := 2^{id \cdot \eta}$
  interpret $b$ as $d$ elements of $\mathbb{Z}_p$
  if $op ==$ del:
    for $i = 1$ to $d$:
      $b_i := -b_i \bmod p$
  $c :=$ AHE.Enc$(k_{\text{e}}, b)$
  $t :=$ AHMAC.MAC$((k_{\text{i}}, k_{\text{a}}), c)$
  $\tau_u :=$ ENCODE$((ut_{\kappa+1}, \xi_\kappa, (c, t)))$
  $S[ut_{\kappa+1}] := (\xi_\kappa, (c, t))$
  return $\tau_u$

VERIF.VERIFY($q, r$):
  $\mathcal{S} := \emptyset$
  $q :=$ SORT($q$)
  $\hat{c} := 0$
  $\hat{k_{\text{e}}} := 0$
  $\hat{k_{\text{a}}} := 0$
  for each $\omega \in q$:
    if $\omega \in \mathcal{S}$:
      continue
    $\mathcal{S} := \mathcal{S} \cup \{\omega\}$
    if $C[\omega]$ undefined:
      continue
    $(\kappa, st_\kappa) := C[\omega]$
    $c' := 0$
    for $i = \kappa$ down to 0:
      $(k_{\text{upd}}, k_{\text{e}}, k_{\text{a}}, k_{\text{i}}) :=$ DK$_{\text{DSSE}}((\omega, i))$
      $\hat{k_{\text{e}}} :=$ AHE.Add$(\hat{k_{\text{e}}}, k_{\text{e}})$
      $\hat{k_{\text{a}}} :=$ AHMAC.Add$(\hat{k_{\text{a}}}, k_{\text{a}})$
      $ut_i := H_1(k_{\text{upd}}, st_i)$
      $(\xi_{i-1}, (c_i, t_i)) := S[ut_i]$
      delete $S[ut_i]$
      $c' :=$ AHE.Add$(c', c_i)$
      $st_{i-1} := \xi_{i-1} \oplus H_2(k_{\text{upd}}, st_i)$
    $S[ut_\kappa] := (\perp, (c', -))$
    $\hat{c} :=$ AHE.Add$(\hat{c}, c')$
  $\hat{t} :=$ AHMAC.MAC$((k_{\text{i}}, \hat{k_{\text{a}}}), \hat{c})$
  $\hat{r} :=$ ENCODE$((\hat{c}, \hat{t}))$
  $(c, t) :=$ DECODE$(r)$
  return $t ==$ AHMAC.MAC$((k_{\text{i}}, \hat{k_{\text{a}}}), c)$
```

Consider the hybrid library above, and suppose the calling program makes a call to VERIF.VERIF(q, r). For all symbols ω in the search query q, the value of κ read from $C[\omega]$ will always be the last value written by VERIF.UPDATETOKEN (refer to the lines highlighted in blue). Therefore, $\hat{c}$ is precisely the one expected to be found in r. Hence, there are three cases:

- Case 1: both $c = \hat{c}$ and $t = \hat{t}$. The MAC verification will return `true`.
- Case 2: $c = \hat{c}$ but $t \neq \hat{t}$. The MAC verification will return `false`.
- Case 3: $c \neq \hat{c}$, and either $t = \hat{t}$ or $t \neq \hat{t}$. In this case, by the MAC security of AHMAC (Claim 2.40), the MAC verification returns `false` with overwhelming probability.

In all cases, the result of the MAC verification statement is `true` if and only if $r = \hat{r}$ (i.e., both $c = \hat{c}$ and $t = \hat{t}$), at least with overwhelming probability.

$\mathcal{L}_{\text{hyb-6}}$:

```
$k \twoheadleftarrow$ EMYS.KeyGen

VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$

VERIF.UPDATETOKEN($id, \omega, op$):
  // ······✂——
  return $\tau_u$

VERIF.VERIFY($q, r$):
  // ······✂——
          $c' := $ AHE.Add($c', c_i$)
          $st_{i-1} := \xi_{i-1} \oplus H_2(k_{\text{upd}}, st_i)$
     $S[ut_\kappa] := (\bot, (c', -))$
     $\hat{c} :=$ AHE.Add($\hat{c}, c'$)
  $\hat{t} :=$ AHMAC.MAC($(k_i, \hat{k}_a), \hat{c}$)
  $\hat{r} :=$ ENCODE($(\hat{c}, \hat{t})$)
  $(c, t) :=$ DECODE($r$)
  return $r == \hat{r}$
```

We have modified the return statement of the VERIF.VERIFY subroutine according to the discussion above. The resulting hybrid is indistinguishable from the previous one.

$\mathcal{L}^{\text{EMYS}}_{\text{verif-rand}}$:

```
$\mathcal{L}^{\text{EMYS}}_{\text{verif-rand}}$
$k \twoheadleftarrow$ EMYS.KeyGen

VERIF.SEARCHTOKEN($q$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  return $\tau_s$

VERIF.UPDATETOKEN($id, \omega, op$):
  $\tau_u \twoheadleftarrow$ EMYS.UpdateToken($k, id, \omega, op$)
  EMYS.Update($\tau_u$)
  return $\tau_u$

VERIF.VERIFY($q, r$):
  $\tau_s \twoheadleftarrow$ EMYS.SearchToken($k, q$)
  $\hat{r} :=$ EMYS.Search($\tau_s$)
  return $r == \hat{r}$
```

We have factored out all function calls, as they were in $\mathcal{L}_{\text{hyb-1}}$, effectively reversing the various simplifications we made along the sequence. The resulting library is precisely $\mathcal{L}^{\text{EMYS}}_{\text{verif-rand}}$.

The complete sequence of hybrids is

$$\mathcal{L}^{\text{EMYS}}_{\text{verif-real}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}^{\text{EMYS}}_{\text{verif-rand}},$$

where the only indistinguishable hop is the one highlighted.

For an arbitrary calling program $\mathcal{A}$, the distinguishing advantage it has against the Emys DSSE scheme in the verifiability security game is

$$\mathrm{Adv}^{\mathsf{verif}}_{\textsc{Emys},\mathcal{A}}(\lambda) \leq p_v(\lambda) \cdot \mathrm{Adv}^{\mathsf{mac}}_{\mathsf{AHMAC},\mathcal{B}_1}(\lambda),$$

where $\mathcal{B}_1$ is an adversary for the MAC security of AHMAC, and the adversary makes a polynomial number $p_v(\lambda)$ of queries to verif.verify, in the security parameter λ.

We have shown that $\mathcal{L}^{\textsc{Emys}}_{\mathsf{verif\text{-}real}} \approx \mathcal{L}^{\textsc{Emys}}_{\mathsf{verif\text{-}rand}}$, so the scheme has verifiability. ∎

4 Implementing for Software Developers

In this chapter, we use the Go programming language [The25a] for all coding purposes. There are several reasons for this choice, which we discuss on a case-by-case basis in the subsequent sections.

4.1 Programming Interface

Go has a compact and simple syntax and implements parametric polymorphism (i.e., generics) in a way that is lightweight for users of generic code. This tends to lead to highly readable code, which is also straightforward to port to other programming languages. Furthermore, Go favors composition over inheritance, which grants great flexibility [Wik25]. As a result, the programming interface presented next, in Listing 4.1, should be easy to understand and adopt in most real-world situations.

The `Query` type represents a search query of an SSE scheme and can take any value for maximum expressiveness. Concrete implementations must document the actual allowed types and are encouraged to provide a concrete type. For example, if a scheme only supports numeric searches, its implementation may provide an integer query type.

The `SearchToken`, `SearchResult`, and `UpdateToken` types represent τ_s, r, and τ_u, respectively, as in Definition 2.6. Since they are meant to be transferred over a network, they are encoded as sequences of bytes. To allow for interoperability, scheme designers must specify the encoding format.

The `Change` type represents a set of changes to a single file. It is parameterized by a comparable type `T` used for file identifiers. For instance, a scheme that uses UUIDs [DPL24] to identify files might instantiate `T` with the `string` type. A value of type `Change` has two fields:

- The `FileID` field must contain the unique file identifier corresponding to the updated file.

- The `Diff` field represents the concrete set of changes encoded as a sequence of bytes. Implementations must document the expected encoding format, taking into consideration the properties of the specific scheme.

```go
package sse

type Query any

type SearchToken []byte

type SearchResult []byte

type Searcher[T comparable] interface {
    Search(query Query) (SearchToken, error)
    OpenResult(query Query, result SearchResult) ([]T, error)
}

type SearchResolver interface {
    ResolveSearch(token SearchToken) (SearchResult, error)
}

type Change[T comparable] struct {
    FileID T
    Diff   []byte
}

type UpdateToken []byte

type Updater[T comparable] interface {
    Update(changes ...Change[T]) ([]UpdateToken, error)
}

type UpdateResolver interface {
    ResolveUpdates(tokens ...UpdateToken) error
}
```

Listing 4.1: *General programming interface for SSE schemes, written in Go. The use of small interfaces allows capturing both static and dynamic schemes.*

The `Searcher` type is the interface implemented by an SSE client that provides search capabilities through the generation of search tokens and parsing of search results. The `Search` method is responsible for parsing the given search query and generating an appropriate search token. The `OpenResult` method expects the same search query and the corresponding search result produced by a proper server, and it extracts the related result set of file identifiers. Moreover, `OpenResult` is also responsible for verifying the correctness of search results, if the scheme has verifiability (Definition 2.9).

The `SearchResolver` type is the interface implemented by an SSE server that provides search capabilities by parsing search tokens and producing search results. Notice how a server does not know the type for file identifiers. In fact, file identifiers should always be opaque to a server by design. The interface contains only the `ResolveSearch` method, which is responsible for performing the actual encrypted search.

The `Updater` type is the interface implemented by an SSE client that performs file updates using update tokens. The `Update` method is responsible for generating the update tokens, and it takes a variable number of file changes to allow implementations to support batch updates and integrate optimizations, such as exploiting common modifications to multiple files.

Finally, the `UpdateResolver` type is the interface implemented by an SSE server that provides file update capabilities by consuming update tokens. The `ResolveUpdates` method updates the internal state of the server to reflect the changes embedded in the given tokens.

Static searchable symmetric encryption schemes only need their client implementation to satisfy the `Searcher` interface and their server implementation to satisfy the `SearchResolver` interface. On the other hand, dynamic schemes should effectively provide client and server implementations satisfying the following interface compositions, respectively.

```go
type SearchUpdater[T comparable] interface {
    Searcher[T]
    Updater[T]
}

type SearchUpdateResolver interface {
    SearchResolver
    UpdateResolver
}
```

Note that the programming interface proposed in Listing 4.1 abstracts away everything related to cryptographic keys. This places key management responsibility on the scheme implementors, who have the opportunity to minimize the amount of cryptographic material that users are required to handle when dealing with a concrete implementation.

4.2 Implementation of Emys

In addition to the advantages already mentioned in the previous section, implementing Emys using Go is an attractive choice because it offers a comprehensive set of cryptographic packages in its standard library [The25b, The25c], which are developed by following sane principles [Val19].

The development of Emys takes place on GitHub[1], and the last commit at the time of writing is `55bd792e`. Presently, the implementation only covers the DSSE scheme (Section 3.2.1) and will be expanded in the future to include the file handling scheme (Section 3.2.2).

[1]https://github.com/BuriedInTheGround/emys

To perform operations with large moduli, necessary for AHE and AHMAC, the current implementation of Emys uses the `filippo.io/bigmod` package, which provides the ability to do constant-time big integer arithmetic. Other time-sensitive operations, such as MAC verification and XOR'ing secret values, utilize the appropriate functions from the `crypto/subtle` package, located in the standard library.

The implementation of the function for canonicalization of context strings (Algorithm 3.18) follows PASETO's pre-authentication encoding (PAE) [Par22]. In addition to the context values specified in Algorithm 3.19 and Algorithm 3.20, the implementation of Emys prefixes two more context strings to further minimize the feasibility of confused deputy attacks [Goo25]:

- The scheme unique identifier and version number `emys-sse.org/v1`. This guarantees a clear separation from other schemes and across major versions of Emys, which may introduce breaking changes.
- A user-associated 192-bit nonce, which makes each context globally unique. However, note that this work does not study the multi-user setting. Thus, this is only a preventive measure.

The implementation provides the following concrete type for search queries.

```go
type Query struct {
    Text               string
    precomputedTrigrams []string
}
```

The `Text` field is the only exported field, and it can take any string value. In fact, the Emys implementation of the `Search` and `OpenResult` methods accepts both a value of type `Query` and a value of type `string`, for user convenience. A pointer to a value of type `Query` is also valid and actually preferred, since it enables a performance optimization. The two methods lazily populate the unexported `precomputedTrigrams` field so that later invocations getting the same pointer to a `Query` can avoid recomputing the search query trigrams.

Encoding for `SearchToken`, `SearchResult`, and `UpdateToken` values gets accomplished by serializing some unexported structs with the `encoding/gob` package from the standard library. Future, more mature iterations of the implementation will either export the structs or provide a more detailed encoding specification to allow for interoperability.

The diff format of Emys is simple: it contains subsequent groups of bytes, where each group starts with either a minus sign (byte `0x2d`) or a plus sign (byte `0x2b`), indicating removal or addition, and then three Unicode characters (values of `rune` type in Go), denoting the trigram, follow. For example, the string

`-Hel-ell-llo+こんに+にちわ+んにち`

is a valid diff from `"Hello"` to `"こんにちわ"`. For convenience, the implementation offers a pair of functions, `Diff` and `ParseDiff`, to produce and parse diffs.

The current implementation uses in-memory state on both the client and the server. To allow the state to persist across different executions, the concrete

`Client` and `Server` types provide `State` and `LoadState` methods for dumping a serialized form of the current state and loading a previously serialized state, respectively. Additionally, since the client state contains sensitive information (i.e., all known trigrams and future internal search tokens), `Client.State` encrypts the serialized state. Correspondingly, `Client.LoadState` decrypts the state before loading it. In particular, the current implementation utilizes ChaCha20-Poly1305 [NL18] for encryption. The encryption key is derived from the master key k using the context string `client state dump encryption`. The random nonce, necessary for getting a randomized encryption, is freshly generated each time and prepended to the ciphertext. Moreover, the context string `client state dump` is put in the AAD. Future improvements to the implementation will likely provide better solutions for managing the state and choosing between in-memory storage and more structured options.

To guarantee exact theoretical security, the implementation currently resamples the integrity key, all authentication keys, and all encryption keys from $\mathbb{Z}_p$ and $\mathbb{Z}_p^d$, as needed, using rejection sampling from the randomness of the 256-bit keys, expanded with the XOF. Performing this process requires a non-trivial computational time, slowing down all key derivation operations. Note that such a heavy approach may not be strictly necessary to achieve the intended security level. Nevertheless, we choose to accept the performance cost, as we currently have no formal proof to justify its removal.

4.2.1 Algorithms and Parameter Choices

The implementation of Emys uses the following parameter values:

- The security parameter $\lambda = 256$ bits, which guarantees an adequate level of security. The detailed rationale for this value is below.
- The maximum number of files ℓ is configurable, with a maximum value of $2^{60} - 1$. Once selected, its value must not change across multiple executions. Otherwise, the correctness and security of the algorithms are not guaranteed.
- The number of bits η used to denote a file is configurable and derived from the maximum number of search query trigrams allowed, which has a maximum value of $2^{16} - 1$ (thus, the maximum value that η can take is 16). Similarly to ℓ, its value must not change across multiple executions.
- The value of p, used as the prime modulus for AHMAC and as the block modulus for AHE, is $2^{256} + 2^{96} - 1$. This value is a generalized Mersenne prime, and thus has properties that enable the implementation of fast modular reduction algorithms.
- The search threshold γ is configurable. However, instead of being expressed as an absolute value, it represents the percentage in $(0, 1]$ of unique trigrams in the search query required to match for including a file identifier in the result set, rounded down. It is permitted to change this value, both within the same execution and across different executions.

We selected the security parameter λ based on the fact that the security of the scheme depends on the total number of update operations through the birthday probability (see proof of Claim 3.21). As a worst-case scenario, we consider the maximum value for the number of supported files, $2^{60} - 1$ (i.e., more than a thousand million billion files), and assume an average of 2^{20} (i.e., roughly one million) updates for each file. Then, the upper bound to the birthday probability is approximately

$$\frac{2^{80} \cdot \left(2^{80} - 1\right)}{2 \cdot 2^{256}} \approx 2^{-97},$$

which can be considered negligible by today's standards.

The choices of the maximum values for ℓ and η guarantee both practical usability of the scheme and simplicity of implementation for the bit array (Section 3.1.1). Indeed, we have that

$$\left(2^{60} - 1\right) \cdot 16 < 2^{64},$$

so we can index every bit in the array simply using a 64-bit unsigned integer value.

The implementation of Emys utilizes BLAKE3 [O'C+24] for all its cryptographic algorithms by taking advantage of the various hashing modes it offers. In particular, the `derive_key` mode is used to instantiate F for all key derivation operations, and the `hash` mode is used to instantiate the extendable-output function E. To instantiate H_1 and H_2, the implementation employs the `keyed_hash` mode, and to domain-separate them, we use two different subkeys derived from k_{upd} as follows.

$$\boxed{\begin{aligned} k_{\text{upd},H_1} &\coloneqq F(k_{\text{upd}}, \texttt{h1}) \\ k_{\text{upd},H_2} &\coloneqq F(k_{\text{upd}}, \texttt{h2}) \end{aligned}}$$

Note that the server can derive these keys on its own, since no knowledge of the master key k or other context strings is required.

There are several reasons why we selected BLAKE3 for use in the implementation of Emys:

- It is fast in practice, since it can exploit general CPU features without needing algorithm-specific hardware instructions, and is highly parallelizable.
- It can instantiate multiple cryptographic primitives, reducing the need for many different algorithms and limiting the risk of misuse.
- It matches the security requirements stemming from the parameter choices. Specifically, the `keyed_hash` mode takes a 256-bit key, and the `hash` mode can produce up to $2^{64} - 1$ output bytes, which are more than enough to cover any supported size of the bit array.

Although not covered in this work, the future implementation of the file handling scheme (Section 3.2.2) will use BLAKE3 in `keyed_hash` mode for H_3 and instantiate the AEAD with XChaCha20-Poly1305 [Arc20]. Moreover, the metadata blobs and file contents will be padded using the Padmé deterministic padding scheme, which has minimal storage overhead [Nik+19], with ISO/IEC 7816-4 padding [Int20].

Alternative Choices. Following the discussion above, the security parameter λ and the maximum number of supported files ℓ should be chosen such that

$$\frac{2^{20}\ell \cdot (2^{20}\ell - 1)}{2 \cdot 2^{\lambda}} < 2^{-80}.$$

The choice of η can be arbitrary. To simplify the implementation, its value should be such that

$$\ell \cdot \eta < 2^{64}.$$

Finally, the threshold γ can also be arbitrary. However, note that any $\gamma > 2^{\eta} - 1$ makes no sense, since a search cannot match more trigrams than those actually searched.

The following table presents some example values that satisfy the requirements. In particular, notice that the values shown in the last row are appropriate for 32-bit systems, as $\ell \cdot \eta < 2^{32}$.

λ (bits)	ℓ	η
Security parameter	**Max supported files**	**Max search trigrams**
256	$2^{60} - 1$	16
224	$2^{40} - 1$	14
192	$2^{20} - 1$	12

Table 4.1: *Examples of sensible parameter values for Emys. The first row contains the values used in the implementation presented in this work.*

Regarding cryptographic algorithms, any state-of-the-art ones satisfying the security requirements imposed by the parameters are valid. In particular, some implementors may be interested in using FIPS 140 compliant algorithms [Nat19]. Go has native support for it built right into the standard library, and enabling FIPS 140-3 mode is as easy as setting an environment variable at compile time [The25d]. For this purpose, HMAC-SHA-256 [Nat08] may be used for the keyed-hash functions H_1 and H_2, HKDF-SHA-256 [KE10] with an empty salt may be employed to instantiate the key derivation function F, and SHAKE256 [Nat15] may be utilized for the extendable-output function E.

4.2.2 Experimental Results

We evaluated the performance of Emys using eight synthetic datasets, containing varying numbers of files, ranging from 10^3 to 10^6 files. For every dataset, each file contains 20 randomly selected words from the top 5000 most frequent English lemmas (which, as of August 28, 2025, correspond to 4380 unique words) [Wor25]. We ran the benchmarks on a desktop computer with a single Intel Core i7-4770K 3.50 GHz CPU, 32 GiB DIMM DDR3 2400 MHz of RAM, running on NixOS 25.05.

On the client, for each known trigram, the local state contains the update counter, which occupies up to 8 bytes, and the last internal search token, which is 32 bytes long. In particular, its size depends neither on the number of files nor on the maximum number of supported search trigrams. The server state, instead, depends on both of those, as well as on the amortized total number of updates (i.e., the minimum number of updates necessary to reconstruct all possible search results, given how previous search queries were able to compress the state). Until only new files get added and no search gets performed, the latter is equivalent to the total number of inserted files. For each update, the server state stores a 32-byte-long masked internal search token, an encrypted bit array of variable size depending on the maximum number of supported files and search trigrams, and its tag, which requires 33 bytes.

The table below reports the values for each of the eight synthetic datasets. Note that state sizes may not perfectly match the actual state dump sizes, as the encoding process may introduce minor bits of information to track the structure and other properties. The number of blocks per bit array is computed as $\lceil \eta\ell/\lambda \rceil$, as specified in Section 3.2.1, where $\lambda = 256$. The number of known trigrams is particular to the specific realizations of the randomly generated files.

Number of files	Search trigrams	Blocks per bit array	Known trigrams	Client state size	Server state size
10^3	3	8	3744	146.25 KiB	1.17 MiB
	15	16	3742	146.17 KiB	2.12 MiB
10^4	3	79	3899	152.30 KiB	9.94 MiB
	15	157	3897	152.23 KiB	19.50 MiB
10^5	3	782	3948	154.22 KiB	97.41 MiB
	15	1563	3949	154.26 KiB	194.49 MiB
10^6	3	7813	3966	154.92 KiB	975.43 MiB
	15	15625	3967	154.96 KiB	1.91 GiB

Table 4.2: *Sizing information about each of the synthetic dataset combinations. The values in the last two columns have been rounded up to two decimal places.*

We performed the benchmarks using Go's built-in benchmarking tool, running them for 20 iterations each. Client and server functions execute sequentially within the same process to avoid counting network transmission times and delays. Results can be reproduced by executing the following two commands from the root directory of the current implementation version, which prepare the random synthetic datasets and run the benchmarks, respectively.

```
$ go test -timeout=0 -run=Prepare ./internal/emys
$ go test -run='^$' -bench=. -count=20 ./internal/emys
```

To obtain statistical summaries of the results, we used the `benchstat` command[2]. Listing 4.2 reports the output of the command.

[2]https://pkg.go.dev/golang.org/x/perf/cmd/benchstat

```
goos: linux
goarch: amd64
pkg: interrato.dev/emys/internal/emys
cpu: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
                                   │  bench.txt   │
                                   │    sec/op    │
Search/files=1000/trigrams=3-8       597.6µ ±  3%
Search/files=1000/trigrams=15-8      2.208m ±  4%
Search/files=10000/trigrams=3-8      1.208m ±  5%
Search/files=10000/trigrams=15-8     4.126m ±  3%
Search/files=100000/trigrams=3-8     4.616m ±  3%
Search/files=100000/trigrams=15-8    21.96m ±  1%
Search/files=1000000/trigrams=3-8    45.75m ±  6%
Search/files=1000000/trigrams=15-8   240.6m ±  1%
Update/files=1000/trigrams=3-8       21.84m ±  4%
Update/files=1000/trigrams=15-8      25.07m ±  6%
Update/files=10000/trigrams=3-8      27.87m ± 10%
Update/files=10000/trigrams=15-8     30.41m ±  5%
Update/files=100000/trigrams=3-8     97.10m ±  1%
Update/files=100000/trigrams=15-8    189.8m ±  1%
Update/files=1000000/trigrams=3-8    895.0m ±  5%
Update/files=1000000/trigrams=15-8    1.888 ±  4%
geomean                              27.48m
```

Listing 4.2: *Statistical summaries of the raw benchmark results of the current implementation of* Emys *computed with the* `benchstat` *command.*

We call *search operation* the sequential execution of the `Client.Search`, `Server.ResolveSearch`, and `Client.OpenResult` methods, in that order. Similarly, we call *update operation* the sequential execution of `Client.Update` and `Server.ResolveUpdates`.

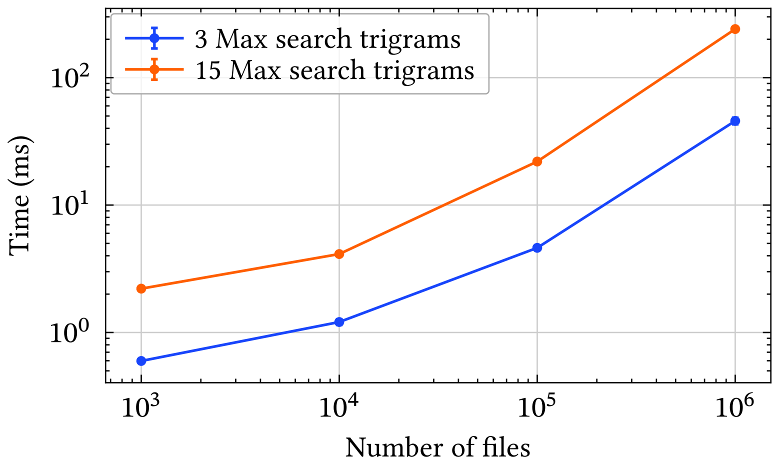

Figure 4.1: *End-to-end search time of* Emys*, including opening of search results and filtering by a threshold of 75% of the search query length. Timings do not include any network transmission.*

Figure 4.1 shows the results for the search operation on a log-log plot. We can see that search time scales sublinearly with the number of files, approaching a linear behavior after 10^5 files. The growth is expected, as the size of the index bit arrays increases with the maximum number of supported files. Indeed, bigger arrays imply more blocks for AHE and AHMAC, and thus require more modular operations. In particular, the server performs many modular additions to aggregate the ciphertexts, and the client also performs a considerable number of modular additions to aggregate the encryption keys. For the same reason, the search time when the maximum number of search trigrams allowed is larger is consistently higher.

The benchmark for the update operation did not take advantage of the optimization contributed by batch updates and performed only one update at a time. Figure 4.2 shows the results on a log-log plot. Similarly to the search operation, we can see that the update time scales almost linearly. However, we can also notice that update operations are about 10 times longer than search operations. By analyzing a CPU profile of the implementation, we could see that the computation of MAC tags dominates the running time. In particular, more than 80% of the time required to compute a tag is spent on doing modular multiplications. The reason for that is simple: the current implementation performs all modular operations regardless of the properties of their operands. Specifically, we could significantly reduce the amount of computing required by exploiting the fact that p is a generalized Mersenne prime. The benchmark uses file updates of 20 randomly selected words, corresponding to an average of approximately 138 unique trigrams per update, and thus to 138 MAC tag computations per update, which explains the slowdown. Finally, we can explain the gap for different values of the maximum number of search trigrams because of the increase in size of the index bit arrays, as is the case for search operations.

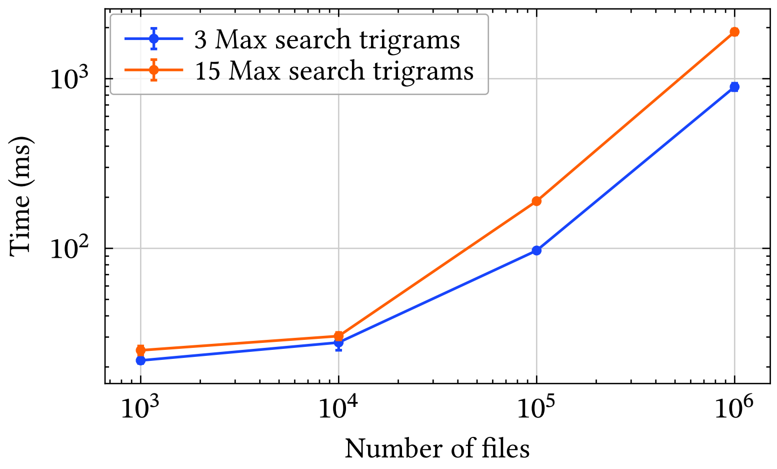

Figure 4.2: *End-to-end update time of Emys, using randomly generated file changes of 20 words. Timings do not include any network transmission.*

5 Conclusions

In this work, we have addressed the effort of bringing searchable symmetric encryption to real-world applications that directly impact individual users, closing the gap between research advancements and practical usability. First, in Chapter 2, we give formal definitions of SSE, its security notions, and other security properties such as forward privacy and backward privacy. In particular, we have also formalized a new type of backward privacy, termed Type-I_B, by clarifying and unifying others introduced in previous literature. We arrange definitions in broad generality to make them flexible enough to describe various schemes that offer different features. In Section 2.3.1 and Section 2.3.2, we present two new cryptographic constructions that are homomorphic under modular blockwise addition, and prove their security. In Chapter 3, we present our novel SSE scheme, which we call Emys, building on the new constructions and existing research literature. The scheme offers full-text fuzzy search and verifiability of search results. Moreover, Emys is a dynamic scheme: it allows adding new files to the search index and updating or removing existing ones. In pursuing the goal of bringing SSE to real-world applications, we also describe a file handling scheme as a drop-in replaceable mechanism for managing actual file contents and metadata. In Section 3.3, we have formally proved, in the random oracle model, the adaptive security of the DSSE scheme and its verifiability property. Further, we give formal descriptions and proofs of its leakage functions. In Chapter 4, we first introduce a programming interface, providing again a general perspective to promote adoption from as many designs as possible, and then implement the Emys DSSE scheme by adhering to said interface. Benchmarks suggest that the current scheme implementation is already usable in small to medium-scale systems.

We now recall the three main problems in the current state of the field of searchable symmetric encryption (see Section 1.2), and discuss the advancements brought by this work. First, we have a lack of confidence in the security of SSE in general and a high prevalence of schemes relying on the honest-but-curious threat model. The formal security proofs contributed are meticulous and provide many details that facilitate the analysis of potential areas of improvement, which in turn delineate an explicit security boundary. Together with high levels of forward and backward privacy, we can place significant confidence in the security of EMYS. At the same time, cryptanalysis would undoubtedly help strengthen it. Further, the threat model of EMYS assumes a malicious server by design, and the scheme satisfies a strong notion of verifiability to thwart malicious servers engaging in dishonest behavior, enriching the landscape of verifiable schemes. Then, we have the poor usability of most schemes for developing a wide range of real-world applications, such as email services and cloud-based note-taking applications. Offering tunable fuzzy search, EMYS narrows this gap. In fact, this approach is quite flexible and can be adapted to diverse contexts, perhaps even allowing end users to select the typo-tolerance threshold inside applications. Finally, we have the lack of easy-to-use and hard-to-misuse software libraries implementing SSE schemes. The programming interface has been carefully designed precisely for this purpose: its simplicity enables developers to interact with SSE schemes without needing to understand the underlying workings of these and the subtle details of our field, and its flexibility allows implementors to adopt it for a wide variety of schemes. The implementation of EMYS and its ongoing development contribute to reducing the problem. Ultimately, that will hopefully serve as an example for many others in the future, helping the ecosystem to grow and allowing real-world applications to benefit from the advantages of searchable symmetric encryption.

Reference	Year	Search type	FP	BP	Verifiability	Impl.
Π^{dyn} [Cas+14]	2014	Single-keyword	×	×	×	○
Σοφος [Bos16]	2016	Single-keyword	✓	×	×	◔[1]
$\text{Diana}_{\text{del}}$ [BMO17]	2017	Single-keyword	✓	Type-III	×	◔[1]
`FBDSSE-CQ` [Zuo+20]	2020	Conjunctive	✓	Type-I_B	×	○
Aura [Sun+21]	2021	Single-keyword	✓	Type-II	×	◔[2]
EXIPNOS [AMZ21]	2021	Single-keyword	✓	Type-II	✓	○
FB-VDSSE [Zha+25]	2024	Single-keyword	✓	Type-II	✓	○
EMYS (this work)	2025	Fuzzy	✓	Type-I_B	✓	◕

Table 5.1: *Comparison between this work and other dynamic SSE schemes from the literature. FP stands for Forward Privacy, while BP stands for Backward Privacy. In the last column, we use Harvey balls to denote the availability and status of each scheme implementation:* ○ *indicates that the implementation is not available,* ◔ *indicates that the implementation is available,* ◑ *indicates that the implementation is available and actively maintained (i.e., the last activity is within 2 years),* ◕ *indicates that the implementation is available, actively maintained and easy-to-use, and* ● *indicates that the implementation is also production-ready.*

[1]https://github.com/OpenSSE/opensse-schemes
[2]https://github.com/MonashCybersecurityLab/Aura

Table 5.1 compares EMYS with other dynamic SSE schemes. The lack of examples with support for fuzzy search is due to the fact that we only found either static schemes, such as VFSA in [Ton+23], or schemes with unclear security and privacy properties (i.e., missing the analysis of CQA2 security, forward privacy, and backward privacy), such as EliMFS in [Che+20], M2FS in [Liu+21], and the scheme of [Fu+16]. Notice how EMYS offers good usability without sacrificing security and privacy.

Nonetheless, we inevitably make some trade-offs. For example, we do not employ any search pattern or result pattern protection measures. Moreover, although fuzzy search is a good match for applications targeted directly at individual users, more structured search types, such as boolean search, may be preferable in other contexts, like relational databases.

The following section concludes this work by discussing its limitations, some theoretical open issues, and possible future improvements for EMYS.

5.1 Open Issues and Future Work

The multi-user setting, although not widely addressed in the literature, is notably lacking in this work. Particularly, multi-user would enable some nice usability features, such as file sharing and the ability to search the contents of other users' authorized files. This aspect is also missing in the design of the programming interface, partially hindering adoption. Despite that, extending the interface without introducing backward-incompatible changes should be possible and reasonably easy. Concerning security proofs, however carefully they have been carried out, they are still manual proofs, lacking computer-based formal verification, which would give us stronger guarantees. Besides, this work lacks the testing of EMYS against well-known attacks targeting SSE schemes.

Regarding the file handling scheme proposed, we have not provided formal security proofs. In particular, since file identifiers are deterministic even in their opaque fashion that servers can see, the scheme intuitively leaks some form of access pattern. Hence, extensive analysis of possible countermeasures is lacking. Furthermore, the current implementation does not include the file handling scheme at all. Therefore, we provide no benchmark for its performance. Nevertheless, if it turns out to be inadequate, it is always possible to replace it with a different file handling mechanism having a suitable input interface.

On the theoretical side of searchable symmetric encryption, an interesting open problem is that of stronger backward privacy. Specifically, we ask ourselves whether it is possible to achieve a notion of backward privacy stronger than both Type-I$_A$ and Type-I$_B$, for example, one such that $\mathcal{L}_{\mathsf{Search}}(q) = \mathcal{L}''(\mathsf{ap}(q), |\mathsf{Updates}(q)|)$, where $\mathcal{L}''$ is a stateless function. Another topic is that of pattern leakages. For the search pattern in particular, it might be of interest to explore if total query indistinguishability is feasible or even meaningful. Finally, exploring the possibility of defining holistic notions of security for SSE that more precisely incorporate the attacker's knowledge, starting from the work of Damie et al. [Dam+25], is quite compelling.

About possible improvements for Emys, proof of Claim 3.21 highlighted the opportunity to avoid leaking the search pattern entirely. Further research is needed to determine whether this is indeed possible and a viable path to pursue, considering its potential impacts on performance and other usability traits. On a practical level, Emys' current implementation leaves several opportunities for improvement. Leveraging the properties of the modulus, we could significantly improve the performance of all modular operations. In particular, the polynomial MAC implementation can benefit from the work of Degabriele et al. [Deg+24]. Lastly, storage options, especially for the server state, are a crucial component that needs to be delivered. The current in-memory state has a high risk of data loss and can quickly become unusable for large datasets.

References

[Goo21] Google LLC, “Google Workspace Encryption [White Paper],” 2021. Accessed: Jul. 08, 2025. [Online]. Available: https://web.archive.org/web/20250708110556/https://services.google.com/fh/files/misc/google-workspace-encryption-wp.pdf

[Dro25] Dropbox, Inc., “How Dropbox keeps your files secure.” Accessed: Jul. 08, 2025. [Online]. Available: https://web.archive.org/web/20250708122909/https://help.dropbox.com/security/how-security-works

[Gen09] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *41st ACM STOC*, M. Mitzenmacher, Ed., ACM Press, May 2009, pp. 169–178. doi: 10.1145/1536414.1536440.

[EKR22] D. Evans, V. Kolesnikov, and M. Rosulek, “A Pragmatic Introduction to Secure Multi-Party Computation.” Accessed: Aug. 12, 2024. [Online]. Available: https://web.archive.org/web/20240812213844/https://securecomputation.org/docs/pragmaticmpc.pdf

[GO96] O. Goldreich and R. Ostrovsky, “Software Protection and Simulation on Oblivious RAMs,” *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996, doi: 10.1145/233551.233553.

[LN18] K. G. Larsen and J. B. Nielsen, “Yes, There is an Oblivious RAM Lower Bound!,” in *CRYPTO 2018, Part II*, H. Shacham and A. Boldyreva, Eds., in LNCS, vol. 10992. Springer, Cham, Aug. 2018, pp. 523–542. doi: 10.1007/978-3-319-96881-0_18.

[HS21] S. Halevi and V. Shoup, “Bootstrapping for HElib,” *Journal of Cryptology*, vol. 34, no. 1, p. 7, Jan. 2021, doi: 10.1007/s00145-020-09368-7.

[SWP00] D. X. Song, D. Wagner, and A. Perrig, “Practical Techniques for Searches on Encrypted Data,” in *2000 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 2000, pp. 44–55. doi: 10.1109/SECPRI.2000.848445.

[Cur+06] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, “Searchable symmetric encryption: improved definitions and efficient constructions,” in *ACM CCS 2006*, A. Juels, R. N. Wright, and S. De Capitani di Vimercati, Eds., ACM Press, Oct. 2006, pp. 79–88. doi: 10.1145/1180405.1180417.

[KPR12] S. Kamara, C. Papamanthou, and T. Roeder, “Dynamic searchable symmetric encryption,” in *ACM CCS 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM Press, Oct. 2012, pp. 965–976. doi: 10.1145/2382196.2382298.

[Bos16] R. Bost, "Σοφος: Forward Secure Searchable Encryption", in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM Press, Oct. 2016, pp. 1143–1154. doi: 10.1145/2976749.2978303.

[BMO17] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM Press, Oct. 2017, pp. 1465–1482. doi: 10.1145/3133956.3133980.

[Zha+25] C. Zhao *et al.*, "Efficient Verifiable Dynamic Searchable Symmetric Encryption With Forward and Backward Security," *IEEE Internet Things J.*, vol. 12, no. 3, pp. 2633–2645, 2025, doi: 10.1109/JIOT.2024.3470772.

[Sun+15] W. Sun, X. Liu, W. Lou, Y. T. Hou, and H. Li, "Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data," in *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, IEEE, 2015, pp. 2110–2118. doi: 10.1109/INFOCOM.2015.7218596.

[MS13] T. Moataz and A. Shikfa, "Boolean symmetric searchable encryption," in *ASIACCS 13*, K. Chen, Q. Xie, W. Qiu, N. Li, and W.-G. Tzeng, Eds., ACM Press, May 2013, pp. 265–276. doi: 10.1145/2484313.2484347.

[Zuo+18] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward (and Backward) Security," in *ESORICS 2018, Part II*, J. López, J. Zhou, and M. Soriano, Eds., in LNCS, vol. 11099. Springer, Cham, Sep. 2018, pp. 228–246. doi: 10.1007/978-3-319-98989-1_12.

[Wan+12] C. Wang, N. Cao, K. Ren, and W. Lou, "Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data," *IEEE Trans. Parallel Distributed Syst.*, vol. 23, no. 8, pp. 1467–1479, 2012, doi: 10.1109/TPDS.2011.282.

[Moa+18] T. Moataz, I. Ray, I. Ray, A. Shikfa, F. Cuppens, and N. Cuppens, "Substring search over encrypted data," *J. Comput. Secur.*, vol. 26, no. 1, pp. 1–30, 2018, doi: 10.3233/JCS-14652.

[Ton+23] Q. Tong, Y. Miao, J. Weng, X. Liu, K.-K. R. Choo, and R. H. Deng, "Verifiable Fuzzy Multi-Keyword Search Over Encrypted Data With Adaptive Security," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 5, pp. 5386–5399, 2023, doi: 10.1109/TKDE.2022.3152033.

[BFP16] R. Bost, P.-A. Fouque, and D. Pointcheval, "Verifiable Dynamic Symmetric Searchable Encryption: Optimality and Forward Security." [Online]. Available: https://eprint.iacr.org/2016/062

[GPT23] Z. Gui, K. G. Paterson, and T. Tang, "Security Analysis of MongoDB Queryable Encryption," in *USENIX Security 2023*, J. A. Calandrino and C. Troncoso, Eds., USENIX Association, Aug. 2023, pp. 7445–7462. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/gui

[Pop+11] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds., ACM, 2011, pp. 85–100. doi: 10.1145/2043556.2043566.

[Pap+14] V. Pappas *et al.*, "Blind Seer: A Scalable Private DBMS," in *2014 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 2014, pp. 359–374. doi: 10.1109/SP.2014.30.

[Mon23] MongoDB, Inc., "Queryable Encryption - Database Manual v7.0 - MongoDB Docs." Accessed: Jul. 09, 2025. [Online]. Available: https://web.archive.org/web/20250709093624/https://www.mongodb.com/docs/v7.0/core/queryable-encryption/

[Wir18] Wired, "Pixek App Encrypts Your Photos From Camera to Cloud." Accessed: Jul. 09, 2025. [Online]. Available: https://web.archive.org/web/20250709095630/https://www.wired.com/story/pixek-app-encrypts-photos-from-camera-to-cloud/

[Dam+25] M. Damie, J.-B. Leger, F. Hahn, and A. Peter, "Revisiting the Attacker's Knowledge in Inference Attacks Against Searchable Symmetric Encryption," in *Applied Cryptography and Network Security - 23rd International Conference, ACNS 2025, Munich, Germany, June 23-26, 2025, Proceedings, Part II*, M. Fischlin and V. Moonsamy, Eds., in Lecture Notes in Computer Science, vol. 15826. Springer, 2025, pp. 370–399. doi: 10.1007/978-3-031-95764-2_15.

[Ros21] M. Rosulek, "The Joy of Cryptography." [Online]. Available: https://joyofcryptography.com/

[Li+24] F. Li, J. Ma, Y. Miao, X. Liu, J. Ning, and R. H. Deng, "A Survey on Searchable Symmetric Encryption," *ACM Comput. Surv.*, vol. 56, no. 5, pp. 1–42, 2024, doi: 10.1145/3617991.

[Zuo+21] C. Zuo, S. Lai, X. Yuan, J. K. Liu, J. Shao, and H. Wang, "Searchable Encryption for Conjunctive Queries with Extended Forward and Backward Privacy." [Online]. Available: https://eprint.iacr.org/2021/1585

[LH25] T. Le and T. Hoang, "Hermes: Efficient and Secure Multi-Writer Encrypted Database," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds., IEEE, 2025, pp. 2865–2884. doi: 10.1109/SP61157.2025.00184.

[Liu+13] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan, "Search Pattern Leakage in Searchable Encryption: Attacks and New Construction." [Online]. Available: https://eprint.iacr.org/2013/163

[OK21] S. Oya and F. Kerschbaum, "Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption," in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, Aug. 2021, pp. 127–142. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/oya

[DHP21] M. Damie, F. Hahn, and A. Peter, "A Highly Accurate Query-Recovery Attack against Searchable Encryption using Non-Indexed Documents," in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, Aug. 2021, pp. 143–160. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/damie

[Xu+23] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-Abuse Attacks Against Forward and Backward Private Searchable Symmetric Encryption," in *ACM*

CCS 2023, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds., ACM Press, Nov. 2023, pp. 3003–3017. doi: 10.1145/3576915.3623085.

[Sha+21] Z. Shang, S. Oya, A. Peter, and F. Kerschbaum, "Obfuscated Access and Search Patterns in Searchable Encryption," in *NDSS 2021*, The Internet Society, Feb. 2021. doi: 10.14722/ndss.2021.23041.

[Du+23] R. Du, C. Ma, M. Li, and Z. Zhang, "Pattern-protecting Dynamic Searchable Symmetric Encryption Based on Differential privacy," in *IEEE International Conference on Web Services, ICWS 2023, Chicago, IL, USA, July 2-8, 2023*, C. A. Ardagna, B. Benatallah, H. Bian, C. K. Chang, R. N. Chang, J. Fan, G. C. Fox, Z. Jin, X. Liu, H. Ludwig, M. Sheng, and J. Yang, Eds., IEEE, 2023, pp. 626–637. doi: 10.1109/ICWS60048.2023.00081.

[SPS14] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in *NDSS 2014*, The Internet Society, Feb. 2014. doi: 10.14722/ndss.2014.23298.

[ZKP16] Y. Zhang, J. Katz, and C. Papamanthou, "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption," in *USENIX Security 2016*, T. Holz and S. Savage, Eds., USENIX Association, Aug. 2016, pp. 707–720. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[Zuo+19] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy," in *ESORICS 2019, Part II*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., in LNCS, vol. 11736. Springer, Cham, Sep. 2019, pp. 283–303. doi: 10.1007/978-3-030-29962-0_14.

[Zuo+20] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and G. Wei, "Forward and Backward Private Dynamic Searchable Symmetric Encryption for Conjunctive Queries." [Online]. Available: https://eprint.iacr.org/2020/1357

[WC81] M. N. Wegman and L. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," *Journal of Computer and System Sciences*, vol. 22, pp. 265–279, 1981.

[CW79] L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *J. Comput. Syst. Sci.*, vol. 18, no. 2, pp. 143–154, 1979, doi: 10.1016/0022-0000(79)90044-8.

[Tho15] M. Thorup, "High Speed Hashing for Integers and Strings," *CoRR*, 2015, [Online]. Available: http://arxiv.org/abs/1504.06804

[Die+92] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger, "Polynomial Hash Functions Are Reliable (Extended Abstract)," in *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, W. Kuich, Ed., in Lecture Notes in Computer Science, vol. 623. Springer, 1992, pp. 235–246. doi: 10.1007/3-540-55719-9_77.

[The24] The Unicode Consortium, "C0 Controls and Basic Latin." Accessed: Jul. 20, 2025. [Online]. Available: https://unicode.org/charts/PDF/U0000.pdf

[Zhu+25] X. Zhu, J. Zhou, Y. Dai, P. Shen, S. K. Kermanshahi, and J. Hu, "A Verifiable and Efficient Symmetric Searchable Encryption Scheme for Dynamic Dataset

With Forward and Backward Privacy," *IEEE Trans. Dependable Secur. Comput.*, vol. 22, no. 3, pp. 2741–2755, 2025, doi: 10.1109/TDSC.2024.3521423.

[Por11] T. Pornin, "What is the "Random Oracle Model" and why is it controversial?." Accessed: Aug. 01, 2025. [Online]. Available: https://crypto.stackexchange.com/a/880

[BR04] M. Bellare and P. Rogaway, "Code-Based Game-Playing Proofs and the Security of Triple Encryption." [Online]. Available: https://eprint.iacr.org/2004/331

[The25] The Go Authors, "The Go Programming Language." Accessed: Aug. 21, 2025a. [Online]. Available: https://go.dev/

[Wik25] Wikipedia contributors, "Composition over inheritance." Accessed: Aug. 21, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Composition_over_inheritance

[DPL24] K. R. Davis, B. Peabody, and P. Leach, "Universally Unique IDentifiers (UUIDs)." [Online]. Available: https://www.rfc-editor.org/info/rfc9562

[The25] The Go Authors, "Go cryptography packages." Accessed: Aug. 22, 2025b. [Online]. Available: https://pkg.go.dev/crypto

[The25] The Go Authors, "Supplementary Go cryptography packages." Accessed: Aug. 22, 2025c. [Online]. Available: https://pkg.go.dev/golang.org/x/crypto

[Val19] F. Valsorda, "Cryptography Principles." Accessed: Aug. 22, 2025. [Online]. Available: https://golang.org/design/cryptography-principles

[Par22] Paragon Initiative Enterprises, "PASETO specification - Authentication Padding." Accessed: Aug. 22, 2025. [Online]. Available: https://github.com/paseto-standard/paseto-spec/blob/af79f25908227555404e7462ccdd8ce106049469/docs/01-Protocol-Versions/Common.md#authentication-padding

[Goo25] Google LLC, "Confused deputy attack example - Additional authenticated data." Accessed: Aug. 22, 2025. [Online]. Available: https://web.archive.org/web/20250709035902/https://cloud.google.com/kms/docs/additional-authenticated-data#confused_deputy_attack_example

[NL18] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." [Online]. Available: https://www.rfc-editor.org/info/rfc8439

[O'C+24] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "BLAKE3." Accessed: Aug. 23, 2025. [Online]. Available: https://c2sp.org/BLAKE3

[Arc20] S. Arciszewski, "XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305," Internet Engineering Task Force, Jan. 2020. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/

[Nik+19] K. Nikitin, L. Barman, W. Lueks, M. Underwood, J.-P. Hubaux, and B. Ford, "Reducing Metadata Leakage from Encrypted Files and Communication with PURBs," *PoPETs*, vol. 2019, no. 4, pp. 6–33, Oct. 2019, doi: 10.2478/popets-2019-0056.

[Int20] "Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange (ISO/IEC 7816-4:2020)," Geneva, Switzerland, May 2020. [Online]. Available: https://www.iso.org/standard/77180.html

[Nat19] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," Washington, D.C., 2019. doi: 10.6028/NIST.FIPS.140-3.

[The25] The Go Authors, "FIPS 140-3 Compliance." Accessed: Aug. 23, 2025d. [Online]. Available: https://go.dev/doc/security/fips140

[Nat08] National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code (HMAC)," Washington, D.C., 2008. doi: 10.6028/NIST.FIPS.198-1.

[KE10] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." [Online]. Available: https://www.rfc-editor.org/info/rfc5869

[Nat15] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," Washington, D.C., 2015. doi: 10.6028/NIST.FIPS.202.

[Wor25] "Word frequency data." Accessed: Aug. 28, 2025. [Online]. Available: https://web.archive.org/web/20250828163206/https://www.wordfrequency.info/samples.asp

[Cas+14] D. Cash *et al.*, "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation," in *NDSS 2014*, The Internet Society, Feb. 2014. doi: 10.14722/ndss.2014.23264.

[Sun+21] S.-F. Sun *et al.*, "Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy," in *NDSS 2021*, The Internet Society, Feb. 2021. doi: 10.14722/ndss.2021.24162.

[AMZ21] N. Aaraj, C. Marcolla, and X. Zhu, "Exipnos: An Efficient Verifiable Dynamic Symmetric Searchable Encryption Scheme with Forward and Backward Privacy," in *INDOCRYPT 2021*, A. Adhikari, R. Küsters, and B. Preneel, Eds., in LNCS, vol. 13143. Springer, Cham, Dec. 2021, pp. 487–509. doi: 10.1007/978-3-030-92518-5_22.

[Che+20] J. Chen *et al.*, "EliMFS: Achieving Efficient, Leakage-Resilient, and Multi-Keyword Fuzzy Search on Encrypted Cloud Data," *IEEE Trans. Serv. Comput.*, vol. 13, no. 6, pp. 1072–1085, 2020, doi: 10.1109/TSC.2017.2765323.

[Liu+21] Q. Liu, Y. Peng, J. Wu, T. Wang, and G. Wang, "Secure Multi-keyword Fuzzy Searches With Enhanced Service Quality in Cloud Computing," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 2, pp. 2046–2062, 2021, doi: 10.1109/TNSM.2020.3045467.

[Fu+16] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, "Toward Efficient Multi-Keyword Fuzzy Search Over Encrypted Outsourced Data With Accuracy Improvement," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 12, pp. 2706–2716, 2016, doi: 10.1109/TIFS.2016.2596138.

[Deg+24] J. P. Degabriele, J. Gilcher, J. Govinden, and K. G. Paterson, "SoK: Efficient Design and Implementation of Polynomial Hash Functions over Prime Fields," in *2024 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 2024, pp. 3128–3146. doi: 10.1109/SP54263.2024.00132.

埋蔵暗号

Colophon

This document was typeset using Typst. Body text is set in *Libertinus Serif*; sans-serif text is set in *Libertinus Sans*. Monospaced text uses *DejaVu Sans Mono* and *M PLUS 1 Code* (for CJK glyphs). Mathematics is set in *STIX Two Math*.