PADERBORN UNIVERSITY
The University for the Information Society

Cryptography for the Crowd

—

A Study of Cryptographic Schemes with Applications to Crowd Work

Nils Löken

Paderborn, 2019-09-20

A dissertation submitted to the
Department of Computer Science, Paderborn University

in partial fullfilment of the requirements for the degree of
Doktor der Naturwissenschaften
(doctor rerum naturalium)

accepted upon recommendation of

Prof. Dr. Johannes Blömer,
Paderborn University

Prof. Dr. Tibor Jager,
University of Wuppertal

defended 2019-12-17

Acknowledgements

I thank my advisor Prof. Dr Johannes Blömer for the opportunity to work in his research group, and the freedom to explore and pursue my own research interests.

I have conducted my research at Paderborn University while working on two projects “Securing the Financial Cloud” and “Digital Future.” These projects gave me motivation and usage scenarios for my research, while also allowing me to meet interesting people that enabled me to expand my knowledge and see new connections between scientific disciplines. I thank the numerous people from these projects for the insights they provided.

I furthermore thank my colleagues for fruitful discussions, professional or otherwise. Their input has helped me view my research from different angles and thus develop a deeper understanding of the topics I have worked on. I particularly thank my co-authors Dr. Peter Günther, Dr. Volker Krummel, and Dr. Jakob Juhnke, the political action committee Dr. Sascha Brauer and Jan Bobolz, the marketing man Fabian Eidens, and the new guys Henrik Bröcher and Laurens Porzenheim. I thank Claudia Jahn for her invaluable administrative work and her persistence.

Last, but not least, I thank my family for their support. Without them, I would not have been able to pursue my studies and research in computer science and cryptography.

Abstract

In this thesis, we investigate three areas of cryptography: searchable encryption, reputation systems, and cryptographic cloud architectures. We mostly investigate these areas from fairly distinct perspectives. However, we find applications for the three concepts in crowd work, and suggest combining aspects of the concepts into a single cryptographic scheme tailored to the crowd work setting.

Regarding searchable encryption, we formally define the desired security notions and provide schemes satisfying these notions. Among others, we propose the first searchable encryption scheme that simultaneously provides fine-grained access control in a multi-reader and multi-writer setting, is dynamic, forward private, and verifiable.

Concerning reputation systems, we propose a novel security notion, a variant of rater anonymity, motivated by the crowd work scenario. We then propose a very simple reputation system that satisfies our anonymity notion, as well as the security requirements typically imposed on reputation systems.

With respect to cloud architectures, we address the trust and security issues in the cloud at the most fundamental level, the internal structure of the cloud. Our proposal relies on hardware security as a trust anchor for the cloud and enables parties to outsource computations on their secret cryptographic keys to the cloud without cloud service providers learning these keys.

Zusammenfassung

In dieser Arbeit untersuchen wir drei Bereiche der Kryptographie: Searchable Encryption, Reputationssysteme und krypographische Cloud-Architekturen. Diese Themen haben unterschiedliche Anwendungsbereiche. Dennoch finden wir Anwendungen der untersuchten Konzepte auf Crowd Work und schlagen vor, Teile der untersuchten Konzepte zu einem auf Crowd Work zugeschnittenen kryptographischen Verfahren zu vereinen.

Hinsichtlich Searchable Encryption definieren wir formal die gewünschten Sicherheitseigenschaften und schlagen Verfahren vor, die unseren Definitionen genügen. Unter anderem schlagen wir das erste Searchable-Encryption-Verfahren vor, das gleichzeitig feingranulare Zugriffskontrolle bei mehreren Lesern und Schreibern ermöglicht, dynamisch und forward-private ist, und verifizierbare Suchergebnisse bietet.

Für Reputationssysteme schlagen wir einen neuen Sicherheitsbegriff, eine Variante von Bewerter-Anonymität, vor, die im Crowd-Work-Szenario motiviert ist. Wir entwickeln ein einfaches Reputationssystem, das sowohl unsere Anonymitätsdefinition, als auch andere für Reputationssysteme typische Sicherheitseigenschaften erfüllt.

Hinsichtlich Cloud-Architekturen adressieren wir Sicherheits- und Vertrauenshemmnisse in der Cloud durch deren grundlegenden inneren Aufbau. Unser Vorschlag basiert auf Hardware-Sicherheit als Vertrauensanker für die Cloud und ermöglicht es, Berechnungen auf kryptographischen Schlüsseln in die Cloud auszulagern, ohne dass Cloud-Betreiber diese Schlüssel lernen.

Contents

List of Acronyms

AAD	append-only authenticated dictionary
ABE	attribute-based encryption
CCA	chosen ciphertext attack
CCS	cryptographic computation service
CPA	chosen plaintext attack
EUF-CMA	existential unforgeability under adaptively chosen message attack
IBE	identity-based encryption
HSM	hardware security module
MA-ABE	multi-authority attribute-based encryption
MUSE	multi-user searchable encryption
OKS	organizational key store
PTR	platform transaction reference
PPT	probabilistic polynomial time
PRF	pseudorandom function
SSE	searchable symmetric encryption
SUF-CMA	strong existential unforgeability under adaptively chosen message attack

List of Figures

Part I

Introduction and Foundations

1 Introduction

In an increasingly interconnected world, cryptography provides many tools to protect legitimate goals and interests from adversarial parties. Private data should be protected from becoming public knowledge. It should be possible to check that data originates from a claimed source and that the data has not been altered, for example, during online banking. Only people who have bought a certain product in an online shop should be able to rate that product and ratings, once made, should be protected from malicious modification. Also, raters should be accountable for their reviews.

The first requirement, keeping data private, is covered by encryption schemes. The second and third requirements, authenticity and integrity, are covered by signature schemes. The fourth, fifth, and sixth requirements are covered by reputation systems, which can rely on signature schemes to hold raters accountable and to prevent rating modification.

The given usage examples of encryption schemes, signature schemes and reputation systems show that cryptographic schemes can have a profound impact on daily life. These schemes protect people from being blackmailed, from having money stolen by attackers claiming to be the victim's bank, or from being scammed into buying products from unreliable vendors. The given examples also show that cryptographic schemes cannot only be used individually, but can be combined and build upon each other. The examples highlight that cryptography is used in a context. One such context relevant for our work is crowd work.

Crowd work. Crowd work is a work model that has been claimed to have significant economic impact in the future. It is a particular kind of crowdsourcing, which according to Estellés-Arolas and González-Ladrón-De-Guevara [EG12], is defined as "a type of participative online activity in which . . . [an individual or organization, called requester] proposes to a group [called crowd] . . . , via a flexible open call, the voluntary undertaking of a task. The undertaking of the task, . . . in which the crowd should participate bringing their . . . [resources] always entails mutual benefit. The user will receive . . . [a reward], while the . . . [requester] will obtain and utilize to their advantage that what the user has brought to the venture . . . " Following the definition, in crowd work, the crowd provides resources in the form of labor and receives a monetary reward in return.

Context of this thesis. Crowd work is subject to research in the research program Digital Future that the author of this thesis participates in. In this thesis, we discuss three main areas of cryptographic research that are not necessarily investigated with a focus on crowd work. After all, cryptography often separates cryptographic schemes from their usage contexts to get general schemes that are applicable in other contexts as well. However, we tie the individual classes of cryptographic schemes together towards the end of the thesis and in the context of crowd work.

Contribution. The three main research areas covered in this thesis are searchable encryption, reputation systems, and cryptographic cloud architectures. These topics are covered by four publications to which the author has made substantial contributions.

At the International Conference on Availability, Reliability and Security, ARES 2017, we have presented some of our work on searchable encryption [Lök17]. We have expanded upon that work and have presented the results at the Foundations and Practice of Security conference, FPS 2019 [BL19a]. At the earlier 2017 edition of FPS, we have presented our initial work on cloud architectures [Blö+17]. Cloud architectures and searchable encryption have been jointly addressed at ARES's 2018 edition; the presented work evolves the previously covered cloud architecture and expands on its features [BL18]. Reputation systems in the context of crowd work have been presented at the International Workshop on Security and Trust Management, STM 2019 [BL19b]. In addition to previously published material, this thesis also contains previously unpublished work on searchable encryption.

Outline. We precede our discussion of the classes of cryptographic schemes by briefly reviewing some of the foundations and approaches of modern cryptography, and by presenting some cryptographic building blocks used throughout this thesis. The material can be found in the remainder of Part I. Our research on searchable encryption can be found in Part II. Parts III and IV cover our research on reputation systems and cloud architectures, respectively. Each part features a more detailed introductions to the class of cryptographic scheme discussed within that part. Finally, in Part V, we bring our results and intuitions from the various research areas together, and address the application of our results to crowd work.

2 Basic Concepts and Notation

In this chapter we briefly review three fundamental concepts in modern cryptography: probabilistic polynomial time algorithms, security parameters, and security experiments. Alongside these concepts, we familiarize ourselves with our notation.

Algorithms and randomness. Throughout this thesis, we consider algorithms that run in probabilistic polynomial time (PPT). That means, the runtime of the algorithm, expressed as a function of the length of the input to the algorithm, is bounded by some polynomial in the length of the algorithm's input. An algorithm running in PPT also means, that the algorithm has access to randomness, allowing it to draw values according to any computable probability distribution.

In our context, this probability distribution is typically a uniform distribution, and we denote the assignment of a value uniformly sampled from a set X to a variable y as $y \leftarrow_\$ X$. In contrast, the assignment of a fixed value x to variable y is denoted as $y \leftarrow x$. We use the same notation to denote the assignment of a value that is the result of a call to some PPT algorithm.

Security parameter λ. Modern provably secure cryptography attempts to quantify the degree of security achieved by concrete instantiations of cryptographic schemes. For quantification purposes, a security parameter λ is used. When instantiating a cryptographic scheme, the parameter is passed to the operation of the scheme that is supposed to be executed first. Often, those operations are called Setup or KeyGen and serve the purpose of deriving other system parameters and cryptographic keys such that an instantiation of the scheme using those parameters and keys provides the desired degree of security.

A popular interpretation of the security parameter is that it represents the length (in bits) of a cryptographic key. This is what is meant when a scheme is referred to as providing 80 or 128 bits of security. However, it must be noted that the intuition "security parameter = key length" only holds for a very limited range of schemes. Oftentimes the actual key or system parameter length is a multiple of the security parameter.

Since we want cryptographic schemes to be practical, we require the algorithms that make up a scheme to run in PPT. However, if we stick with the flawed intuition of "security parameter = key length," we immediately see that no PPT algorithm, given the security parameter λ as input, coded in binary, can output anything of length λ. Simply producing that output requires time exponential in the length of λ. In order to circumvent this, the security parameter is passed in unary, denoted as 1^λ. Again relying on the intuition of "security parameter = key length," the other algorithms that make up a cryptographic scheme, by using keys of the given length, can run in time polynomial in 1^λ.

Quantifying security. The security of cryptographic schemes is assessed through security experiments. In these experiments, an algorithm denoted "adversary" attempts to compute something that depends on what security notion the experiment captures. For example, to assess the security of encryption schemes, the adversary, given a ciphertext, has to identify the message whose encryption resulted in the ciphertext, with there being two options to choose from. If the adversary successfully computes an acceptable output, again depending on the security notion, the experiment's outcome is 1. If the adversary does not produce an acceptable output, the experiment's outcome is 0.

During the experiment, the adversary can typically interact with the experiment, or a challenger running the experiment, via its inputs and outputs, and via oracles. Oracles allow the adversary to see a cryptographic scheme in action. For example, an oracle may cause a party played by the experiment/challenger to encrypt a message of the adversary's choice. Thus, oracles enable the adversary to gather information that may help in computing whatever the adversary needs to compute in order to make the experiment's outcome 1.

Note that our experiments are parameterized with the security parameters that the experiment/challenger uses to instantiate cryptographic schemes. When evaluating the security of a cryptographic scheme $\mathsf{\Pi}$ relative to an experiment, we consider the probability of the outcome of experiment **Exp** to be 1 if run with adversary $\mathcal{A}$, denoted $\Pr[\mathbf{Exp}_{\mathcal{A},\mathsf{\Pi}}(\lambda) = 1]$. Specifically, we consider the advantage of the adversary. The advantage considers the probability of the experiment's outcome to be 1 by chance. For example, in the mentioned encryption example, the adversary has to decide what message was encrypted to a given ciphertext and the adversary knows the two options. The experiment/challenger chooses the message to encrypt uniformly at random from its options. So by simply guessing, an adversary can cause the experiment's result to be 1 50% of the time. Therefore, if **Exp** denotes the security experiment for encryption schemes and $\mathsf{\Pi}$ denotes an encryption scheme, the advantage of an adversary $\mathcal{A}$ is computed as $|\Pr[\mathbf{Exp}_{\mathcal{A},\mathsf{\Pi}}(\lambda) = 1] - 1/2|$, meaning how much better the adversary performs when compared to a random guess.

Qualifying security. The above approach allows us to quantify security of a cryptographic scheme as a probability or advantage for an adversary to perform well in an experiment. Yet, we still need to ask what types of adversaries we need to consider, how much computational power adversaries can wield, and what adversarial advantage we can allow for a scheme to be still considered secure. An adversary can be assumed to make full use of its powers, for example, by using the information gained from decrypting a ciphertext the adversary is not allowed to decrypt. Therefore, we typically restrict ourselves to adversaries that complete their computations within a manageable time frame. This is captured by the notion of polynomial time. Therefore, cryptography generally restricts adversaries to run in PPT, where schemes' security parameters are taken as reference, so adversaries asymptotically have the same resources as someone running the scheme.

We generally consider schemes secure with respect to some experiment if, in the experiment, no PPT adversary can perform significantly better than the random guess. What is considered significant, depends on the runtime of the adversary, and thus, the security parameter. Fixing a security parameter λ, we allow our adversaries runtimes polynomial in λ, denoted as $\mathsf{poly}(\lambda)$. We consider a PPT adversary's advantage against a scheme negligible in the security parameter λ, denoted $\mathsf{negl}(\lambda)$, if the adversary's advantage, as a function in λ, is asymptotically less than $1/\mathsf{poly}(\lambda)$. This means the advantage converges to 0 faster than every inverse polynomial.

Given this notation, our security definitions for cryptographic schemes take the following form. A cryptographic scheme Π, is called secure relative to experiment **Exp** if, for all PPT adversaries $\mathcal{A}$, $\Pr[\mathbf{Exp}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$ or $|\Pr[\mathbf{Exp}_{\mathcal{A},\Pi}(\lambda) = 1] - 1/2| \leq \mathsf{negl}(\lambda)$, depending on context, where the probabilities are taken over the random choices of the experiments and the adversaries.

Interaction with oracles. Before, we have mentioned oracles that an adversary can use to interact with a security experiment. Notation-wise, in descriptions of experiments, an adversary $\mathcal{A}$ on input 1^λ with access to oracle $\mathcal{O}$ is denoted as $\mathcal{A}^{\mathcal{O}(\cdot,\cdot)}(1^\lambda)$ or $\mathcal{A}^{\mathcal{O}(x,\cdot)}(1^\lambda)$. The first version means that the oracle takes two inputs, both of which the adversary can freely choose. In the second version, the oracle also takes two inputs, but the first input is fixed to x, so the adversary can only choose the second input. Particularly, x remains hidden from the adversary if the adversary does not get access to x by other means. For example, if x is a secret key that the adversary should not learn, we use oracles so the adversary can evaluate a function of the secret key without learning the key. If the adversary was allowed to learn x, we could give x to the adversary directly, so the adversary could compute on x itself, rather than relying on the oracle.

3 Building Blocks for Cryptographic Schemes

This chapter presents the cryptographic primitives that we use in various constructions throughout this thesis. Our discussion of primitives spans fundamental functions, such as hash functions, and other basic building blocks, such as encryption and signature schemes. We address each of these primitives, as well as some relations between them, in the upcoming sections.

3.1 The Fundamental Functions

Hash functions, pseudorandom functions, and trapdoor permutations are some of the most fundamental cryptographic primitives. They can be used to realize many other cryptographic concepts, including some of the other primitives presented in this chapter.

3.1.1 Hash Functions

A *hash function* can be used to efficiently map objects, for example, bit strings, into a finite set. The result of applying a hash function to an object is called a *hash or digest.* A typical security requirement is for different objects to result in different hashes. Of course, given that we have no restriction on the sizes of objects, particularly the lengths of bit strings being hashed, there must be two objects that result in the same hash. Therefore, the above requirement is slightly relaxed to the notion of *collision resistance.* That is, it is hard for any reasonable attacker to come up with two objects that result in the same digest under the hash function. A pair of distinct objects that result in the same digest is called *collision.* The notion of hash functions can be formalized as follows.

Definition 1 (Def. 5.1 of [KL14])**.** *A* hash function *is a pair* $\Pi = (\mathsf{Gen}, \mathsf{Hash})$ *of PPT algorithms.*

Gen *takes a security parameter* 1^λ*, and outputs a key* s*.*

Hash *takes a key* s *and a bit string* $x \in \{0,1\}^*$*, and outputs a bit string* $y \in \{0,1\}^{\ell(\lambda)}$*, where* ℓ *is a polynomial and* λ *is implicit in* s*. The algorithm is deterministic.*

We note that the key of a hash function is public. Although our formal definition operates on bit strings, the notion of hash functions can be generalized. For that, we use the fact that objects can be represented as bit strings, thus allowing objects to be input to and output by hash functions. Such conversions from objects to bit strings and bit strings to objects remain implicit throughout this thesis.

```
$\mathbf{Exp}^{\text{coll-res}}_{\mathcal{A},\Pi}(\lambda)$
• $s \leftarrow \Pi.\mathsf{Gen}(1^\lambda)$
• $(x,x') \leftarrow \mathcal{A}(s)$
• the experiment outputs 1 if
    1. $x \neq x'$ and
    2. $\Pi.\mathsf{Hash}(s,x) = \Pi.\mathsf{Hash}(s,x')$
    otherwise the experiment outputs 0.
```

Figure 3.1: Collision resistance experiment for hash function $\Pi = (\mathsf{Gen}, \mathsf{Hash})$ and attacker $\mathcal{A}$

The security notion of collision resistance is defined relative to experiment $\mathbf{Exp}^{\text{coll-res}}$ as presented in Figure 3.1.

Definition 2 (Def. 5.2 of [KL14]). *A hash function $\Pi = (\mathsf{Gen}, \mathsf{Hash})$ is called* collision resistant *if, for all PPT adversaries $\mathcal{A}$,*

$$\Pr[\mathbf{Exp}^{\textit{coll-res}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random bits of the adversary and the experiment.

3.1.2 Pseudorandom Functions

A *pseudorandom function* (PRF) is an efficiently computable keyed function that maps bit strings from some range to bit strings from some domain. The idea is for the mapping to look unpredictable — or pseudorandom — even though it is deterministic. Therefore, the notion for PRFs considers a truly random function. That is, a function chosen uniformly at random from the set of functions that, for every security parameter λ, have the same domain and range as the PRF. No PPT attacker can be able to distinguish the PRF from a truly random function. The notion of PRFs can be formalized as follows.

Definition 3 (Def. 3.25 of [KL14]). *A* pseudorandom function *is a pair $\Pi = (\mathsf{Gen}, \mathsf{Eval})$ of PPT algorithms.*

Gen *takes a security parameter 1^λ, and outputs a key k.*

Eval *takes a key k and a bit string $x \in \{0,1\}^*$, and outputs a bit string $y \in \{0,1\}^*$. The algorithm is deterministic.*

We require, for every PPT distinguisher D,

$$|\Pr[D^{\mathsf{Eval}(k,\cdot)}(1^\lambda) = 1] - \Pr[D^{f(\cdot)}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda),$$

where $k \leftarrow \mathsf{Gen}(1^\lambda)$, f is chosen uniformly at random from the set of functions that, for security parameter λ, have the same domain and range as Eval*, the superscript denotes oracle access, and the probability is taken over the random choices of f, k, and the adversary.*

PRFs look very similar to hash functions, and indeed PRFs naturally satisfy the collision resistance definition. Nevertheless, PRFs and hash functions are not the same. Particularly, the key used in a PRF is private.

```
$\mathbf{Exp}^{\text{invert}}_{\mathcal{A},\Pi}(\lambda)$
• $(I,td) \leftarrow \mathsf{Gen}(1^\lambda)$
• $x \leftarrow \mathsf{Sample}(I)$
• $x' \leftarrow \mathcal{A}(I,x)$
• the experiment outputs 1 if $\mathsf{Eval}(x) = \mathsf{Eval}(x')$ ; otherwise the experiment outputs 0.
```

Figure 3.2: Inversion experiment for trapdoor permutation $\Pi = (\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Inv})$ and adversary $\mathcal{A}$

3.1.3 Trapdoor Permutations

A *trapdoor permutation* is a permutation that is easy to compute, but hard to invert, unless one knows a "trapdoor" that allows for efficient inversion. This can be formalized as follows.

Definition 4 (Def. 13.1 of [KL14])**.** *A* trapdoor permutation *is a tuple* $\Pi = (\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Inv})$ *of* PPT *algorithms.*

Gen *takes a security parameter* 1^λ *and outputs tuple* (I, td) *with* $|I| \geq \lambda$*. The pair* (I, td) *implicitly defines* $\mathbb{D}$*, the domain and range of the permutation.*

Sample *on input* I *outputs a uniformly chosen element from* $\mathbb{D}$*.*

Eval *on input* I *and* $x \in \mathbb{D}$ *outputs* $y \in \mathbb{D}$*.*

Inv *on input* td *and* $y \in \mathbb{D}$ *outputs* $x \in \mathbb{D}$*.*

We require, for all $(I, td) \leftarrow \mathsf{Gen}(1^\lambda)$*,*

1. Eval *is a bijection and,*

2. *for all* $x \in \mathbb{D}$*,* $\mathsf{Inv}(td, \mathsf{Eval}(I, x)) = x$*.*

For $\Pi = (\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Inv})$ *to be a trapdoor permutation, we also require, for all* PPT *adversaries* $\mathcal{A}$*,*

$$\Pr[\mathbf{Exp}^{invert}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where experiment $\mathbf{Exp}^{invert}$ *is as defined in Figure 3.2 and the probability is taken over the randomness of the experiment and the adversary.*

3.1.4 The Random Oracle Model

The *random oracle* methodology offers techniques to prove cryptographic schemes secure in an idealized setting [BR93]. The random oracle itself is an idealization of objects such as hash functions, PRFs and trapdoor permutations. Random oracles behave like randomly chosen functions that have no publicly known description. Without a public description, in order to evaluate the oracle at some value, the oracle has to be queried. To anyone querying a value at the oracle, the answers they eventually receive are unpredictable if they have not queried the oracle on the same value before.

The unpredictability of random oracles is useful in security proofs. For that, in a proof of security of some construction, a primitive, for example, a hash function, used within the construction is treated as a random oracle. When the construction is implemented, the random oracle is instead replaced by an actual function, for example, the AES scheme [DR98] if the random oracle represents a PRF.

In security proofs, random oracles come in two flavors: non-programmable and programmable. Non-programmable random oracles just produce unpredictable output as discussed above. Programmable random oracles, on the other hand, are under the proof's control, so the oracle can be made to produce outputs that are convenient for the proof. A particular use of programmability is to "patch" random oracles. That is the proof determines inputs (to an oracle) that result in a certain output that is unknown at the time the inputs are defined. Later on in the proof, the outputs become clear, and the random oracle is retroactively defined ("patched") to behave in that particular way. Occasionally, patching the oracle may fail, namely, if the inputs have already been defined to result in a different output.

It has been shown that some cryptographic schemes can be proven secure if a primitive is modelled as a random oracle, but no matter how the primitive is instantiated, the instantiation is insecure [CGH04]; it must however be noted that such schemes are highly contrived. A scheme's proof of security in the random oracle model thus does not necessarily prove security of any instantiation of the scheme, but, as Katz and Lindell put it (p. 181 of [KL14]), "a proof of security for a scheme in the random oracle model indicates that the scheme's design is "sound," in the sense that the only possible attacks on a real-world instantiation of the scheme are those that arise due to a weakness in the hash function [or other primitive] used to instantiate the random oracle." So, we hope that instantiations of primitives are good enough at mimicking random oracles.

3.2 Encryption

Encryption aims at providing confidentiality of messages. That is, no unauthorized entity can learn any bit of the plaintext message from the encrypted message, also called ciphertext. While the concept of encryption is widely applicable, there are significant differences in usage scenarios that need to be accounted for in formal definitions of security. Differences may be due to the number of entities that may author a message, or the number of addressees of a message. We must also consider differences in an attacker's capabilities.

Concerning an attacker's capabilities, we mostly consider attackers that know many plaintext–ciphertext pairs for plaintexts of the attacker's choice. This is captured by the notion of security against chosen plaintext attacks (CPA). The notion is defined relative to a security experiment. As part of the CPA-security experiment, the attacker chooses two messages of its choice. One of these messages is encrypted, and the resulting ciphertext is given to the attacker. The attacker then has to decide which of the two messages was encrypted. This necessitates the two messages to be of equal length. Otherwise, it might be trivial to decide which message was encrypted, because short messages generally result in short ciphertexts, whereas long messages result in long ciphertexts. An encryption scheme is CPA-secure if the attacker is unable to decide which message was encrypted. This means the attacker essentially guesses what message was encrypted, despite the attacker knowing the two options, and knowing lots of ciphertexts for plaintexts of its choice, including plaintext–ciphertext pairs for the challenge messages. Achieving this security notion necessitates encryption to be probabilistic.

In addition to CPA-security, we also consider a stronger notion called security against chosen ciphertext attacks (CCA). In the CCA-security experiment, the attacker does not only know ciphertexts for plaintexts of its choice, but also plaintexts for ciphertexts of its choice (with the exception of the challenge ciphertext).

We also consider a notion of security called security against eavesdropping attacks. In this notion, in contrast to the CPA setting, the attacker has no knowledge of plaintext–ciphertext pairs. Depending on the number of potential authors and addressees in a scheme, eavesdropping-security is weaker than CPA-security or equivalent to CPA-security.

Concerning the numbers of potential authors and addressees of messages, we consider three cases.

1. The symmetric setting, in which the author is the addressee, or the author and the addressee share a secret. A prime example for the use of symmetric encryption is in order to prevent cloud service providers from learning the data that they store on behalf of their customers.

2. The public key setting, which features multiple authors, but a single addressee. Public key encryption can be used to encrypt communication directed at the addressee. Encryption of e-mails is a popular example.

3. The attribute-based setting, which features multiple authors and addressees that the author of a message may not identify by name, but by certain attributes. Attribute-based encryption is best used within (large) organizations. In such a setting, communication may be directed at all individuals that fill a certain role, like department heads, that is reflected in their attributes.

In this section, we now formally define the three cases, what security means in the respective context, and how the cases are often combined.

3.2.1 Symmetric Encryption

As stated, symmetric encryption targets the setting in which the author of a message and the addressee are the same or share a secret. In this setting, the shared secret is used during encryption and decryption. This can be formalized as follows.

Definition 5 (Def. 3.7 of [KL14]). *A* symmetric encryption scheme *is a triple* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *of PPT algorithms.*

KeyGen *takes a security parameter* 1^λ *and outputs key sk.*

Enc *takes key sk and a message msg, and outputs a ciphertext ct.*

Dec *takes key sk and ciphertext ct, and outputs a message msg.*

We require, for all $sk \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and all messages msg,* $\mathsf{Dec}(sk, \mathsf{Enc}(sk, msg)) = msg$.

For symmetric encryption schemes, CPA-security is defined relative to experiment $\mathbf{Exp}^{\text{SKE-cpa}}$, as presented in Figure 3.3. In this experiment, an attacker can request ciphertexts for arbitrary messages by asking the encryption oracle provided.

Definition 6 (Def. 3.22 of [KL14]). *A symmetric encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CPA-secure *if, for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$|\Pr[\mathbf{Exp}_{\mathcal{A},\Pi}^{\textit{SKE-cpa}}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

```
$\mathbf{Exp}^{\text{SKE-cpa}}_{\mathcal{A},\Pi}(\lambda)$
• $sk \leftarrow \mathsf{KeyGen}(1^\lambda)$
• $(st, msg_0, msg_1) \leftarrow \mathcal{A}_1^{\mathsf{Enc}(sk,\cdot)}(1^\lambda)$
• $b \leftarrow_\$ \{0,1\}$
• $ct^* \leftarrow \mathsf{Enc}(sk, msg_b)$
• $b' \leftarrow \mathcal{A}_2^{\mathsf{Enc}(sk,\cdot)}(st, ct^*)$
• the experiment outputs 1 if
  1. $|msg_0| = |msg_1|$ and
  2. $b = b'$;
  otherwise the experiment outputs 0.
```

Figure 3.3: CPA-security experiment for symmetric encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

```
$\mathbf{Exp}^{\text{SKE-cca}}_{\mathcal{A},\Pi}(\lambda)$
• $sk \leftarrow \mathsf{KeyGen}(1^\lambda)$
• $(st, msg_0, msg_1) \leftarrow \mathcal{A}_1^{\mathsf{Enc}(sk,\cdot),\mathsf{Dec}(sk,\cdot)}(1^\lambda)$
• $b \leftarrow_\$ \{0,1\}$
• $ct^* \leftarrow \mathsf{Enc}(sk, msg_b)$
• $b' \leftarrow \mathcal{A}_2^{\mathsf{Enc}(sk,\cdot),\mathsf{Dec}(sk,\cdot)}(st, ct^*)$
• the experiment outputs 1 if
  1. $|msg_0| = |msg_1|$,
  2. $b = b'$, and
  3. $\mathcal{A}_2$ has not queried $ct^*$ at the decryption oracle;
  otherwise the experiment outputs 0.
```

Figure 3.4: CCA-security experiment for symmetric encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

```
KeyGen(1^λ)
  • $sk \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
Enc(sk,msg)
  • $r \leftarrow_\$ \{0,1\}^\lambda$
  • $c \leftarrow \mathsf{PRF.Eval}(sk,r) \oplus msg$
  • output $(r,c)$
Dec(sk,ct)
  • output $\mathsf{PRF.Eval}(sk,r) \oplus c$
```

Figure 3.5: The CSSE symmetric encryption scheme built from a PRF; $\oplus$ denotes bit-wise xor; *msg* is assumed to be no longer than the output of the PRF

```
$\mathbf{Exp}^{\text{SKE-eav}}_{\mathcal{A},\Pi}(\lambda)$
  • $sk \leftarrow \mathsf{KeyGen}(1^\lambda)$
  • $(st, msg_0, msg_1) \leftarrow \mathcal{A}_1(1^\lambda)$
  • $b \leftarrow_\$ \{0,1\}$
  • $ct^* \leftarrow \mathsf{Enc}(sk, msg_b)$
  • $b' \leftarrow \mathcal{A}_2(st, ct^*)$
  • the experiment outputs 1 if
      1. $|msg_0| = |msg_1|$ and
      2. $b = b'$;
    otherwise the experiment outputs 0.
```

Figure 3.6: Eavesdropping-security experiment for symmetric encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

Security of symmetric encryption schemes against chosen-ciphertext attacks is defined relative to experiment $\mathbf{Exp}^{\text{SKE-cca}}$ form Figure 3.4. The experiment is similar to the CPA-security experiment, but the adversary can query a decryption oracle in addition to the encryption oracle. However, the adversary is prohibited from querying the challenge chiphertext at the decryption oracle, as to not allow the adversary to trivially win the experiment.

Definition 7 (Def. 3.33 of [KL14]). *A symmetric encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CCA-secure *if, for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$|\Pr[\mathbf{Exp}^{\textit{SKE-cca}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

Symmetric encryption schemes are tightly linked to pseudorandom functions (PRFs). An example is shown in Figure 3.5, which presents the CSSE (CPA-secure symmetric encryption) scheme.

Relaxation and use of symmetric encryption. In our use of symmetric encryption, CPA-security is occasionally stronger than what we need. Schemes achieving CPA-security

```
KeyGen($1^\lambda$)
• $k \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $r \leftarrow_\$ \{0,1\}^\lambda$
• $sk \leftarrow (k,r)$
• output $sk$

Enc($sk$,$msg$)
• output $\mathsf{PRF.Eval}(k,r) \oplus msg$

Dec($sk$,$ct$)
• output $\mathsf{PRF.Eval}(k,r) \oplus ct$
```

Figure 3.7: The ESSE symmetric encryption scheme built from a PRF; $\oplus$ denotes bit-wise xor; *msg* is assumed to be no longer than the output of the PRF

are impractical for some of our uses due to the inherent randomization in such schemes, see, for example, the use of random r in the above CSSE scheme. We therefore settle for the weaker model of security against eavesdropping attacks. Eavesdropping-security is defined relative to experiment $\mathbf{Exp}^{\text{SKE-eav}}$ as shown in Figure 3.6. This experiment is similar to experiment $\mathbf{Exp}^{\text{SKE-cpa}}$, but does not grant the adversary access to an encryption oracle, so the adversary only observes a single ciphertext encrypted under the secret key.

Definition 8 (Def. 3.8 of [KL14]). *A symmetric encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* eavesdropping-secure *if, for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$|\Pr[\mathbf{Exp}^{\textit{SKE-eav}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

As a consequence of the experiments, every CPA-secure symmetric encryption scheme is also eavesdropping-secure. The converse is not true, as we see in the following.

Consider the ESSE (eavesdropping-secure symmetric encryption) scheme shown in Figure 3.7. Note the similarities between the CSSE and ESSE schemes. Both schemes use the exact same mechanisms. The schemes only differ in the use of the random string r and when it is chosen. In the CSSE scheme, r is chosen during encryption and only used in the generation of a single ciphertext, whereas in the ESSE scheme, r is part of the secret key and used for all ciphertexts.

This subtle difference results in the ESSE scheme only being eavesdropping-secure, but not CPA-secure: assume two ESSE ciphertexts ct and ct' computed from messages msg and msg', respectively; then $ct \oplus ct' = msg \oplus msg'$, so the key cancels out. In experiment $\mathbf{Exp}^{\text{SKE-cpa}}$, the adversary can easily exploit this property to distinguish ciphertexts for either of the two challenge messages. In contrast, in experiment $\mathbf{Exp}^{\text{SKE-eav}}$, the adversary only learns one ciphertext, so the property cannot be exploited.

3.2.2 Public Key Encryption

Public key encryption schemes target a setting with multiple potential authors of messages. It is somewhat impractical for authors to establish shared secrets with every potential addressee and vice versa. Therefore, public key encryption uses a key pair, consisting of a public and a secret key. The key pair is generated by an addressee who keeps the secret

$\mathbf{Exp}^{\text{PKE-cpa}}_{\mathcal{A},\Pi}(\lambda)$

- $(pk,sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $(st,msg_0,msg_1) \leftarrow \mathcal{A}_1(pk)$
- $b \leftarrow_\$ \{0,1\}$
- $ct^* \leftarrow \mathsf{Enc}(pk,msg_b)$
- $b' \leftarrow \mathcal{A}_2(st,ct^*)$
- the experiment outputs 1 if
  1. $|msg_0| = |msg_1|$ and
  2. $b = b'$;

  otherwise the experiment outputs 0.

Figure 3.8: CPA-security experiment for public key encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

key to herself. In contrast, the public key, as the name implies, is published, and every author who wishes to communicate a message to the addressee uses the addressee's public key during encryption. The addressee users her secret key during decryption. All in all, public key encryption can be formalized as follows

Definition 9 (Def. 11.1 of [KL14])**.** *A* public key encryption scheme *is a triple* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *of PPT algorithms.*

KeyGen *takes a security parameter* 1^λ *and outputs public–private key pair* (pk, sk)*.*

Enc *takes public key pk and a message msg, and outputs a ciphertext ct.*

Dec *takes secret key sk and a ciphertext ct, and outputs a message msg.*

We require, for all $(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and all messages msg.*

$$\Pr[\mathsf{Dec}(sk, \mathsf{Enc}(pk, msg)) = msg] \geq 1 - \mathsf{negl}(\lambda).$$

For public key encryption schemes, CPA-security is defined relative to experiment $\mathbf{Exp}^{\text{PKE-cpa}}$, as presented in Figure 3.8. In contrast to the CPA-security experiment for symmetric encryption schemes, the attacker is not given explicit access to an encryption oracle, because the attacker gets a public key. Therefore, the attacker can encrypt messages on its own.

Definition 10 (Prop. 11.3 of [KL14])**.** *A public key encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CPA-secure *if, for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*,*

$$|\Pr[\mathbf{Exp}^{\textit{PKE-cpa}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

Concerning the notion of eavesdropping-security in the context of public key encryption, eavesdropping-security is equivalent to the notion of CPA-security. This is due to attackers having access to the public key, which turns the experiment for eavesdropping-security into a CPA-security experiment.

The security notion of CCA-security can also be defined for public key encryption schemes. Figure 3.9 shows the relevant security experiment. As in the symmetric case, the adversary gets access to a decryption oracle, but must not query the challenge ciphertext at the oracle.

```
$\mathbf{Exp}^{\text{PKE-cpa}}_{\mathcal{A},\Pi}(\lambda)$
• $(pk,sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
• $(st,msg_0,msg_1) \leftarrow \mathcal{A}_1(pk)$
• $b \leftarrow_\$ \{0,1\}$
• $ct^* \leftarrow \mathsf{Enc}(pk,msg_b)$
• $b' \leftarrow \mathcal{A}_2(st,ct^*)$
• the experiment outputs 1 if
    1. $|msg_0|=|msg_1|$ and
    2. $b=b'$;
  otherwise the experiment outputs 0.
```

Figure 3.9: CCA-security experiment for public key encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

Definition 11 (Prop. 11.13 of [KL14])**.** *A public key encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CCA-secure *if, for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*,*

$$|\Pr[\mathbf{Exp}^{\text{PKE-cca}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

3.2.3 Attribute-Based Encryption

Attribute-based encryption (ABE) schemes are used in settings in which authors address multiple addressees, potentially without knowing them directly. Particularly, addressees are known and addressed by their properties or roles, called attributes, rather than their names. ABE can be seen as an extension of public key encryption in which there are multiple secret keys — one per attribute — for the same public key, and multiple secret keys have to be used simultaneously during decryption. Whether or not a given combination of secret keys can be used to decrypt a ciphertext depends on a policy that served as input to the encryption operation that resulted in the ciphertext. This is reflected by our formalization of ABE.

Definition 12 ([SW05])**.** *An* ABE scheme *is a tuple* $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *of four PPT algorithms.*

Setup *takes a security parameter* 1^λ *and outputs a public parameters–master secret pair* (pp, msk)*.*

KeyGen *takes master secret msk and attribute set* U*, and outputs a user secret* usk_U*.*

Enc *takes public parameters pp, an access policy* $\mathbb{A}$*, and a message msg, and outputs a ciphertext ct.*

Dec *takes parameters pp, user secret usk and ciphertext ct, and outputs a message msg.*

We require, for all $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$*, all messages msg, and all access policy–attribute set pairs* $(\mathbb{A}, U)$*, if* U *satisfies* $\mathbb{A}$*, then, for* $usk_U \leftarrow \mathsf{KeyGen}(msk, U)$*, we have*

$$\Pr[\mathsf{Dec}(pp, usk_U, \mathsf{Enc}(pp, \mathbb{A}, msg)) = msg] \geq 1 - \mathsf{negl}(\lambda).$$

```
$\mathbf{Exp}^{\text{ABE-cpa}}_{\mathcal{A},\Pi}(\lambda)$
• $(pp,msk) \leftarrow \mathsf{Setup}(1^\lambda)$
• $(st,\mathbb{A}^*,msg_0,msg_1) \leftarrow \mathcal{A}_1^{\mathsf{KeyGen}(msk,\cdot)}(pp)$
• $b \leftarrow_\$ \{0,1\}$
• $ct^* \leftarrow \mathsf{Enc}(pp,\mathbb{A}^*,msg_b)$
• $b' \leftarrow \mathcal{A}_2^{\mathsf{KeyGen}(msk,\cdot)}(st,ct^*)$
• the experiment outputs 1 if
  1. $|msg_0| = |msg_1|$,
  2. $b = b'$, and
  3. for no query to oracle $\mathsf{KeyGen}(msk,U)$, $U$ satisfies $\mathbb{A}^*$;
  otherwise the experiment outputs 0.
```

Figure 3.10: CPA-security experiment for ABE scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

For ABE schemes, CPA-security is defined relative to experiment $\mathbf{Exp}^{\text{ABE-cpa}}$, as presented in Figure 3.10. Since ABE requires a policy during encryption, in the experiments, the adversary not only outputs two messages to be challenged upon, but also a policy $\mathbb{A}^*$ used to create the challenge ciphertext ct^*. As in the case of public key encryption, the attacker can encrypt arbitrary messages under arbitrary policies. In contrast to the case of public key encryption, the attacker gets access to a $\mathsf{KeyGen}(msk, \cdot)$ oracle, so it can gain decryption keys for arbitrary attribute sets. This means the adversary controls some corrupted users. However, this necessitates that the attacker must not ask for keys for attribute sets that satisfy the challenge policy $\mathbb{A}^*$.

Definition 13 ([SW05])**.** *An* ABE *scheme* $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CPA-secure *if, for all* PPT *adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*,*

$$|\Pr[\mathbf{Exp}^{\mathit{ABE\text{-}cpa}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random choices of the adversary and the experiment.

In the CPA case, the KeyGen oracle directly outputs the generated key. In contrast, in the CCA case, the KeyGen oracle does not output the key, but rather a handle of the key. The attacker can obtain the key by calling the Reveal oracle on the handle. The handle is also used to identify the key to use for queries to oracle Dec, which attempts to decrypt a given ciphertext by using the key identified by the handle.

Using the handle enables the adversary and the experiment to re-use the same key for multiple queries. As an alternative, some authors have suggested to generate a new user secret for each key generation or decryption query, and to use that secret only to answer that single query, see [KV08] for an example. It can be argued that both notions capture the intuitive description of CCA-security. However, it has been shown that the model we chose is strictly stronger than the alternative model [BL17]. Their CCA-security experiment $\mathbf{Exp}^{\text{ABE-cca}}$ for ABE schemes is presented in Figure 3.11, and CCA-security is defined relative to that experiment.

Definition 14 ([BL17])**.** *An* ABE *scheme* $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* CCA-secure *if, for all* PPT *adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*,*

$$|\Pr[\mathbf{Exp}^{\mathit{ABE\text{-}cca}}_{\mathcal{A},\Pi}(\lambda)] - 1/2| \leq \mathsf{negl}(\lambda),$$

$\underline{\mathbf{Exp}^{\text{ABE-cca}}_{\mathcal{A},\Pi}(\lambda)}$
- $(pp,msk) \leftarrow \mathsf{Setup}(1^\lambda)$
- $(st,\mathbb{A}^*,msg_0,msg_1) \leftarrow \mathcal{A}_1^{\mathsf{KeyGen}(msk,\cdot),\mathsf{Reveal}(\cdot),\mathsf{Dec}(\cdot,\cdot)}(pp)$
- $b \leftarrow_\$ \{0,1\}$
- $ct^* \leftarrow \mathsf{Enc}(pp,\mathbb{A}^*,msg_b)$
- $b' \leftarrow \mathcal{A}_2^{\mathsf{KeyGen}(msk,\cdot),\mathsf{Reveal}(\cdot),\mathsf{Dec}(\cdot,\cdot)}(st,ct^*)$
- the experiment outputs 1 if
  1. $|msg_0| = |msg_1|$,
  2. $b = b'$, and,
  3. for every query to oracle $\mathsf{KeyGen}(msk,U)$ that resulted in handle h, if U satisfies $\mathbb{A}^*$, then there is no query $\mathsf{Reveal}(h)$, and there is no query $\mathsf{Dec}(h,ct^*)$;

  otherwise the experiment outputs 0.

Figure 3.11: CCA-security experiment for ABE scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

where the probability is taken over the random choices of the adversary and the experiment.

A security property implicit in the definitions of CPA and CCA-security for ABE is *collusion resistance.* That is, users cannot combine their secret keys in order to decrypt ciphertexts that neither user can decrypt on her own. Given the collusion resistance property, ABE can be used to cryptographically enforce access control, and we use ABE in this capacity.

We note that there are multiple flavors of ABE, most notably *ciphertext-policy* [BSW07] and *key-policy* [Goy+06] ABE. Our presentation of ABE presents the ciphertext-policy case. In contrast, in key-policy ABE, the uses of attributes and policies are switched, so policies are used in key generation, and attributes are used in encryption.

The literature on ABE distinguishes schemes with different sizes of attribute universes, namely, *small-universe* ABE and *large-universe* ABE. In small-universe ABE, the total number of attributes supported by an instantiation of the scheme is polynomially bounded by the security parameter. In contrast, large-universe ABE supports attribute universes of super-polynomial or even infinite size.

Supporting key delegation. For our use of ABE, it is important that schemes feature *key delegation* [BSW07]. That is, given public parameters pp and a user secret for attribute set U, it is possible to efficiently derive a user secret for every set $U' \subset U$, and the derived key is functionally equivalent to a user key for attribute set U' computed via KeyGen. This is formalized as follows.

Definition 15. *An ABE scheme* $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *supports* key delegation *if there is a PPT algorithm* $\mathsf{Delegate}$ *such that, for all sets* $U \neq \emptyset \neq U'$ *of attributes from* Π*'s attribute universe with* $U' \subset U$*, the probability distributions*

$$\left\{ \begin{array}{l} (pp, msk, usk_U, usk_{U'}) : (pp, msk) \leftarrow \mathsf{Setup}(1^\lambda), usk_U \leftarrow \mathsf{KeyGen}(msk, U), \\ \qquad\qquad\qquad\qquad\qquad\quad usk_{U'} \leftarrow \mathsf{KeyGen}(msk, U') \end{array} \right\}$$

and

$$\left\{ \begin{array}{l} (pp, msk, usk_U, usk_{U'}) : (pp, msk) \leftarrow \mathsf{Setup}(1^\lambda), usk_U \leftarrow \mathsf{KeyGen}(msk, U), \\ \qquad\qquad\qquad\qquad\qquad\quad usk_{U'} \leftarrow \mathsf{Delegate}(pp, usk_U, U') \end{array} \right\}$$

are computationally indistinguishable.

Key delegation is supported by various ABE schemes, among others [BSW07; Wat11; RW13]. Due to indistinguishability of delegated user keys from keys computed via KeyGen, CCA-security, CPA-security and collusion resistance hold in the presence of delegated keys. Particularly, it is not possible to (re-)combine user secrets derived from the same original user secret.

3.2.4 Hybrid Encryption

In practice, asymmetric encryption, for example, in the form of public key and attribute-based encryption, suffers from low throughput during encryption and decryption. This is due to computationally expensive primitives used to implement asymmetric encryption. Symmetric encryption does not have this drawback.

Therefore, a common approach when dealing with large volumes of data is to use *hybrid encryption,* c.f. Section 11.3 of [KL14]. In hybrid encryption, in order to encrypt the data, one first samples a key for symmetric encryption, and then symmetrically encrypts the data and asymmetrically encrypts the symmetric key. Under hybrid encryption, a ciphertext thus consist of a symmetric and an asymmetric ciphertext. For decryption, first the symmetric key is restored from the asymmetric ciphertext, and then the data is recovered from the symmetric ciphertext. We note that a hybrid encryption scheme is CPA-secure if the asymmetric component is CPA-secure, and the symmetric component is eavesdropping-secure.

As a convention for this thesis, *asymmetric encryption is always used as part of hybrid encryption*, even if we do not explicitly say so.

3.2.5 Non-committing Encryption

For some of our security proofs, we need to create dummy ciphertexts. At the time these ciphertexts are created, it is unknown, even to the ciphertext creator, what plaintexts correspond to any of the ciphertexts. That relation between ciphertexts and plaintexts only becomes known at a later point in time. This dilemma is targeted by non-committing encryption, introduced by Canetti et al. [Can+96], and formalized, for example, by Hemenway et al. [HOR15]. While Hemenway et al. define non-committing encryption in the public key setting, we adapt their definition to the symmetric key setting.

Definition 16. *An encryption scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is called* non-committing *if there exists a pair of PPT algorithms* $(\mathsf{SimKeyGen}, \mathsf{SimEnc})$ *such that, for every message msg, the probability distributions*

$$\{(msg, sk, r_1, r_2) : sk \leftarrow \mathsf{KeyGen}(1^\lambda; r_1), ct \leftarrow \mathsf{Enc}(sk, msg; r_2)\}$$

and

$$\{(msg, sk, r_1, r_2) : (ct, t) \leftarrow \mathsf{SimEnc}(1^\lambda), (sk, r_1, r_2) \leftarrow \mathsf{SimKeyGen}(m, t)\}$$

are computationally indistinguishable, where r_1 *and* r_2 *are the random bits used by the probabilistic algorithms.*

Note that the ciphertexts ct are not considered as part of the tuples. This is because the ciphertexts deterministically depend on tuple components msg, sk, and r_2 via the Enc algorithm. Including ct in the tuples would therefore be redundant.

Definition 16 requires the existence of two simulator algorithms $\mathsf{SimKeyGen}$ and SimEnc. Algorithm SimEnc produces a dummy ciphertext ct and a trapdoor t. The trapdoor is

```
KeyGen(1^λ)
• $k \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $r \leftarrow_\$ \{0,1\}^\lambda$
• $sk \leftarrow (k,r)$
• output $sk$

Enc(sk,msg)
• output $\mathsf{PRF.Eval}(k,r) \oplus msg$

Dec(sk,ct)
• output $\mathsf{PRF.Eval}(k,r) \oplus ct$

SimEnc(1^λ)
• $(k,r) = sk \leftarrow \mathsf{KeyGen}(1^\lambda)$
• let $r_1$ be the random bits used by KeyGen
• $ct \leftarrow_\$ \{0,1\}^\lambda$
• $t \leftarrow (ct,sk,r_1)$
• output $(ct,t)$

SimKeyGen(t,msg)
• $r_2 \leftarrow \perp$
• program PRF such that $\mathsf{PRF.Eval}(k,r) = ct \oplus msg$
• output $(sk,r_1,r_2)$
```

Figure 3.12: Non-committing variant of the ESSE symmetric encryption scheme

then used as input by algorithm SimKeyGen, which additionally takes a message *msg* as input. SimKeyGen computes a symmetric key *sk* and random bits r_1, r_2 such that algorithm KeyGen run with random bits r_1 produces the key *sk* and algorithm Enc run with random bits r_2 on inputs *sk* and *msg* outputs the ciphertext *ct*.

We now illustrate how to construct non-committing, but secure, symmetric and hybrid encryption from PRFs and asymmetric encryption in the random oracle model. For the symmetric case, we rely on the ESSE scheme from Figure 3.7. Modelling the PRF as a programmable random oracle, the ESSE scheme is a non-committing eavesdropping-secure encryption scheme. The simulator algorithms are as shown in Figure 3.12, which also shows the original algorithms of the ESSE scheme. For the sake of simplicity, here we fix the PRF's key space, range, and domain as $\{0,1\}^\lambda$, which restricts the length of messages the scheme can encrypt to λ. The SimEnc algorithm declares a randomly chosen bit string as dummy ciphertext. The algorithm also chooses a secret key *sk* and randomness r_1 that it outputs as the trapdoor t. Algorithm SimKeyGen later on programs the random oracle to map the chosen secret key to a bit string that is the bit-wise xor of the dummy ciphertext and the plaintext. Randomness r_2 is set to the error symbol, because in ESSE, the encryption algorithm is deterministic and does not use random bits. Note that we choose the key and randomness as part of algorithm SimEnc, although the formalization of non-committing encryption allows us to wait and choose the randomness and key as part of the SimKeyGen algorithm. This choice comes in handy when constructing non-committing hybrid encryption.

Given the above simulator algorithms SimEnc and SimKeyGen, the ESSE scheme is non-committing in the random oracle model: In the random oracle model, the ciphertext *ct*, randomness r_1, and randomness r_2 are independent from each other and from the message *msg*, while randomness r_1 fixes the secret key. Therefore, the probability distribution induced by KeyGen and Enc, for every message *msg*, has fixed components *msg* and r_2, uniformly distributed component r_1, and component *sk* is determined by r_1 via the KeyGen algorithm. The probability distribution induced by the simulator algorithms has the same properties, so both distributions are identical, and thus indistinguishable.

Going asymmetric. Building on this non-committing eavesdropping-secure symmetric encryption scheme, we can also achieve non-committing CPA-secure hybrid encryption in the random oracle model. For this, we produce a dummy ESSE ciphertext *ct* via SimEnc, and use CPA-secure asymmetric encryption to encrypt the symmetric key from the trapdoor t also output by SimEnc. Later on, when the hybrid ciphertext is to be decrypted, algorithm $\mathsf{SimKeyGen}$ is executed on the desired plaintext message.

One particular advantage of combining the ESSE scheme with an asymmetric scheme is that the SimEnc algorithms of the ESSE scheme outputs a symmetric key as part of the trapdoor. This allows us to encrypt these keys using asymmetric encryption right away, before any plaintext messages are revealed. As a consequence, we can produce many ciphertext that can be decrypted to arbitrary messages under the same public key or system parameters.

We use non-committing asymmetric encryption in multiple proofs, but do not make explicit use of the simulator algorithms. Instead, we refer to dummy ciphertexts to indicate that these ciphertexts can be decrypted to arbitrary plaintexts of adequate length.

3.3 Commitment Schemes

A *commitment* is the cryptographer's equivalent to a sealed envelop [KL14]. The envelop contains information that the seal protects from modification after the envelop has been sealed. At the same time, as long as the envelop remains sealed, its contents remain private, so no party except for the sender of the envelop knows the information contained in it.

Definition 17 (Adapted from p. 188 of [KL14])**.** *A* commitment scheme *is a tuple* $\mathsf{\Pi} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ *of three PPT algorithms.*

Setup *takes in a security parameter* 1^λ *and outputs public parameters pp.*

Commit *takes in parameters pp and message msg, and outputs a pair* (c, d) *consisting of a commitment c and a decommit value d.*

Open *takes in parameters pp, a commitment c and decommit value d, and outputs a bit.*

We require, for $pp \leftarrow \mathsf{Setup}(1^\lambda)$ *and all messages msg, if* $(c, d) \leftarrow \mathsf{Commit}(pp, msg)$*, then*

$$\Pr[\mathsf{Open}(pp, c, d) = 1] \geq 1 - \mathsf{negl}(\lambda).$$

The formal definition does not make the relation of commitment schemes and the sealed envelop metaphor explicit. The Commit operation creates the sealed envelop represented as commitment c, while the decommit value represents a tool needed to crack the seal. Typically, the message *msg* is considered to be part of the decommit value d. Submitting the envelop then means that the commitment is published or sent to some addressee. Only when the creator of a commitment is willing to crack the seal on some envelop, she sends or publishes the decommit value that corresponds to the commitment.

$\mathbf{Exp}^{\text{Comm-bind}}_{\mathcal{A},\Pi}(\lambda)$

- $pp \leftarrow \mathsf{Setup}(1^\lambda)$
- $(c,d_1,d_2) \leftarrow \mathcal{A}(1^\lambda)$
- the experiment outputs 1 if
  1. $msg_1 \neq msg_2$ (with messages msg_1,msg_2 contained in decommit values d_1,d_2) and
  2. $\mathsf{Open}(pp,c,d_1) = 1 = \mathsf{Open}(pp,c,d_2)$;

  otherwise the experiment outputs 0.

Figure 3.13: Binding experiment for commitment scheme $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ and attacker $\mathcal{A}$

$\mathbf{Exp}^{\text{Comm-hide}}_{\mathcal{A},\Pi}(\lambda)$

- $pp \leftarrow \mathsf{Setup}(1^\lambda)$
- $(st,msg_0,msg_1) \leftarrow \mathcal{A}_1(1^\lambda)$
- $b \leftarrow_\$ \{0,1\}$
- $c \leftarrow \mathsf{Commit}(pp,msg_b)$
- $b' \leftarrow \mathcal{A}_2(st,c)$
- the experiment outputs 1 if $b=b'$; otherwise the experiment outputs 0.

Figure 3.14: Hiding experiment for commitment scheme $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

For security, some commitment schemes require the Setup operation to be executed honestly or by a party not involved in the other operations, for example, the scheme proposed in [GS12] and schemes based on public key encryption. Security notions for commitment schemes must ensure that authors cannot change the message they committed to after submitting their commitment. The notions must also ensure that receivers of a commitment cannot learn the message from the commitment before obtaining the corresponding decommit value. The latter two properties are called binding and hiding, and are defined relative to experiments $\mathbf{Exp}^{\text{Comm-bind}}$ and $\mathbf{Exp}^{\text{Comm-hide}}$ as shown in Figures 3.13 and 3.14, respectively.

Definition 18 (Def. 5.13 of [KL14])**.** *A commitment scheme* $\Pi = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ *is called* binding *and* hiding *if, for all PPT adversaries,*

$$\Pr[\mathbf{Exp}^{\textit{Comm-bind}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda) \quad \textit{and} \quad |\Pr[\mathbf{Exp}^{\textit{Comm-hide}}_{\mathcal{A},\Pi}(\lambda) = 1] - 1/2| \leq \mathsf{negl}(\lambda),$$

respectively. A commitment scheme is called secure *if it is binding and hiding.*

Note that we restrict our security notions above to computational notions, that is, security is defined relative to PPT adversaries. Stronger security notions exist, for example, statistical and perfect variants of the binding and hiding properties, and these notions can easily be achieved. However, computational security is sufficient in our settings.

A simple commitment scheme can be constructed from CPA-secure public key encryption as follows. For Setup, run the encryption scheme's KeyGen algorithm and output the resulting public key as *pp*. For committing to a message *msg*, encrypt *msg* under *pp*; *c* is the resulting ciphertext, *d* consists of *msg* and the randomness used by the encryption

$\mathbf{Exp}^{\text{Sig-euf-cma}}_{\mathcal{A},\Pi}(\lambda)$

- $(sk,vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $(msg^*,\sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(vk)$
- the experiment outputs 1 if (1) $\mathsf{Verify}(vk,msg^*,\sigma^*)=1$ and (2) none of $\mathcal{A}$'s oracle queries $\mathsf{Sign}(sk,m)$ satisfies $m=msg^*$; otherwise the experiment outputs 0.

Figure 3.15: EUF-CMA-security experiment for signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and attacker $\mathcal{A}$

algorithm. For opening a commitment, recompute the commitment c from the message and randomness from the decommit value. If the recomputed value equals the original commitment, output 1; otherwise, output 0.

The described scheme is binding. If it was not, there were two messages that can be encrypted to the same ciphertext under the same public key. This, however, violates the reconstruction requirement of public key encryption schemes. On the other hand, the above commitment scheme is hiding, because otherwise, the encryption scheme was not CPA-secure.

3.4 Signature Schemes

Signatures are supposed to guarantee a signed message (document) to originate from the signer or being approved by the signer, while making sure that the message has not been altered. Digital/cryptographic signatures realize these properties with cryptographic guarantees. In contrast to handwritten signatures, digital signatures should always be verifiable, and not just by comparing it to another signature from the same origin. Therefore, the verification mechanism for digital signatures includes a verification key, and all signatures of the same origin can be verified relative to that key.

Definition 19 (Def. 12.1 of [KL14])**.** *A* signature scheme *is tuple* $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *of three PPT algorithms.*

KeyGen *takes a security parameter* 1^λ *and outputs signing–verification key pair* (sk, vk)*.*

Sign *takes signing key sk and message msg, and outputs a signature* σ*.*

Verify *takes verification key vk, message msg and signature* σ*, and outputs a bit.*

We require, for $(sk, vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and all messages msg,*

$$\mathsf{Verify}(vk, msg, \mathsf{Sign}(sk, msg)) = 1.$$

Similar to handwritten signatures, digital signatures must be hard to forge. The unforgeability property of signatures is defined relative to experiment $\mathbf{Exp}^{\text{Sig-euf-cma}}$, as presented in Figure 3.15.

Definition 20 (Def. 12.2 of [KL14])**.** *A signature scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *provides* existential unforgeability under adaptively chosen message attack*, or EUF-CMA-security, if, for all PPT adversaries* $\mathcal{A}$*,*

$$\Pr[\mathbf{Exp}^{\textit{Sig-euf-cma}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda).$$

$\mathbf{Exp}^{\text{Sig-suf-cma}}_{\mathcal{A},\Pi}(\lambda)$

- $(sk,vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $(msg^*,\sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(vk)$
- the experiment outputs 1 if (1) $\mathsf{Verify}(vk,msg^*,\sigma^*)=1$ and (2) none of $\mathcal{A}$'s oracle queries $\mathsf{Sign}(sk,m)$ satisfies $m=msg^*$ and resulted in σ^*; otherwise the experiment outputs 0.

Figure 3.16: SUF-CMA-security experiment for signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and attacker $\mathcal{A}$

The notion of EUF-CMA-security means that it is hard to forge signatures on messages not queried at the signing oracle. For some applications, EUF-CMA-security is insufficient. A stronger notion of security, denoted as SUF-CMA-security, means that it is even hard to forge *new* signatures on messages queried at the signing oracle. The strong unforgeability notion is defined relative to experiment $\mathbf{Exp}^{\text{Sig-suf-cma}}$, as presented in Figure 3.16.

Definition 21 (Def. 1 of [BSW06])**.** *A signature scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *provides* strong existential unforgeability under adaptively chosen message attack*, or SUF-CMA-security, if, for all PPT adversaries* $\mathcal{A}$*,*

$$\Pr[\mathbf{Exp}^{\textit{Sig-suf-cma}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda).$$

Obviously, an SUF-CMA-secure signature scheme is also EUF-CMA-secure, but the converse does not hold. There are re-randomizable signatures that allow the derivation of new signatures on a given message from other signatures on the message, for example, the re-randomizable sigantures from [PS16]. It is easy to see that EUF-CMA-secure re-randomizable signatures cannot be SUF-CMA-secure.

Signing long messages: hash–then–sign. Signature schemes in practice suffer from inefficiencies when being applied to long messages, similar to asymmetric encryption. In asymmetric encryption, we deal with such inefficiencies by combining asymmetric and symmetric encryption, yielding hybrid encryption. A similar approach is used for signature schemes: one uses a hash function to shorten long messages to a short hash, and then signs the hash. This hash–then–sign approach yields EUF-CMA-secure signatures if the underlying signature scheme is EUF-CMA-secure and the hash function is collision resistant, c.f. Theorem 12.4 of [KL14]. An analogous result holds for SUF-CMA-secure signatures.

Hiding the message. Some of our applications necessitate signed messages to remain hidden from the public for some time. At the same time, the signature should be public in order to communicate the existence of the signed message, and the origin of the signature should be verifiable. At some later point in time, the signer can release an open value to the public that allows the signature itself (not just its origin) to be verified.

These ideas are captured by the notion of signature schemes with delayed verifiability [BL19a]. Such signature schemes have two verification algorithms: one for verifying a signature's origin and one for verifying the signature relative to the signed message.

Definition 22 ([BL19a])**.** *A* signature scheme with delayed verifiability *is a tuple* $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ *of four PPT algorithms.*

KeyGen *takes a security parameter* 1^λ *and outputs a signing–verification key pair* (sk, vk)*.*

```
$\mathbf{Exp}^{\text{DSig-euf-cma}}_{\mathcal{A},\Pi}(\lambda)$
• $(sk,vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
• $(msg^*,\sigma^*,d^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(vk)$
• the experiment outputs 1 if
    1. $\mathsf{VSig}(vk,msg^*,\sigma^*,d^*)=1$ and
    2. none of $\mathcal{A}$'s oracle queries $\mathsf{Sign}(sk,m)$ satisfies $m=msg^*$;
    otherwise the experiment outputs 0.
```

Figure 3.17: Signature unforgeability experiment for signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ with delayed verifiability and attacker $\mathcal{A}$

```
$\mathbf{Exp}^{\text{DSig-orig}}_{\mathcal{A},\Pi}(\lambda)$
• $(sk,vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
• $(\sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(vk)$
• the experiment outputs 1 if
    1. $\mathsf{VOrig}(vk,\sigma^*)=1$ and
    2. $\sigma^*$ was not output by oracle $\mathsf{Sign}(sk,\cdot)$;
    otherwise the experiment outputs 0.
```

Figure 3.18: Origin unforgeability experiment for signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ with delayed verifiability and attacker $\mathcal{A}$

Sign *takes signing key sk and a message msg, and outputs a signature–open value pair* (σ, d).

VOrig *takes verification key vk and signature* σ*, and outputs a bit.*

VSig *takes verification key vk, message msg, signature* σ *and open value d, and outputs a bit.*

We require, for $(sk, vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and all messages msg, if* $(\sigma, d) \leftarrow \mathsf{Sign}(sk, msg)$*, then*

$$\mathsf{VOrig}(vk, \sigma) = 1 = \mathsf{VSig}(vk, msg, \sigma, d).$$

Signature schemes with delayed verifiability must satisfy unforgeability properties similar to signature schemes, and these properties must hold relative to both verification algorithms. Unforgeability properties are defined relative to experiments $\mathbf{Exp}^{\text{DSig-orig}}$ and $\mathbf{Exp}^{\text{DSig-euf-cma}}$, as show in Figures 3.17 and 3.18, respectively. The notion of keeping the signed message hidden until the corresponding open value is released is defined relative to experiment $\mathbf{Exp}^{\text{DSig-hide}}$, as show in Figure 3.19.

Definition 23 ([BL19a])**.** *A signature scheme* $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ *with delayed verifiability is called* secure *if, for all triples* $(\mathcal{A}, \mathcal{A}', \mathcal{A}^* = (\mathcal{A}_1, \mathcal{A}_2))$ *of PPT adversaries,*

$$\Pr[\mathbf{Exp}^{\textit{DSig-euf-cma}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

$$\Pr[\mathbf{Exp}^{\textit{DSig-orig}}_{\mathcal{A}',\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

$\mathbf{Exp}^{\text{DSig-hide}}_{\mathcal{A},\Pi}(\lambda)$

- $(sk,vk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $(st,msg_0,msg_1) \leftarrow \mathcal{A}_1^{\mathsf{Sign}(sk,\cdot)}(vk)$
- $b \leftarrow_\$ \{0,1\}$
- $(\sigma^*,d^*) \leftarrow \mathsf{Sign}(sk,msg_b)$
- $b' \leftarrow \mathcal{A}_2^{\mathsf{Sign}(sk,\cdot)}(st,\sigma^*)$
- the experiment outputs 1 if $b=b'$; otherwise the experiment outputs 0.

Figure 3.19: Message hiding experiment for signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ with delayed verifiability and attacker $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

and

$$|\Pr[\mathbf{Exp}^{\textit{DSig-hide}}_{\mathcal{A}^*,\Pi}(\lambda) = 1] - 1/2| \leq \mathsf{negl}(\lambda).$$

Secure signature schemes with delayed verifiability can be constructed from SUF-CMA-secure signature schemes and secure commitment schemes in a commit–then–sign fashion. A construction of this kind is shown in Figure 3.20. Origin unforgeability of our construction is implied by the SUF-CMA-security of the underlying signature scheme. Signature unforgeability of our construction can be proven in the same way hash–then–sign is proven secure by Katz and Lindell (Theorem 12.4, resp. Theorem 5.6 of [KL14]), exchanging the hash function's collision resistance for the commitment scheme's binding property. The hiding property of our construction follows from the hiding property of the commitment scheme.

Throughout this thesis, whenever we make use of a signature scheme with delayed verifiability, we assume the scheme to be constructed following the commit–then–sign approach, that is the construction follows the template laid out in Figure 3.20.

Note that commitment schemes may not support committing to messages of arbitrary length. This problem can be mitigated in the same way it is mitigated in signature schemes, namely, by employing a hash–then–commit strategy. If the hash function is collision resistant, the hash–then–commit scheme retains the security properties of the underlying commitment scheme, namely, the hiding and binding property.

Also note that the parameters of the commitment scheme can be computed during key generation, as is done in our above construction. Alternatively, the parameters can be supplied externally. If commitment scheme parameters are supplied externally, we denote this by passing the parameters to the KeyGen algorithm.

Both approaches have security implications in practice that stem from concrete constructions of commitment schemes, for example, trapdoors in commitment schemes. Particularly, if the commitment scheme needs its parameters to be chosen by a trusted party, it may be advantageous to supply commitment scheme parameters to our construction externally. This is especially true if we cannot guarantee that the KeyGen algorithm is executed honestly. Throughout this thesis, potentially non-honest execution of the KeyGen algorithm is the norm; therefore we supply commitment scheme parameters externally.

Non-committing signatures. In the random oracle model, we can make signatures with delayed verifiability non-committing, c.f. non-committing encryption in Section 3.2.5. Such signatures are not binding, so we can sign unknown messages, and the message remains undetermined until the signature's open value gets revealed. Meanwhile, the

```
$\underline{\mathsf{KeyGen}(1^\lambda)}$
• $(sk',vk') \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
• $(pp) \leftarrow \Gamma.\mathsf{Setup}(1^\lambda)$
• output $(sk,vk) = ((sk',pp),(vk',pp))$

$\underline{\mathsf{Sign}(sk,msg)}$
• $(c,d) \leftarrow \Gamma.\mathsf{Commit}(pp,msg)$
• $\sigma' \leftarrow \Sigma.\mathsf{Sign}(sk',c)$
• $\sigma \leftarrow (c,\sigma')$
• output $(\sigma,d)$

$\underline{\mathsf{VOrig}(vk,\sigma)}$
• output $\Sigma.\mathsf{Verify}(vk',c,\sigma')$

$\underline{\mathsf{VSig}(vk,msg,\sigma,d))}$
• if $\Sigma.\mathsf{Verify}(vk',c,\sigma') = 1$ and $\Gamma.\mathsf{Open}(pp,c,d) = 1$: output 1
• otherwise: output 0
```

Figure 3.20: Construction of a signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{VOrig}, \mathsf{VSig})$ with delayed verifiability from signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and commitment scheme $\Gamma = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$

signature on an unknown message cannot be distinguished from a signature on a known message, both before and after the open value that corresponds to the signature has been revealed. As soon as the open value is revealed, the signature becomes binding. As with non-committing encryption, non-committing signatures with delayed verifiability are a tool used in security proofs.

We construct non-committing signatures with delayed verifiability using the above idea of commit–then–sign, but model the commitment part by a random oracle as follows: Let H be the random oracle; for committing to a message msg, sample $r \leftarrow_\$ \{0,1\}^\lambda$, compute $c \leftarrow H(\langle r, msg\rangle)$, $d \leftarrow (r, msg)$, and output (c, d); for opening the commitment $(c, d = (r, msg))$, output 1 if $c = H(\langle r, msg\rangle)$ and 0 otherwise. If the random oracle cannot be patched, this commitment scheme is binding, but if the random oracle can be patched, randomness r allows the commitment to be opened to arbitrary messages, even to messages for which other commitments are already known. When the open value eventually gets published, patch the random oracle so it maps the open value to the commitment c. Patching the oracle only fails if the open value is already mapped to a different commitment. This however, is rarely the case due to the randomness r being chosen from an exponentially large set (in the security parameter λ), so oracle patching only fails with negligible probability. Incidentally, r being chosen from a set of exponential size is also what makes the above scheme hiding.

3.5 Dictionaries

A dictionary is a dynamic data structure that maps labels to (multi-)sets of values.[1] The data structure supports two main operations: Appending new labels and values to the

[1]The data structures literature uses the term key instead of label. We prefer the term label as to avoid confusion with cryptographic keys.

data structure, and looking up values for any given label. In this thesis, we make use of append-only authenticated dictionaries (AADs). These dictionaries provide cryptographic guarantees, particularly a guarantee that labels and values cannot be removed from the dictionary once they have been added, and a guarantee that we can be confident that results to lookup queries are correct, so results can be verified. The cryptographic guarantees necessitate that AADs feature several helper operation, for example, a setup operation for generating verification keys and operations for verifying results to lookup queries.

The setup operation is to be executed by a trusted party. A server is to store the dictionary, to answer lookup queries, and to provide proofs that make the server's operations verifiable by users. Proof verification is performed relative to dictionary digests that users store locally. Occasionally, every user will request an updated digest from the server. The server provides the most recent digest and proves that the new digest represents a dictionary that originates from a sequence of append operation that have been applied to the dictionary represented by the user's old digest. The proof can be verified relative to the user's old digest. It is expected that the user then replaces her old digest by the newly received one. Similarly, whenever a user causes the server to perform a lookup operation, the server responds by sending the result of the operation together with a proof that the result is correct and complete, and the user can verify the proof relative to her dictionary digest.

These ideas have been formalized by Tomescu et al. [Tom+19], who also provide the first implementation of AADs with succinct proofs. That is, proofs of size sub-linear in the number of append operations. In previous constructions, at least one of the two proof types was non-succinct.

Definition 24 ([Tom+19]). *An* append-only authenticated dictionary *is a tuple* $\Pi = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Lookup}, \mathsf{PAppend}, \mathsf{VResult}, \mathsf{VAppend}, \mathsf{Version})$ *of seven polynomial time algorithms.*

Setup *takes a security parameter λ and a capacity β, and outputs parameter–verification key pair (pp, vk).*

Append *takes parameters pp, dictionary–digest pair $(D, \mathrm{digest}(D))$ and a set of label–value pairs $\{(\ell_1, v_1), \dots, (\ell_k, v_k)\}$, and outputs an updated dictionary–digest pair $(D, \mathrm{digest}(D))$.*

Lookup *takes parameters pp, dictionary D and label ℓ, and outputs a set $\mathbf{v}$ together with a proof $\pi_{\ell,\mathbf{v}}$.*

PAppend *takes parameters pp and two dictionaries D, D', and outputs a proof $\pi_{D,D'}$.*

VResult *takes verification key vk, digest $\mathrm{digest}(D)$, label ℓ, set $\mathbf{v}$ and proof π, and outputs a bit.*

VAppend *takes verification key vk, two digests $\mathrm{digest}(D), \mathrm{digest}(D')$ and proof π, and outputs a bit.*

Version *takes a dictionary digest $\mathrm{digest}(D)$ and outputs an integer.*

Algorithm Setup *is probabilistic; the other algorithms are deterministic. We require AADs to be* membership correct *(mc),* append-only correct *(aoc), and* version correct *(vc).*

mc: *For $(pp, vk) \leftarrow \mathsf{Setup}(1^\lambda, \beta)$ and every non-empty sequence of* Append *operations that result in dictionary D with digest $\mathrm{digest}(D)$, we have that $(\mathbf{v}, \pi) \leftarrow \mathsf{Lookup}(pp, D, \ell)$ satisfies $\mathsf{VResult}(vk, \mathrm{digest}(D), \ell, \mathbf{v}, \pi) = 1$.*

$\mathbf{Exp}_{\mathcal{A},\Pi}^{\text{AAD-ms}}(\lambda,\beta)$
- $(pp,vk) \leftarrow \mathsf{Setup}(1^\lambda,\beta)$
- $(\text{digest}(D),\ell,\mathbf{v},\mathbf{v}',\pi,\pi') \leftarrow \mathcal{A}(pp)$
- the experiment outputs 1 if
  1. $\mathbf{v} \neq \mathbf{v}'$,
  2. $0 < \mathsf{Version}(\text{digest}(D)) \leq \beta$, and
  3. $\mathsf{VResult}(vk,\text{digest}(D),\ell,\mathbf{v},\pi) = 1 = \mathsf{VResult}(vk,\text{digest}(D),\ell,\mathbf{v}',\pi')$;

  otherwise the experiment outputs 0.

Figure 3.21: Membership security experiment for AAD $\Pi = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Lookup}, \mathsf{PAppend}, \mathsf{VResult}, \mathsf{VAppend}, \mathsf{Version})$ adversary $\mathcal{A}$

aoc: *For* $(pp, vk) \leftarrow \mathsf{Setup}(1^\lambda, \beta)$*, every non-empty sequence of* Append *operations that result in dictionary* D *with digest* digest(D)*, and every non-empty sequence of* Append *operations that, applied to* D*, results in dictionary* D' *with digest* digest(D')*, we have that* $\pi \leftarrow \mathsf{PAppend}(pp, D, D')$ *satisfies* $\mathsf{VAppend}(vk, \text{digest}(D), \text{digest}(D'), \pi) = 1$.

vc: *For* $(pp, vk) \leftarrow \mathsf{Setup}(1^\lambda, \beta)$*, every non-empty sequence of* Append *operations that result in dictionary* D *with digest* digest(D)*, and every* Append *operation that, applied to* D*, results in dictionary* D' *with digest* digest(D')*, we have that* $\mathsf{Version}(\text{digest}(D)) > 0$*,* $\mathsf{Version}(\text{digest}(D')) = \mathsf{Version}(\text{digest}(D)) + 1$*, and* $\mathsf{Version}(\text{digest}(D')) \leq \beta$*.*

Note that we have modified the definition of Tomescu et al. in order to make versions more explicit and extractable from digests. This change results in the notion of version correctness that is not present in the original definition.

Conceptually, the correctness notions from the above definition mean

- that answers to Lookup queries can be successfully verified via VResult,
- that proofs resulting from PAppend called on a pair dictionaries of which one is a successor of the other are accepted by VAppend, and
- that Append operations increment the version counter, as well as that the number of Append operation is limited by the capacity of the data structure.

In AADs, there is a correspondence between correctness and security notions. Membership correctness corresponds to membership security, append-only correctness corresponds to append-only security, and version correctness corresponds to fork consistency. Membership security means that an adversary cannot claim two different sets to be associated with the same label relative to the same digests. Append-only security means that an adversary cannot claim a value to be associated with some label at some time, but not at a later time. Fork consistency means that an adversary cannot claim a dictionary digest to be a successor digest of two different digests that have the same version number. In contrast to the append-only security and fork consistency definitions of Tomescu et al., we allow the successor property on dictionary digests to be established indirectly, through a sequence of append-only proofs. Tomescu et al. only allow the successor property to be established by a single append-only proof. The three security notions are defined relative to experiments shown in Figures 3.21, 3.22, and 3.23. They are formalized as follows.

```
$\mathbf{Exp}^{\text{AAD-aos}}_{\mathcal{A},\mathsf{\Pi}}(\lambda,\beta)$
• $(pp,vk) \leftarrow \mathsf{Setup}(1^\lambda,\beta)$
• $((\text{digest}(D_n),\pi_n)_n,\text{digest}(D'),\ell,\mathbf{v},\mathbf{v}',\pi',\pi'') \leftarrow \mathcal{A}(pp)$
• the experiment outputs 1 if
  1. $\mathbf{v} \not\subseteq \mathbf{v}'$,
  2. $0 < \mathsf{Version}(\text{digest}(D_1))$,
  3. for $1 \leq i < n$:
      – $\mathsf{VAppend}(vk,\text{digest}(D_i),\text{digest}(D_{i+1}),\pi_i) = 1$,
      – $\mathsf{Version}(\text{digest}(D_i)) < \mathsf{Version}(\text{digest}(D_{i+1}))$,
  4. $\mathsf{VAppend}(vk,\text{digest}(D_n),\text{digest}(D'),\pi_n) = 1$,
  5. $\mathsf{Version}(\text{digest}(D_n)) < \mathsf{Version}(\text{digest}(D')) \leq \beta$, and
  6. $\mathsf{VResult}(vk,\text{digest}(D_1),\ell,\mathbf{v},\pi') = 1 = \mathsf{VResult}(vk,\text{digest}(D'),\ell,\mathbf{v}',\pi'')$;
  otherwise the experiment outputs 0.
```

Figure 3.22: Append-only security experiment for AAD $\mathsf{\Pi}$ = (Setup, Append, Lookup, PAppend, VResult, VAppend, Version) adversary $\mathcal{A}$

```
$\mathbf{Exp}^{\text{AAD-fc}}_{\mathcal{A},\mathsf{\Pi}}(\lambda,\beta)$
• $(pp,vk) \leftarrow \mathsf{Setup}(1^\lambda,\beta)$
• $((\text{digest}(D_n),\pi_n)_n,(\text{digest}(D'_m),\pi'_m)_m,\text{digest}(D^*)) \leftarrow \mathcal{A}(pp)$
• the experiment outputs 1 if
  1. $\text{digest}(D_1) \neq \text{digest}(D'_1)$,
  2. $0 < \mathsf{Version}(\text{digest}(D_1)) = \mathsf{Version}(\text{digest}(D'_1))$,
  3. for $1 \leq i < n$ and $1 \leq j < m$:
      – $\mathsf{VAppend}(vk,\text{digest}(D_i),\text{digest}(D_{i+1}),\pi_i) = 1$,
      – $\mathsf{Version}(\text{digest}(D_i)) < \mathsf{Version}(\text{digest}(D_{i+1}))$,
      – $\mathsf{VAppend}(vk,\text{digest}(D'_j),\text{digest}(D'_{j+1}),\pi'_j) = 1$,
      – $\mathsf{Version}(\text{digest}(D'_j)) < \mathsf{Version}(\text{digest}(D'_{j+1}))$,
  4. $\mathsf{VAppend}(vk,\text{digest}(D_n),\text{digest}(D^*),\pi_n) = 1$,
  5. $\mathsf{VAppend}(vk,\text{digest}(D'_m),\text{digest}(D^*),\pi'_m) = 1$,
  6. $\mathsf{Version}(\text{digest}(D_n)) < \mathsf{Version}(\text{digest}(D^*)) \leq \beta$, and
  7. $\mathsf{Version}(\text{digest}(D'_m)) < \mathsf{Version}(\text{digest}(D^*)) \leq \beta$;
  otherwise the experiment outputs 0.
```

Figure 3.23: Fork consistency experiment for AAD $\mathsf{\Pi}$ = (Setup, Append, Lookup, PAppend, VResult, VAppend, Version) adversary $\mathcal{A}$

Definition 25 ([Tom+19]). *An append-only authenticated dictionary* $\Pi = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Lookup}, \mathsf{PAppend}, \mathsf{VResult}, \mathsf{VAppend}, \mathsf{Version})$ *is called* membership secure *(ms), append-*only secure *(aos) and* fork consistent *(fc) if, for all* PPT *adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ *and all* $\beta \in O(\mathsf{poly}(\lambda))$*,*

$$\Pr[\mathbf{Exp}^{AAD\text{-}ms}_{\mathcal{A}_1,\Pi}(\lambda, \beta) = 1] \leq \mathsf{negl}(\lambda),$$

$$\Pr[\mathbf{Exp}^{AAD\text{-}aos}_{\mathcal{A}_2,\Pi}(\lambda, \beta) = 1] \leq \mathsf{negl}(\lambda), \text{ and}$$

$$\Pr[\mathbf{Exp}^{AAD\text{-}fc}_{\mathcal{A}_3,\Pi}(\lambda, \beta) = 1] \leq \mathsf{negl}(\lambda),$$

respectively. An AAD *is called* secure*, if it is membership secure, append-only secure, and fork consistent.*

Properties of secure append-only authenticated dictionaries. As a consequence of membership security, we have that *dictionary digests are unlikely to coincide* for the same version number. That means PPT adversaries, given pp and any dictionary D with digest digest(D), only succeed with negligible probability in computing two distinct sets LV, LV' of label–value pairs, such that $(D', \text{digest}(D')) \leftarrow \mathsf{Append}(pp, D, LV)$, $(D^*, \text{digest}(D^*)) \leftarrow \mathsf{Append}(pp, D, LV')$, $1 \leq \mathsf{Version}(\text{digest}(D')) \leq \beta$, and $\text{digest}(D') = \text{digest}(D^*)$.

As a consequence of AAD algorithms being deterministic, a dictionary digest cannot be associated with multiple version numbers, so *dictionary digests cannot coincide for different versions.*

As a consequence of fork consistency, we have that *dictionary digests depend on the order of* Append *operations.* We see this by considering $(pp, vk) \leftarrow \mathsf{Setup}(1^\lambda, \beta)$, dictionary D with digest digest(D), two distinct sets LV, LV' of label–value pairs, and operations $(D_1, \text{digest}(D_1)) \leftarrow \mathsf{Append}(pp, D, LV)$, $(D_2, \text{digest}(D_2)) \leftarrow \mathsf{Append}(pp, D_1, LV')$, $(D_1', \text{digest}(D_1')) \leftarrow \mathsf{Append}(pp, D, LV')$, and $(D_2' \, \text{digest}(D_2')) \leftarrow \mathsf{Append}(pp, D_1', LV)$. Assuming the server is honest, we have

$$\mathsf{Version}(\text{digest}(D_1)) = \mathsf{Version}(\text{digest}(D_1')) < \mathsf{Version}(\text{digest}(D_2')) = \mathsf{Version}(\text{digest}(D_2)),$$

and the server can obviously produce append-only proofs π, π' such that

$$\mathsf{VAppend}(vk, \text{digest}(D_1), \text{digest}(D_2), \pi) = 1 = \mathsf{VAppend}(vk, \text{digest}(D_1'), \text{digest}(D_2'), \pi').$$

In this setting, $\text{digest}(D_2) = \text{digest}(D_2')$ poses a breach of fork consistency. Therefore, because our AAD is secure, we have $\text{digest}(D_2) \neq \text{digest}(D_2')$.

As a combined consequence of membership security and fork consistency, two distinct sequences of Append operations of equal length are unlikely to result in the same dictionary digest. Consider the earliest set of label–value pairs that the sequences of Append operations differ in. Due to membership security, the respective Append operations, except with negligible probability, result in distinct dictionary digests. Due to fork consistency, except with negligible probability, the remaining Append operations result in distinct final digests.

Part II

Searchable Encryption

4 Introduction to Searchable Encryption

Searchable encryption is a technology that allows for secure — encrypted — storage of data in an unsecure location, without sacrificing the ability to search the data. Typically, as a consequence of the security properties of encryption, searching encrypted documents should be infeasible without decrypting the document ciphertexts first. However, searchable encryption uses dedicated index structures to enable search, and the search process can even be performed remotely by the server that also stores the encrypted data without that server learning the searched keywords or the contents of documents.

In this chapter, we discuss some of the fundamental properties and security guarantees of searchable encryption on an intuitive level, illustrating what we aim for in our constructions of searchable encryption, particularly combining searchable encryption and access control in a multi-user environment.

Goals of searchable encryption. As stated, the goal in searchable encryption is to store a document collection remotely and in a secure fashion, while maintaining the ability to search the document collection efficiently. As a consequence of the notions of security and efficiency, the search procedure has to be performed remotely, and the server that stores the document collection and performs search cannot learn the contents of (parts of) the document collection, neither before, nor during, nor after search. The requirement of the server not learning the contents of the document collection point is due to, typically, the remote server being operated by entities other than the owner of the searchable document collection. Such entities generally should not be trusted with data, for example, in the setting of cloud computing.

For searchable encryption to function as described above, it must feature two main operations. The first operation captures the search process. The second operation captures the creation of and updates to the remote document collection. Depending on the setting of searchable encryption, the search and update operations can be modeled as interactive protocols, or as collections of probabilistic algorithms (which then imply a, typically 2-round, protocol). If a process is modelled as a collection of algorithms, it typically consist of a query formulation algorithm for the user, and a query processing algorithm for the server.

Whenever a user searches the remote document collection for a keyword, the server is to respond by sending all documents ever added to the collection that contain the keyword and are not excluded due to restrictions arising from the applicable usage scenario. For example, in a multi-user scenario, documents may be excluded from the reply due to access restrictions.

Document collections. We illustrate this point by giving an example. However, in order to do so, we first we need to discuss the structure of documents and document collections.

A document collection is a set of documents. A document consists of the document's content, and additional information such as document name (identifier) and access policies. Particularly, our documents can be represented by a 6-tuple (D, V, R, W, O, K). Components D and V of the tuple identify the document: D is the document's name and V is the document's version number, meaning V counts the number of times the document has been modified. Components R, W, and O represent the document's access policies: R is the policy for read access, W is the policy for write access, and O is the policy for ownership access. Component K represents the document's content as a multi-set of keywords contained in the document. A person that has read access to a document can access its content. A person that has write access to a document is capable of modifying the document's content, creating a new document of the same name but different version number, and defining the new document's read policy. A person that has ownership access to a document is capable of creating a new document of the same name but different version number, defining the new document's write and ownership policies. Particularly, ownership access to a version of a document does not imply write access to that version of the document.

Example rights and document collection. We now turn to our example, showing why documents may be excluded from search results due to access restriction, even though the excluded documents contain the searched keywords. Our example features three users Alice, Bob, and Charly, who hold access rights $\{a\}$, $\{b\}$, and $\{b, c\}$, respectively. Our example document collection consists of four document (versions)

- $d_{1,1} = (\mathrm{D}{:}1, \mathrm{V}{:}1, \mathrm{R}{:}\mathbb{A}(b), \mathrm{W}{:}\mathbb{A}(a), \mathrm{O}{:}\mathbb{A}(b), \mathrm{K}{:}\{w_1, w_2\})$,
- $d_{2,1} = (\mathrm{D}{:}2, \mathrm{V}{:}1, \mathrm{R}{:}\mathbb{A}(b), \mathrm{W}{:}\mathbb{A}(b), \mathrm{O}{:}\mathbb{A}(c), \mathrm{K}{:}\{w_1\})$,
- $d_{3,1} = (\mathrm{D}{:}3, \mathrm{V}{:}1, \mathrm{R}{:}\mathbb{A}(a), \mathrm{W}{:}\mathbb{A}(a), \mathrm{O}{:}\mathbb{A}(b), \mathrm{K}{:}\{w_1, w_3\})$, and
- $d_{1,2} = (\mathrm{D}{:}1, \mathrm{V}{:}2, \mathrm{R}{:}\mathbb{A}(c), \mathrm{W}{:}\mathbb{A}(a), \mathrm{O}{:}\mathbb{A}(b), \mathrm{K}{:}\{w_2, w_3\})$,

where $\mathbb{A}(i)$ is an access policy that is satisfied by access right i for $i \in \{a, b, c\}$.

The need to respect access restrictions. We use this example document collection to illustrate the aforementioned need to consider access restrictions during search result computation: consider Alice to search for keyword w_1, and suppose, the search result presented to Alice contains document version $d_{1,1}$ despite Alice not being allowed to read that document. Then, Alice would have learned some information on the contents of document $d_{1,1}$, namely, the information that the document contains w_1. This however, contradicts the purpose of access control, which was put in place in order to prevent Alice from learning such information.

Concerning access control, we point out that all measures taken to enforce access restrictions must be collusion resistant. That is, if multiple users come together, they are still unable to access any data that neither of the colluding users can access on her own. As an example, assume a document that requires access rights $\{a\}$ and $\{c\}$ in order to gain read access. Then, even if our example users Alice and Charly work together, they cannot access the document, even though together they hold the necessary rights.

Security of searchable encryption. The setting of searchable encryption asks for document collections to be stored and searched remotely, for example, in the cloud. As outlined above, in such settings, it is advisable to not trust the remote server. Therefore, we need means to verify that the server indeed provides correct search results.

With updatable document collections, we also need to make sure that the option to update the collection cannot be abused. The update mechanism can be abused in several ways and by various parties. For example, in a multi-user scenario, a malicious user may attempt to harm the collection and prevent the server from serving future search requests. On the other hand, in some scenarios, the server is able to update the searchable document collection, for instance, with email. The server may abuse the update mechanism in order to gather information on what users have searched for, and thus threaten the privacy of user's search requests. This, in turn, leads to the server learning information about the contents of the document collection, from knowing what has been searched for, and the fact that search results have not been empty.

Our contribution. Our contributions to the research on searchable encryption is two-fold. First, we have devised SEAC, which is the first searchable encryption scheme with access control that uses a backward index structure for search and includes access control in that index structure. SEAC has first been presented at ARES 2017 [Lök17]. Previous approaches to searchable encryption with access control use a different type of index structures, namely, forward indexes, or apply access control separate from search, for example to filter out unsuited documents from search results.

Second, we construct dSEAC, which builds upon SEAC and improves it in multiple ways. For example, SEAC is a static scheme, which does not allow the document collection to be extended or modified after it has been set up. On the other hand, dSEAC is dynamic. dSEAC also satisfies the security notions of fork consistency, verifiability, and forward privacy. This makes dSEAC the first dynamic searchable encryption scheme with access control that satisfies all these notions simultaneously. The dSEAC scheme has been published at FPS 2019 [BL19a].

As minor contributions, we present two more searchable encryption schemes, RLS and DIA, that address some shortcomings in dSEAC. As byproducts of our research on searchable encryption, we have also made contributions to the research on attribute-based encryption and signature schemes. For SEAC, we have developed a novel notion of attribute-based encryption called authority customization for multi-authority attribute-based encryption (MA-ABE), although the version of SEAC we present in this thesis does not make use of multi-authority attribute-based encryption with authority key customization. For dSEAC, we have developed the novel notion of signature schemes with delayed verifiability.

Overview of Part II. In the upcoming chapter, we take a look at the state of the art in searchable encryption research and its relation to our research. This includes a brief discussion of the history of searchable encryption and the main approaches to searchable encryption. The chapter also includes discussions of concepts and schemes from the literature that have significant influence on our constructions of searchable encryption schemes.

In Chapter 6, we formalize the notions and properties discussed above and present the SEAC and dSEAC searchable encryption schemes that have the desired properties. Both schemes are discussed in great detail.

Discussing some shortcomings of the previous schemes, in Chapter 7, we present and explore the additional RLS and DIA constructions of searchable encryption schemes that attempt to address the shortcomings using techniques from the literature. While one of the attempts is not very successful, the other scheme addresses the shortcomings, but has drawbacks of its own. We then briefly compare our constructions, discussing the trade-offs they make.

We conclude our discussion of searchable encryptions with some closing remarks in Chapter 8.

5 Searchable Encryption: Related Work and Influential Schemes

The literature on searchable encryption features many recurring concepts and approaches. In this chapter, we address these recurring themes and relate them and the state of the art to our own research on searchable encryption.

To that end, we present in the general approaches to searchable encryption in the upcoming section. The concept of leakage — the information an observer or participant may learn from searchable encryption — and the classification of leakage is presented in Section 5.2. Leakage classification plays an important role in comparing searchable encryption schemes and evaluating their real-world security.

In Sections 5.3–5.6, we briefly present some searchable encryption schemes, focussing on their concepts, data structures, and design choices, but also discussing how these schemes have influenced our own searchable encryption schemes. The schemes presented in this chapter allow us see the general approaches to searchable encryption in action. They also allow us to illustrate the data structures used in searchable encryption and how the data structures relate to each other. Finally, we present a brief general overview of the state of the art, relating it to our own research.

5.1 Approaches to Searchable Encryption

The majority of searchable encryption schemes can be broadly classified into two categories. Both types of schemes use index structures to enable search on encrypted data. There are searchable encryption schemes that rely on forward indexes, and schemes that rely on backwards indexes. In a *forward* index, a document is mapped to the set of keywords contained in the document. Conversely, in a *backward or inverted* index, a keyword is mapped to the set of documents that contain the keyword. Both types of index structures have inherent advantages and disadvantages that make them suitable in certain application scenarios, but unsuitable in other scenarios.

For example, a forward index, mapping documents to keyword sets, generally allows for searchable document collections to be updated efficiently: compute an index for the new document, and add the document and its index to the document collection. Particularly, the index of the new document is independent of anything presently contained in the document collection. However, when searching for a keyword, one needs to check every document's index for the searched keyword. As a result, forward indexes are well-suited in scenarios with uncertainties about communication partners or the contents of the searchable document collection. Prime examples are multi-user scenarios in which the set of users that may access the data is not fixed or unknown at the time the index is computed, for instance, a document collection that encompasses all of an organization's storage system. Another good example scenario for forward indexes is encrypted emails, because the person that writes a searchable email does not need to know the document collection, which consists of the addressee's other searchable emails.

In contrast to forward indexes, a backward index, mapping keywords to document sets, are harder to maintain during updates to the searchable document collection. This is because updates alter the already existing index structure, so the whole existing collection has to be taken into consideration when performing an update. On the other hand, if the mapping from keywords to document sets is realized using appropriate data structures, backward indexes generally allow for very efficient search. Therefore, backward indexes are appropriately used in scenarios that feature static document collections, or in which search is much more frequent than updates to the searchable document collection. An example scenario is document collections that span an organization's archive.

A third category of searchable encryption schemes may be identified: schemes that rely on (search) trees as their index structures explicitly, rather than just allowing the index structure to be implemented as a tree. However, the number of schemes in this potential category is rather low. It can also be argued that tree-based searchable encryption schemes are a sub-category in the class of schemes relying on inverted indexes, because the tree structure is typically used to implement an inverted index of some kind. The tree structure introduces some inefficiencies in the search procedure, typically involving logarithmic factors as overhead, but the update procedure benefits from the tree structure and makes updating the index more efficient when compared to other implementations of inverted indexes.

5.2 Leakage in Searchable Encryption

In the introduction to searchable encryption, Chapter 4, we state that the server that stores and processes the document collection in searchable encryption learns no information about the collection. It is hardly possible to construct searchable encryption schemes that achieve this degree of secrecy while retaining any practical degree of efficiency. Instead, most schemes accept that the server learns some information over time, but hope that the information leakage to the server is insignificant.

The information leaked by a searchable encryption scheme is captured in so-called leakage functions. There is one such function for each type of operation that the server participates in, and each function relates what the server can learn from the respective operation to the parameters of the operation, for example, the keyword that is searched for or the document that gets added to the document collection.

Leakage functions play a role in security definitions, but also allow searchable encryption schemes to be compared based on their leakage. However, leakage varies slightly between schemes, making comparisons hard. In order to solve this issue, Cash et al. [Cas+15] have developed a framework that classifies searchable encryption schemes into four classes $L_1, \dots, L_4$; the classification glances over minor differences in leakage specifications, and instead focuses on certain characteristics of leakage functions. These characteristics then relate to the potential damage that can arise from an attacker that abuses the characteristics, as well as how hard it is to mount such attacks. It is clear that the more information is leaked, the larger the potential for abuse, and the larger the potential damage of an attack.

Searchable encryption schemes classified L_1 leak the least information to the server; schemes classified L_4 leak the most information. With all leakage classes, the server, which is seen as the adversary, learns, for every keyword in a document collection, a substitution of the keyword. Such a substitution can be computed from a pseudorandom function, c.f. Definition 3.

L_1: Cash et al. characterize L_1 leakage as "query-revealed occurrence pattern," [Cas+15]. That is, the server can identify the set of documents from the collection that contain a keyword *kw* only after a search query for *kw* has been issued.

L_2: This class captures schemes that fully reveal keyword occurrence patterns [Cas+15]. With schemes in this class, the server can, for every keyword *kw*, identify the set of documents from the collection that contain *kw* as soon as the document collection is stored at the server.

L_3: Schemes in this class are characterized as fully revealing keyword occurrence patterns and keyword order in documents [Cas+15]. Hence, the server can, for every keyword *kw*, identify the set of documents from the collection that contain *kw* as soon as the document collection is stored at the server. Additionally, for every document from the collection, the server can determine the order of keywords' first occurrence in the document.

L_4: Cash et al. describe this class as leaking "full plaintext under deterministic word-substitution cipher," and describe what the server learns as "the pattern of locations in the text where each word occurs and its total number of occurrences in the document collection" [Cas+15]. Therefore, the server can, for every keyword *kw*, identify the set of documents from the collection that contain *kw* as soon as the document collection is stored at the server. Furthermore, for every document from the collection, the server can determine the orders of keywords' occurrences in the document.

The difference between L_1 and L_2 leakage is the moment at which the leakage occurs, upon search or at setup. Leakage L_2 differs from L_3 in that the order of keyword's first occurrence remains hidden from the server in L_2 schemes, but is revealed to the server in L_3 schemes. In contrast to L_3 schemes, L_4 schemes reveal the order not only of each keyword's first occurrence in a document, but also for every subsequent occurrence. We stress that with all leakage classes the server never learns the actual keywords, but only their substitutions.

(Un-)surprisingly, the leakage of schemes, and thus the schemes' classifications, can be traced to the fundamental design decisions on how the schemes achieve searchability of document collections. For example, the naïve approach of deterministically substituting every word in a document by some (pseudorandom) string that depends on the original word results in schemes from class L_4. Searchable encryption schemes that rely on encrypted inverted indexes generally fall into class L_1. Most schemes presented in this thesis follow the encrypted inverted index approach, and thus belong to class L_1, and thus, within Cash et al.'s leakage hierarchy, achieve the least leakage that can be achieved by schemes that are practical in therms of their performance.

As hinted at above, there are searchable encryption schemes that are hardly practical, but achieve even less leakage than schemes from class L_1. Such schemes rely on techniques such as oblivious RAM [GO96]; indeed, the original work on oblivious RAM explains how secure search on a remotely stored document collection can be achieved, even though the notion of searchable encryption would only be established several years later. However, Naveed extensively discusses why combining searchable encryption and oblivious RAM is a "fallacy" [Nav15].

There are alternatives to oblivious RAM that have been proposed in order to suppress leakage [KMO18; KM19]. However, it remains to be seen whether these techniques are practical and applicable. Particularly, we leave the application of these techniques to our constructions as future work.

Nevertheless, the fact that schemes exist that achieve lower leakage than what is permitted by Cash et al.'s leakage hierarchy is an indication that the hierarchy is incomplete. Below, we give an overview of some proposed extensions of the hierarchy. We also have a look at the impact of leakage as a motivation for the leakage hierarchy and its extensions.

Consequences of leakage. Several papers study the practical impact of leakage by trying to abuse leakage in order to reconstruct a (plaintext) document collection or determine what terms users search for [IKK12]. It has been shown that reconstructing document collections is relatively easy for searchable encryption schemes from leakage classes L_2–L_4 [Cas+15; Gir+17]. Even searchable encryption schemes from leakage class L_1 may be successfully attacked if they are dynamic, meaning they allow for documents to be added to the collection, and do not provide any countermeasures [ZKP16]. Such countermeasures should make dynamic schemes forward private. A particular countermeasure is batch updates, which means that instead of adding one document at a time, multiple documents should be added simultaneously. It has also been shown that the iterative testing approach to search present in many searchable encryption schemes that rely on forward indexes can often be attacked with relative ease [RMÖ17]. Examples of schemes following the iterative testing approach are the PEKS scheme described below and its derivatives.

Extensions to the leakage hierarchy. Some sub-classes of leakage class L_1 have been proposed, most notably leakage classes L_1^+ [Gir+17] and L_{KWAP} [RMÖ17]. Class L_1^+ has the same leakage as class L_1 as soon as searches are being conducted, but the leakage at setup time, when the searchable document collection is initially stored at the server, is slightly different: while class L_1 schemes leak (an upper bound on) the number of keyword–document pairs in the document collection, L_1^+ schemes leak, for every keyword, the number of documents that contain the keyword. Class L_{KWAP} is a leakage class for schemes that seemingly leak the same information as L_1 schemes, but follow the iterative testing approach mentioned in the previous paragraph. The proposals of classes L_1^+ by Giraud et al. [Gir+17] and L_{KWAP} by van Rompay et al. [RMÖ17] are accompanied by examples of leakage abuse attacks against the respective class. These attacks are generically applicable to schemes from the respective class, but not to other schemes from class L_1.

5.3 The PEKS Scheme

Boneh et al. have presented a searchable encryption primitive, *public key encryption with keyword search,* or PEKS for short [Bon+04], that provides searchable encryption in a multi-writer, single-reader scenario, with the particular example of encrypted e-mail in mind. The implementation of PEKS proposed by Boneh et al. relies on the forward index and iterative testing approaches, as outlined above.

Index structure. Conceptually, the forward index is realized as a bunch of tags that are attached to a document ciphertext; one tag for each keyword that is contained in the corresponding plaintext document. A tag is computed from a keyword, some randomness, and the addressee's public key by applying hash functions and bilinear maps. During search, the searched keyword and the addressee's private key are iteratively combined with

Key Generation: output $(sk,pk)=(\alpha,(g,h=g^{\alpha}))$
Tag Generation: output $(t_1,t_2)=(g^r,H_2(e(H_1(kw),h^r)))$
Query Generation: output $q=H_1(kw)^{\alpha}$
Test for Match: output 1 if $H_2(e(q,t_1))=t_2$, otherwise output 0

Figure 5.1: The PEKS implementation proposed by Boneh et al. Function e is a bilinear map; H_1 and H_2 are hash functions with appropriate domains.

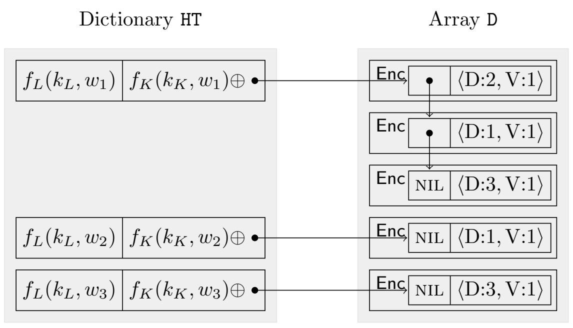


Figure 5.2: The index structure used in the CGKO scheme. A dictionary HT and a number of encrypted linked lists stored in an array D. Since the CGKO scheme does not allow the searchable document collection to be updated, only the first version of each document from our example document collection, see Page 38, is present in the shown example.

tags' randomness by applying the bilinear map, and the result is then compared to the respective tag. If the result of the computation equals part of the tag, then the document ciphertext associated with the tag is added to the search result.

Abdalla et al. [Abd+08] provide a study of the PEKS primitive and its derivatives, particularly focussing on its consistency property, and expand on the relation between PEKS and identity-based encryption already observed in the original work on PEKS by Boneh et al. [Bon+04].

Influence of PEKS. The work of Boneh et al. has been very influential regarding schemes that provide searchable encryption in settings that feature multiple data readers or writers. Many of those schemes exchange the public key (ElGamal [ElG85]) encryption of PEKS for attribute-based encryption, additionally achieving access control in the process.

5.4 The CGKO Scheme

The work of Curtmola et al. [Cur+11] (conference version: [Cur+06]) has had significant impact on the design of searchable symmetric encryption (SSE) schemes. Particularly Curtmola et al. have presented security notions that must currently be considered standard. Their proposals have also influenced the actual construction of SSE schemes, for example, in terms of data structures used in searchable encryption schemes.

Data structures, conceptually. Curtmola et al.'s SSE-1 scheme (hereafter called CGKO scheme) relies on a backward index that consist of two data structures, as shown in Figure 5.2. There is a dictionary HT that contains an entry for each keyword from the searchable document collection. The label of a keyword's HT entry is derived from the keyword using a pseudorandom function. Similarly, a pseudorandom function is applied to the keyword to obtain an encryption key that is used to encrypt the HT entry's value. The value itself is a pointer to a keyword dependent linked list that is stored in some memory array D. The array D is shared among all such linked lists. Each node of the D list for some keyword *kw* points to a document that contains keyword *kw*. D lists are encrypted in a node-by-node fashion. A node's decryption key is stored as part of the pointer to that list node. Hence, the key is either stored as part of the successor pointer of another list node or as part of the pointer to a list head, which is stored in an HT entry.

Whenever a user wants to search the document collection for some keyword, she applies pseudorandom functions to the keyword in order to reconstruct the keyword dependent label and decryption key for the keyword's HT entry. The user then sends the label and key for the search query to the server. The server uses the key to decrypt the value of the HT entry identified by the label, and then uses the pointer to a D list to recover the (plaintext) D list in a node-by-node fashion. The server then sends its response — the search result — to the user. The search result consist of the set of documents that the decrypted D list points to.

Data structures and design choices. In the CGKO scheme, in addition to keyword dependent entries, the dictionary HT contains dummy entries that are indistinguishable from keyword dependent HT entries that have not yet been accessed. Similarly, the memory array D contains dummy nodes that are indistinguishable from list nodes that have not been decrypted yet. When the index structure is computed, list nodes are assigned random free positions in array D. The order of document pointers in any D list is independent of the order in which the corresponding documents have been processed during computation of the index structure.

These measures serve to limit the information a server can gain about the underlying document collection, both before and after a search request has been served. Dummy entries in HT and D prevent the server from knowing the exact number of keywords in the document collection, and prevent the server from knowing whether search requests for all keywords have been made. Pseudorandom labels of HT entries prevent the server from learning the (plaintext) keywords from the document collection. Encryption of list nodes prevents the server from learning whether documents share any keyword, at least until a search request is served and both documents occur in the search result.

Influence and usage of CGKO. The CGKO scheme has influenced subsequent proposals of searchable encryption schemes, particularly concerning the index structures that the schemes rely on. This holds true for many of the schemes presented in this chapter and throughout the thesis, for example, the schemes in Sections 5.5 and 5.6, the schemes in Chapters 6 and 7, but also for other schemes from the literature, not discussed in detail, like the proposals of Kamara et al. [KPR12] and Cash et al. [Cas+14].

5.5 The Σοφος Scheme

Bost's Σοφος scheme [Bos16] is the first dynamic forward-private SSE scheme whose efficiency is comparable to SSE schemes that do not feature forward privacy, like the dynamic version of the CGKO scheme presented in [KPR12]. Σοφος is dynamic in that the

Dictionary HT

$f_L(k_{w_1}, \pi_S^{-2}(sk_S, seed_{w_1}))$	$f_K(k_{w_1}, \pi_S^{-2}(sk_S, seed_{w_1})) \oplus \langle \text{D:2}, \text{V:1} \rangle$
$f_L(k_{w_2}, \pi_S^{-1}(sk_S, seed_{w_2}))$	$f_K(k_{w_2}, \pi_S^{-1}(sk_S, seed_{w_2})) \oplus \langle \text{D:1}, \text{V:1} \rangle$
$f_K(k_{w_1}, \pi_S^{-3}(sk_S, seed_{w_1}))$	$f_K(k_{w_1}, \pi_S^{-3}(sk_S, seed_{w_1})) \oplus \langle \text{D:3}, \text{V:1} \rangle$
$f_L(k_{w_1}, \pi_S^{-1}(sk_S, seed_{w_1}))$	$f_K(k_{w_1}, \pi_S^{-1}(sk_S, seed_{w_1})) \oplus \langle \text{D:1}, \text{V:1} \rangle$
$f_L(k_{w_3}, \pi_S^{-1}(sk_S, seed_{w_3}))$	$f_K(k_{w_3}, \pi_S^{-1}(sk_S, seed_{w_3})) \oplus \langle \text{D:3}, \text{V:1} \rangle$

Figure 5.3: The index structure used in the Σοφος scheme, computed from our example document collection from Page 38. A dictionary HT that stores multiple labels for each keyword from the document collection; the number of labels depends on the number of documents from the collection that contain the keyword.

scheme allows documents to be added to the searchable document collection over time, rather than setting the collection up once and for all.

Data structure, conceptually. Σοφος uses a dictionary HT as its search structure. The dictionary is employed as a backwards index, as can be seen in Figure 5.3. For each keyword in a searchable document collection, HT holds multiple entries. Namely, there are as many HT entries for keyword *kw* as there are documents that contain *kw*: Intuitively, the Σοφος scheme integrates the D lists of the CGKO scheme into the dictionary HT.

For the computation of the dictionary, the occurrences of keywords are enumerated, and labels of HT entries depend on keywords, as well as a counter. The labels are derived from keywords and counters by application of a keyed function, for example, a pseudorandom or hash function. The key depends on the keyword. The counter determines how often the inverse of a trapdoor permutation is applied to a keyword dependent seed. The result of that computation on the seed is then passed as an argument to the function that determines HT entries' labels. The value stored in an HT entry for keyword *kw* is a pointer to a document that contains *kw*. The value is encrypted under a pseudorandom bitstring derived in the same way the label of the HT entry is derived, albeit using a different function.

The owner of a document collection keeps track of the keyword dependent keys, seeds, and occurrence counters, and stores this information as her state. When searching for keyword *kw*, the user iteratively applies the inverse of the trapdoor permutation to the seed for *kw* or the result of previous inversions. The total number of inversions equals *kw*'s occurrence counter. The result *tv* of the final iteration, together with *kw*'s keyword dependent key and occurrence counter are sent to the server. The server then uses *tv* and the keyword dependent key to derive labels and decryption keys for HT entries. After each derivation, the server applies the trapdoor permutation to *tv* in order to derive a new *tv* value. The occurrence counter tells the server how many iterations of this process to perform. The search result then is the set of documents pointed to by the decrypted HT entries.

When adding a new document to the document collection, the user, for each keyword in the new document, increases the keyword's occurrence counter from the user's state. As before, the user then applies the inverse of the trapdoor permutation on the keyword dependent seed multiple times. The result of that computation is used together with the keyword dependent key to derive the label and encryption key for an HT entry that points to the new document. That HT entry is then sent to the server for storage.

Throughout this thesis, we refer to the application of a trapdoor function in the above iterative manner as *"Bost's technique."*

Security and design decisions. The use of the trapdoor permutation makes the Σοφος scheme forward private, as the server is prevented from predicting the labels and decryption keys for future HT entries for any given keyword, even if that keyword has been searched for before. On the other hand, the trapdoor permutation makes Σοφος efficient in that the size of a query is independent of the result size. Particularly, the user does not need to compute and send all labels for the searched keyword, because the server can compute these by itself by repeatedly applying the trapdoor permutation to the appropriate value from the query.

Influence and usage of Σοφος. The Σοφος scheme has introduced Bost's technique, as discussed above. We use Bost's technique in some of our constructions to achieve the forward privacy security notion, which has been demonstrated to be crucial for searchable encryption schemes to be secure in practice [ZKP16]. Although Σοφος is a relatively recent proposal, alternative proposals that explicitly mention Σοφος have been made, specifically criticizing the use of the trapdoor function [Ete+18].

5.6 The AMR scheme

The searchable encryption scheme of Alderman et al. [AMR17] adapts Curtmola et al.'s single-user CGKO scheme [Cur+06], c.f. Section 5.4, to the multi-reader setting with a hierarchy of access levels. An example of such a hierarchy is the well-established hierarchy top secret > secret > eyes only. Most notably, the AMR scheme considers access restrictions to documents during search, so only users that are allowed to access a certain document may find that document in search results presented to them.

The main observation of Alderman et al. is that, in a setting with hierarchical access levels, the documents accessible to users from one level is a subset of the documents accessible to users from any higher level. Regarding the index structure of the CGKO scheme, the subset becomes a sub-list: the list that represents the pre-computed search result for one access level forms the tail of the list that represents the pre-computed search result for the next level higher up the hierarchy. The data structure that results from this change is shown in Figure 5.4

Where Curtmola et al.'s CGKO scheme stores, for each keyword, one dictionary entry that points to the encrypted result list associated with that keyword, AMR stores a dictionary entry for each keyword–access level pair. The result list in AMR contains document pointers not in completely random order. Instead, documents with equal access levels are grouped together and random order is maintained within each group. The result list features groups with high access levels closer to the list head. As a result, dictionary entries for a given access level point to a sub-list of the result list, and due to the hierarchical structure of access levels and the order of document groups in the list, users are allowed to access all subsequent sub-lists.

Access to sub-lists is restricted by keys given to users. Essentially, for every key used in the CGKO scheme, AMR features one version of the key per access level, as can be seen from Figure 5.4 and comparing it with Figure 5.2.

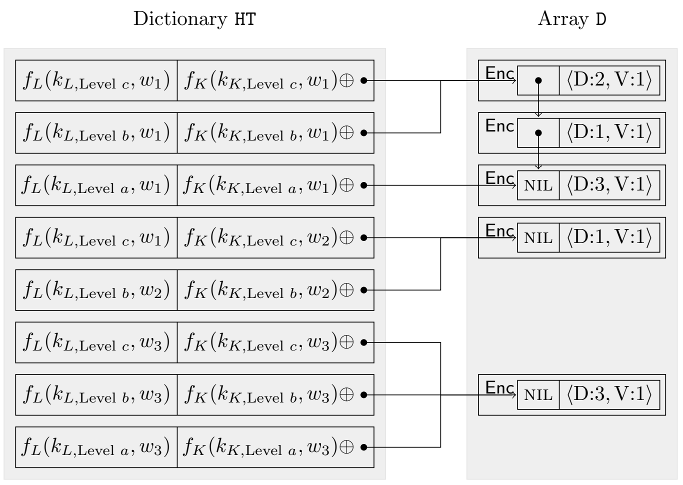

Dictionary HT
Array D
$f_L(k_{L,\text{Level } c}, w_1)$
$f_K(k_{K,\text{Level } c}, w_1) \oplus$
$f_L(k_{L,\text{Level } b}, w_1)$
$f_K(k_{K,\text{Level } b}, w_1) \oplus$
$f_L(k_{L,\text{Level } a}, w_1)$
$f_K(k_{K,\text{Level } a}, w_1) \oplus$
$f_L(k_{L,\text{Level } c}, w_1)$
$f_K(k_{K,\text{Level } c}, w_2) \oplus$
$f_L(k_{L,\text{Level } b}, w_2)$
$f_K(k_{K,\text{Level } b}, w_2) \oplus$
$f_L(k_{L,\text{Level } c}, w_3)$
$f_K(k_{K,\text{Level } c}, w_3) \oplus$
$f_L(k_{L,\text{Level } b}, w_3)$
$f_K(k_{K,\text{Level } b}, w_3) \oplus$
$f_L(k_{L,\text{Level } a}, w_3)$
$f_K(k_{K,\text{Level } a}, w_3) \oplus$
Enc
$\langle$D:2, V:1$\rangle$
Enc
$\langle$D:1, V:1$\rangle$
Enc
NIL
$\langle$D:3, V:1$\rangle$
Enc
NIL
$\langle$D:1, V:1$\rangle$
Enc
NIL
$\langle$D:3, V:1$\rangle$

Figure 5.4: The index structure used in the AMR scheme, computed from our example document collection from Page 38, assuming the hierarchy of access rights is $c > b > a$. See, for example, the result list for keyword w_1. All documents that contain w_1 are accessible to users holding at least access level b. Therefore, users holding at least access level b get access to a list containing all document that contain w_1, while users holding only access level a are restricted to the sub-list that only contains document $\langle$D:3, V:1$\rangle$.

Usage of AMR. One of our main results, the dSEAC searchable encryption scheme with access control from Chapter 6.3, achieves strong security guarantees. One drawback of the scheme is that, in order to achieve one of the security guarantees, verifiability, it has to leak the access policies of documents that contain a searched keyword. If every access policy is shared by many documents, the leakage does not pose a problem. If, however, there is essentially a one-to-one mapping of access policies to documents, the leakage is significant. We construct the RLS searchable encryption scheme that is based on the AMR scheme, and that mitigates the leakage of dSEAC at the cost of using AMR's simpler model of access control (hierarchical access levels) when compared to dSEAC's model. At the same time, RLS provides the same security guarantees that dSEAC provides.

5.7 Development of Searchable Encryption

Ever since its inception in the seminal work of Song et al. [SWP00], searchable encryption has been an actively researched topic. For a general discussion of the development of searchable encryption and its flavors, see the survey of Bösch et al. [Bös+14]. See the survey of Poh et al. [Poh+17] for an extensive discussion of SSE schemes and their properties, as well as an overview of the historic development of these schemes.

Searchable encryption schemes that rely on forward indexes have been proposed numerous times. Among proposed SSE schemes, early work mostly follows the approach of forward indexes [Goh03; CM05]. The majority of schemes with forward indexes implement searchable encryption in an asymmetric setting, so they feature multiple data writers or readers, [Bon+04; DRD08; PZ13; RMÖ15] amongst others. Occasionally, this type of searchable encryption is referred to as multi-user searchable encryption (MUSE).

As stated before, the work of Boneh et al. has had significant influence on MUSE, and their PEKS scheme has influenced numerous schemes providing searchable encryption in a multi-reader setting. Such schemes achieve access control by applying attribute-based encryption instead of public key encryption, for example, [LZ14; Sun+14; ZXA14; CD15; DGC15; Sun+16]. Several of the listed proposals explicitly deal with the challenge of revoking users' access rights in order to prevent users from accessing data after they have been corrupted.

Searchable encryption schemes that rely on backward indexes are often classified as SSE, although this does does not necessarily restrict these schemes to the single user setting. Examples of SSE schemes are numerous, both for schemes that explicitly use backward indexes [Cur+06; Lie+10; Cur+11; KPR12; KO12; KO13; Cas+14; LZC15; Ash+16; Bos16; GMP16; Ete+18], as well as schemes with tree index structures [KP13; SPS14; Xia+16]. Typically, one would expect SSE to be restricted to the single user setting, but this restriction can be mitigated by applying broadcast encryption, as proposed by Curtmola et al. [Cur+11], by applying dedicated cloud architectures [BL18], or by carefully modifying the index structure [AMR17]. The latter proposals even achieve access control. The principles and data structures from SSE and inverted indexes can also be combined with attribute-based encryption [KB14; Lök17; BL19a; Mic19].

The research mentioned above mainly focuses on searchable encryption schemes that allow users to search for single keywords. Nevertheless there is significant body of work aimed at allowing richer types of search queries, for example, Boolean queries, which allow conjunctions and disjunction of keywords. Boolean queries are subject to research that focusses on minimizing leakage of such schemes in comparison to the naïve approach, which performs single keyword search for each keyword involved in the query and then computes the intersections or unions of result sets as described in [Fer+18]. Examples of research into these queries can be found in [Cas+13; Öre+16].

Other types of rich queries are similarity search and range queries. Searchable encryption supporting such queries is presented in [ÖAS15; ÖKS13]. However, it must be noted that supporting rich query types potentially worsens the consequences of leakage, as demonstrated in [Gru+18; MT19] for the case of range queries; the attacks are practical and achieve (full) database reconstruction.

Chase and Kamara have established a line of research that has branched off of searchable encryption and aims to generalize it. Their research is into *structured encryption* [CK10]. In structured encryption, data is seen as structured. For example, data may be organized in a dictionary, graph (list, tree, ...), or relational database, and queries to the structured data can be answered in a privacy preserving, yet meaningful way. In this mindset, searchable encryption schemes can be seen as structured encryption schemes for certain simple classes of structures that support keyword search. However, structured encryption schemes provide solutions for more complex types of structures and queries like adjacency queries on general graphs [CK10].

Throughout this thesis, we discuss some of the searchable encryption schemes from this chapter in more detail. This is either because these schemes are very influential or serve as a basis for our own constructions, c.f. Sections 5.3–5.6, or are construction of our own, like SEAC [Lök17] in Section 6.2, dSEAC [BL19a] in Section 6.3, and the RLS and DIA schemes in Section 7.

6 Searchable Encryption with Access Control

As we have pointed out in Chapter 4, search and access control must be coupled tightly. Otherwise, search results can easily be used to circumvent access control. The problem of coupling search and access control is addressed by two of our main results, which we present in this chapter. Our searchable encryption schemes consider access control during the search process in such a way that even the server that performs search on users' behalf is unable to bypass access restrictions.

We construct our searchable encryption scheme in multiple steps. First, we present SEAC, a *static* searchable encryption scheme with access control [Lök17]. The SEAC scheme is static in that the searchable document collection is set up once and for all, and cannot be updated. The restriction to a static document collection has implications for the usage scenario. Particularly, there is only one owner/data writer, and the owner can be assumed to behave honestly. Hence, for security of SEAC, we only need to consider the server and users/data readers to be adversarial, and for them to collude amongst each other.

As a second step, we extend upon SEAC in multiple dimensions. The ultimate goal of these extensions is to allow us to use SEAC in constructing a searchable encryption scheme with access control in the multi-writer setting, which is inherently dynamic. Dynamics and the multi-writer setting come with security requirements that our extensions address.

Our final step is to construct dSEAC, a *dynamic* searchable encryption scheme with access control [BL19a]. dSEAC integrates multiple (extended) SEAC instances, one instance per update. However, simply having multiple SEAC instances is insufficient, and we have to take additional measures in order to achieve proper integration. For example, it would be somewhat impractical if users needed to get new or additional cryptographic keys for every SEAC instance.

Before we turn to our searchable encryption schemes in Sections 6.2 and 6.3, we formalize the notion of searchable encryption and define its security properties in Section 6.1.

6.1 Preliminaries and Definitions

This section focusses on a formalization of searchable encryption and its security. Our formalization revolves around two main operations, namely, a Search protocol by which a user requests a search result from the server and the server provides the requested result to the user, as well as an Update operation by which users, in cooperation with the server, can extend the searchable document collection stored at the server by new or updated documents.

We have additional operations for setting up system parameters, enrolling users to the system, and initializing the server. Due to the security model we consider, we also need a Resolve operation by which the server can blame a malicious user's actions on that user. Via the Resolve operation and the notifications it outputs, a judge certifies that an action has been taken by a malicious user. Our definition features a Version operation that helps in formalizing some security notions in a way similar to the notions for append-only authenticated dictionaries (AADs), c.f. Section 3.5.

Definition 26. *A* searchable encryption scheme with access control *is a tuple* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *of two PPT algorithms, and five protocols.*

Setup *takes a security parameter 1^λ and outputs system parameters pp, a key issuer's master secret msk, a judge's secret key jsk, and a judge's state st_{judge}. The algorithm is executed by a trusted party.*

UserJoin *involves a* prospective user *and the* key issuer. *Both parties take in the system parameters pp; the user also takes in her desired attribute set U; the key issuer also takes in her private key msk. The user privately outputs a secret key usk and state st_{user}. The key issuer publicly outputs a user certificate crt.*

Init *involves a* user *and the* server. *Both parties take in the system parameters pp; the user also takes in her secret key usk and state st_{user}; the server takes in its state st_{server}. Both parties output updated states.*

Update *involves a* user *and the* server. *Both parties take in the system parameters pp; the user also takes in her secret key usk, state st_{user}, and a document collection DC; the server also takes in its state st_{server}. The server may reject the update. Both parties output updated states.*

Search *involves a* user *and the* server. *Both parties take in the system parameters pp; the user also takes in her secret key usk, state st_{user}, and a keyword kw; the server also takes in its state st_{server}. Both parties output updated states; the user also outputs a search result X or an error symbol $\perp$.*

Resolve *involves the* server *and the* judge. *Both parties take in the system parameters pp; the server also takes in its state st_{server}; the judge also takes in her secret key jsk and her state st_{judge}. The server outputs an updated server state; the judge publicly outputs a number of notifications ntf, and privately outputs an updated judge's state.*

Version *takes pp and user state st_{user}, and outputs an integer vector or an error symbol $\perp$.*

We require result correctness. *That is, for every* Setup*, for all secret keys usk of users enrolled via* UserJoin*, for every document doc contained in any document collection input to a non-rejected execution of* Update*, for every user state st_{user} of the enrolled user holding key usk, and every server state st_{server} that resulted from the given update operations, we have that if $(st'_{user}, X) \leftarrow \mathsf{Search}(pp, usk, st_{user}, kw | pp, st_{server})$, then*

1. *$X = \perp$, or*
2. *kw does not occur in doc, or*
3. *kw's read policy is not satisfied by usk, or*
4. *$doc \in kw$, or*

5. *there is a notification that originates from an execution of* Resolve *that bars doc from being included in X.*

Additionally, we require state correctness*. That is,*

1. *for every non-rejected* Update *operation, the post-*Update *user state of the user performing the* Update *has a greater version number than the user's pre-*Update *state, so all vector components in the post-*Update *state's version are at least as large as in the pre-*Update *state's version, and at least one vector component is strictly larger, and*

2. *that every search operation results in the same user state that the user that performed the most recent non-rejected* Update *operation held after the operation.*

For state correctness, the Resolve *operation is treated the same as the* Update *operation, exchanging the user state for the judge's state.*

Result correctness means that search fails or the reported search results contain all documents that contain the searched keyword, except those document that are not accessible to the user on whose behalf search is performed, as well as documents that are certifiably broken, as documented by a notification. The state correctness notion, like the Version algorithm, exist for technical reasons: They are used to define and prove security of searchable encryption schemes.

In the introduction to this chapter, we have discussed static and dynamic searchable encryption schemes. Definition 26 addresses dynamic searchable encryption with access control. The following definition restricts dynamic searchable encryption to static searchable encryption.

Definition 27. *A searchable encryption scheme* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *is called* static *if the* Update *operation can be called at most once,* Init *and* Update *take master secret msk instead of a user's secret usk, and* UserJoin *cannot be called before* Update *has been called; the key issuer keeps a state* st_{issuer} *that is additionally output by* Setup*, that replaces the user state in operations* Init *and* Update*, and that is an additional input to operation* UserJoin*.*

In the definition of static searchable encryption, the key issuer's state is present for technical reasons. The state allows the key issuer to keep track of operations and to enforce the order of operations laid out in the definition.

Both, static and dynamic searchable encryption schemes must not only be correct, but also secure. We address this by four security notions, namely, data confidentiality, forward privacy, verifiability, and fork consistency. Data confidentiality means that the server that performs search on users' behalves does not learn more than some specified leakage, even if the server behaves maliciously and colludes with dishonest users. Forward privacy means that the server cannot use past search queries to obtain information about updates that occurred after the query has been made. Verifiability means that result correctness holds even in the presence of a malicious server. Fork consistency means that if two users hold distinct states, then one is a successor of the other or in the future they will never hold states such that one of these states is a successor of the other.

Data confidentiality. As noted in Chapter 5.2, security, and more specifically, data confidentiality of searchable encryption schemes is defined relative to leakage functions. These functions describe what an adversary (controlling the server and dishonest users) can learn from participating in searchable encryption. An adversary cannot learn more than the leakage if the adversary is unable to distinguish a real instantiation of the searchable encryption scheme that uses real data chosen by the adversary from a simulated instantiation that only relies on the leakage of the adversarially chosen data.

Since leakage functions describe what an adversary can learn from participating in searchable encryption, there is one leakage function for every type of operation that can be observed by the adversary. The adversary observes an operation either because the adversary participates as the server or as a dishonest user, or because the operation produces publicly observable outputs such as the user certificate *crt*. The leakage functions also consider the information gain from corrupting a once honest user.

As a result, we have seven leakage functions ($\mathcal{L}_{\text{HonestJoin}}, \mathcal{L}_{\text{CorruptJoin}}, \mathcal{L}_{\text{Corruption}}, \mathcal{L}_{\text{Init}}, \mathcal{L}_{\text{Update}}, \mathcal{L}_{\text{Search}}, \mathcal{L}_{\text{Resolve}}$). Functions $\mathcal{L}_{\text{HonestJoin}}$ models what can be learned from enrolling honest users. Function $\mathcal{L}_{\text{CorruptJoin}}$ models what can be learned from enrolling dishonest users. Function $\mathcal{L}_{\text{Corruption}}$ describes what can be learned from corrupting honest users. Functions $\mathcal{L}_{\text{Init}}$, $\mathcal{L}_{\text{Update}}$, $\mathcal{L}_{\text{Search}}$, and $\mathcal{L}_{\text{Resolve}}$ model what the adversary can learned from participating in operations Init, Update, Search, and $\mathsf{Resolve}$, respectively; in these cases the user or judge that the adversary interacts with behaves honestly.

Note that leakage functions are stateful with a shared state. That means that the evaluation of one leakage function depends on the inputs to all previous evaluations of all of the functions. Notation-wise, we keep the shared state implicit.

We define data confidentiality of searchable encryption schemes with access control relative to two experiments $\mathbf{Exp}^{\text{SE-conf-real}}$ and $\mathbf{Exp}^{\text{SE-conf-sim}}$, as shown in Figure 6.1. Experiment $\mathbf{Exp}^{\text{SE-conf-real}}$ represents a real instantiation of a searchable encryption scheme, while experiment $\mathbf{Exp}^{\text{SE-conf-sim}}$ represents a simulation of an instantiation. The adversary can interact with the real instantiation and the simulator, respectively, via oracles. Oracles correspond to leakage functions, so there are oracles for enrolling honest and dishonest users, corrupting users, for causing honest users to initialize, update, or search the remote document collection, and for resolving conflicts. In our depiction of experiment $\mathbf{Exp}^{\text{SE-conf-sim}}$, we denote that the simulator $\mathcal{S}$ provides functions that correspond to oracles; for the sake of notational consistency, the simulator also provides a function $\mathcal{O}_{\text{Setup}}$.

Using our experiments from Figure 6.1, we formalize data confidentiality by adapting the security notion of Curtmola et al. [Cur+11] to the dynamic multi-user setting. Our formalization is as follows.

Definition 28. *A searchable encryption scheme with access control* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *provides* data confidentiality *relative to leakage functions* ($\mathcal{L}_{\mathit{HonestJoin}}, \mathcal{L}_{\mathit{CorruptJoin}}, \mathcal{L}_{\mathit{Corruption}}, \mathcal{L}_{\mathit{Init}}, \mathcal{L}_{\mathit{Update}}, \mathcal{L}_{\mathit{Search}}, \mathcal{L}_{\mathit{Resolve}}$) *if, for all* PPT *adversaries* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that*

$$|\Pr[\mathbf{Exp}_{\mathcal{A},\Pi}^{\textit{SE-conf-real}}(\lambda) = 1] - \Pr[\mathbf{Exp}_{\mathcal{A},\mathcal{S},\Pi}^{\textit{SE-conf-sim}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda),$$

where the probabilities are taken over the random choices of the adversary and the simulator.

Fork consistency. As stated before, fork consistency means that if two honest users hold distinct states, then one state is a successor of the other or the users will never hold states such that one of them is a successor of the other. The notion is defined relative to experiment $\mathbf{Exp}^{\text{SE-fc}}$, as shown in Figure 6.2. The experiment plays two honest users, while the adversary gets to play the server and an arbitrary number of additional users. The honest users played by the experiment start out with identical initial states and then perform two sequences of operations using inputs chosen by the adversary. The adversary

$\mathbf{Exp}^{\text{SE-conf-real}}_{\mathcal{A},\Pi}(\lambda)$

- $(pp) \leftarrow \mathsf{Setup}(1^\lambda)$
- $b \leftarrow \mathcal{A}^{\mathcal{O}_{\text{HonestJoin}}(\cdot),\mathcal{O}_{\text{CorruptJoin}}(),\mathcal{O}_{\text{Corruption}}(\cdot),\mathcal{O}_{\text{Init}}(\cdot),\mathcal{O}_{\text{Update}}(\cdot,\cdot),\mathcal{O}_{\text{Search}}(\cdot,\cdot),\mathcal{O}_{\text{Resolve}}()}(pp)$, where the experiment reacts to oracle queries as follows:
  - $\mathcal{O}_{\text{HonestJoin}}$ with argument U: run protocol $\mathsf{UserJoin}$, playing the prospective user and the key issuer; the user takes attribute set U as her input; remember the user's key and state
  - $\mathcal{O}_{\text{CorruptJoin}}$ run protocol $\mathsf{UserJoin}$, playing the key issuer; the prospective user is played by $\mathcal{A}$; mark the user certificate *crt* that results from the protocol execution as corrupt
  - $\mathcal{O}_{\text{Corruption}}$ with argument *crt*: if no run of protocol $\mathsf{UserJoin}$ has resulted in certificate *crt* or *crt* is marked as corrupt, output $\perp$; otherwise output the key and state of the user that participated in the run of protocol $\mathsf{UserJoin}$ that resulted in *crt*; mark *crt* as corrupt
  - $\mathcal{O}_{\text{Init}}$ with argument *crt*: if no run of protocol $\mathsf{UserJoin}$ has resulted in certificate *crt* or *crt* is marked corrupt, output $\perp$; otherwise run protocol Init playing the user that participated in the run of protocol $\mathsf{UserJoin}$ that resulted in *crt*; remember the updated user state; the server is controlled by $\mathcal{A}$
  - $\mathcal{O}_{\text{Update}}$ with arguments (crt,DC): if no run of protocol $\mathsf{UserJoin}$ has resulted in certificate *crt*, *crt* is marked corrupt, or DC is not a document collection, output $\perp$; otherwise, run protocol Update playing the user that participated in the run of protocol $\mathsf{UserJoin}$ that resulted in *crt*; use document collection DC as the user's input; remember the updated user state; the server is controlled by $\mathcal{A}$
  - $\mathcal{O}_{\text{Search}}$ with arguments (crt,kw): if no run of protocol $\mathsf{UserJoin}$ has resulted in certificate *crt*, *crt* is marked corrupt, or *kw* is not a keyword, output $\perp$; otherwise, run protocol Search playing the user that participated in the run of protocol $\mathsf{UserJoin}$ that resulted in *crt*; use keyword *kw* as the user's input; remember the updated user state; the server is controlled by $\mathcal{A}$
  - $\mathcal{O}_{\text{Resolve}}$ run protocol $\mathsf{Resolve}$ playing the judge; the server is controlled by $\mathcal{A}$
- the experiment outputs b.

$\mathbf{Exp}^{\text{SE-conf-sim}}_{\mathcal{A},\mathcal{S},\Pi}(\lambda)$

- $pp \leftarrow \mathcal{S}.\mathcal{O}_{\text{Setup}}(1^\lambda)$
- $b \leftarrow \mathcal{A}^{\mathcal{O}_{\text{HonestJoin}}(\cdot),\mathcal{O}_{\text{CorruptJoin}}(),\mathcal{O}_{\text{Corruption}}(\cdot),\mathcal{O}_{\text{Init}}(\cdot),\mathcal{O}_{\text{Update}}(\cdot,\cdot),\mathcal{O}_{\text{Search}}(\cdot,\cdot),\mathcal{O}_{\text{Resolve}}()}(pp)$, where the experiment reacts to oracle queries as follows:
  - Upon call $\mathcal{O}_X(args)$, call $\mathcal{S}.\mathcal{O}_X(\mathcal{L}_X(args))$, where $X \in \{\text{HonestJoin},\text{CorruptJoin},\text{Corruption},\text{Init},\text{Update},\text{Search},\text{Resolve}\}$
- the experiment outputs b.

Figure 6.1: Data confidentiality experiments for searchable encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ with access control with leakage functions $(\mathcal{L}_{\text{HonestJoin}}, \mathcal{L}_{\text{CorruptJoin}}, \mathcal{L}_{\text{Corruption}}, \mathcal{L}_{\text{Init}}, \mathcal{L}_{\text{Update}}, \mathcal{L}_{\text{Search}}, \mathcal{L}_{\text{Resolve}})$, simulator $\mathcal{S}$, and adversary $\mathcal{A}$

$\mathbf{Exp}^{\text{SE-fc}}_{\mathcal{A},\Pi}(\lambda)$

- $(pp, msk, jvk, st_{\text{judge}}) \leftarrow \mathsf{Setup}(1^\lambda)$
- $(U_0, U_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\text{UserJoin}}()}(pp)$, where the experiment reacts to queries to oracle $\mathcal{O}_{\text{UserJoin}}$ by playing the key issuer's part of protocol $\mathsf{UserJoin}$ on input (pp, msk)
- $(usk_0, st_0 | crt_0) \leftarrow \mathsf{UserJoin}(pp, U_0 | pp, msk)$
- $(usk_1, st_1 | crt_1) \leftarrow \mathsf{UserJoin}(pp, U_1 | pp, msk)$
- $(i,j,k,l) \leftarrow \mathcal{A}_2^{\mathcal{O}_{\text{UserJoin}}(),\mathcal{O}_{\text{Init}}(\cdot),\mathcal{O}_{\text{Update}}(\cdot,\cdot),\mathcal{O}_{\text{Search}}(\cdot,\cdot),\mathcal{O}_{\text{Resolve}}()}(st_{\mathcal{A}}, crt_0, crt_1)$, where the experiments reacts to oracle queries as follows:
  - $\mathcal{O}_{\text{UserJoin}}$ run the key issuer's part of protocol $\mathsf{UserJoin}$ on input (pp, msk)
  - $\mathcal{O}_{\text{Init}}$ on input b: if $b \notin \{0,1\}$, output $\perp$; otherwise run protocol Init playing user b; $\mathcal{A}$ plays the server's part
  - $\mathcal{O}_{\text{Update}}$ on input b and DC: if $b \notin \{0,1\}$ or DC is not a document collection, output $\perp$; otherwise run protocol Update playing the user holding certificate crt_b using DC as the user's input; $\mathcal{A}$ plays the server's part
  - $\mathcal{O}_{\text{Search}}$ on input b and kw: if $b \notin \{0,1\}$ or kw is not a keyword, output $\perp$; otherwise run protocol Search playing the user holding certificate crt_b; $\mathcal{A}$ plays the server's part.
  - $\mathcal{O}_{\text{Resolve}}$ run the judges part of protocol $\mathsf{Resolve}$ on input $(pp, jsk, st_{\text{judge}})$

  For every query to $\mathcal{O}_{\text{Init}}$, $\mathcal{O}_{\text{Update}}$, and $\mathcal{O}_{\text{Search}}$, the experiment remembers the number of previous queries processed, b and the post-operation state of the user identified by b.
- The experiment outputs 1 if
  1. $st_0 = st_1$,
  2. queries i and j were processed as user 0 and resulted in user states st'_0 and st^*, respectively,
  3. queries k and l were processed as user 1 and resulted in user states st'_1 and st^*, respectively,
  4. $st'_0 \neq st'_1$, and
  5. $\mathsf{Version}(st'_0) = \mathsf{Version}(st'_1) < \mathsf{Version}(st^*)$;

  otherwise the experiment outputs 0.

Figure 6.2: Fork consistency experiment for searchable encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ and adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

picks two operations for each of the users and the result of the experiment is determined relative to the states that the two honest users held after the chosen operations: at some point, the users states must be distinct, but for the same version, and at a later point, the states must be identical once more.

Our definition is inspired by the *original* fork consistency experiment for append-only authenticated dictionaries (AADs) by Tomescu et al. [Tom+19], see also Figure 3.23 and Definition 25. However, in the case of AADs, a user state is considered reachable from another state if there is an append-only proof relating the two states, and in the fork consistency experiment for AADs, the adversary outputs the proofs and the users' distinct states. Since searchable encryption with access control does not feature append-only proofs, we need to ensure reachability of states in other ways. Additionally, we explicitly want the users' distinct states (st'_0 and st'_1 in our experiment) to be reachable from their identical initial states. These requirements complicate our definition of fork consistency of searchable encryption schemes with access control when compared to the fork consistency definition for AADs.

Definition 29. *A searchable encryption scheme with access control* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *is called* fork consistent*, if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

$$\Pr[\mathbf{Exp}^{\textit{SE-fc}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random bits of the adversary and the experiment.

Verifiability. Before, we have discussed the possibility of malicious servers in the context of data confidentiality, considering what the malicious server can learn from participating in searchable encryption. We now discuss what guarantees we get with a malicious server in terms of search results. Verifiability means that, if the server answers search queries at all, then we can verify whether the returned result is indeed correct and complete, meaning that the result only contains documents satisfying our query, and contains all of them. If a presented search result cannot be verified, we require the Search algorithm to output $\perp$ as the search result.

We define verifiability relative to an experiment $\mathbf{Exp}^{\text{SE-vrfy}}$, as shown in Figure 6.3. The experiment plays the roles of two honest users with identical access rights, while the adversary may play the roles of additional users. The honest users played by the experiment each perform a number of operations of the adversary's choice. Among these, both users perform a search query for a keyword of the adversary's choice. Both queries have resulted in the users adopting identical post-search user states, but the users have obtained distinct results. Furthermore, both users must have verified and accepted the result, so neither result can be $\perp$.

As with fork consistency, the notion of verifiability can be related to a security notion for AADs, particularly the *original* notion of membership security by Tomescu et al. [Tom+19], see also Figure 3.21 and Definition 25. The adaptions we have made to the original membership security experiment are similar to those made to the original fork consistency experiment (see above).

Definition 30. *A searchable encryption scheme* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *with access control is called* verifiable*, if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$

$$\Pr[\mathbf{Exp}^{\textit{SE-vrfy}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the random bits of the adversary and the experiment.

Forward privacy. Forward privacy means that the server does not learn whether any newly added document contains a specific keyword, even if that keyword has been searched for before. Intuitively, forward privacy can only be guaranteed if the server does not collude with users, because otherwise, the server could use a corrupt user's secret key to extract information from messages exchanged during an Update operation. Forward privacy may also only be guaranteed if no fork has occurred, because the server states for different forks may reveal information on keywords that the forks have in common, and this may not be preventable. These intuitions can be used to formalize forward privacy. The following definition has been adapted from Bost [Bos16] to the multi-user setting.

Definition 31. *A searchable encryption scheme* $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ *with access control is* forward private *if, in the absence of dishonest users and forks, its* Update*-specific leakage function* $\mathcal{L}_{\mathsf{Update}}$ *contains no information on keywords beyond an upper bound on the number of keywords affected by the* Update *operation.*

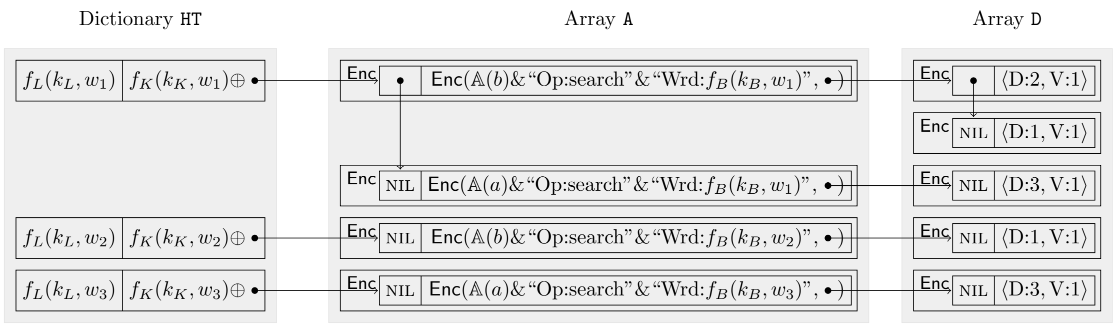
$\mathbf{Exp}^{\text{SE-vrfy}}_{\mathcal{A},\Pi}(\lambda)$

- $(pp, msk, jvk, st_{\text{judge}}) \leftarrow \mathsf{Setup}(1^\lambda)$
- $(U, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\text{UserJoin}}()}(pp)$, where the experiment reacts to queries to oracle $\mathcal{O}_{\text{UserJoin}}$ by playing the key issuer's part of protocol $\mathsf{UserJoin}$ on input (pp, msk)
- $(usk_0, st_0 | crt_0) \leftarrow \mathsf{UserJoin}(pp, U | pp, msk)$
- $(usk_1, st_1 | crt_1) \leftarrow \mathsf{UserJoin}(pp, U | pp, msk)$
- $(i, j) \leftarrow \mathcal{A}_2^{\mathcal{O}_{\text{UserJoin}}(), \mathcal{O}_{\text{Init}}(\cdot), \mathcal{O}_{\text{Update}}(\cdot,\cdot), \mathcal{O}_{\text{Search}}(\cdot,\cdot), \mathcal{O}_{\text{Resolve}}()}(st_{\mathcal{A}}, crt_0, crt_1)$, where the experiments reacts to oracle queries as follows:
  - $\mathcal{O}_{\text{UserJoin}}$ run the key issuer's part of protocol $\mathsf{UserJoin}$ on input (pp, msk)
  - $\mathcal{O}_{\text{Init}}$ on input b: if $b \notin \{0,1\}$, output $\perp$; otherwise run protocol Init playing user b; $\mathcal{A}$ plays the server's part
  - $\mathcal{O}_{\text{Update}}$ on input b and DC: if $b \notin \{0,1\}$ or DC is not a document collection, output $\perp$; otherwise run protocol Update playing the user holding certificate crt_b using DC as the user's input; $\mathcal{A}$ plays the server's part
  - $\mathcal{O}_{\text{Search}}$ on input b and kw: if $b \notin \{0,1\}$ or kw is not a keyword, output $\perp$; otherwise run protocol Search playing the user holding certificate crt_b; $\mathcal{A}$ plays the server's part.
  - $\mathcal{O}_{\text{Resolve}}$ run the judges part of protocol $\mathsf{Resolve}$ on input $(pp, jsk, st_{\text{judge}})$

  For every query to $\mathcal{O}_{\text{Init}}$, $\mathcal{O}_{\text{Update}}$, and $\mathcal{O}_{\text{Search}}$, the experiment remembers the number of previous queries processed, b and the post-operation state of the user identified by b, and the user's input and output for the operation.
- The experiment outputs 1 if
  1. $st_0 = st_1$,
  2. query i was a Search operation processed as user 0 with input kw_0, and resulted in user state st'_0 and search result $X_0 \neq \perp$,
  3. query j was a Search operation processed as user 1 with input kw_1, and resulted in user state st'_1 and search result $X_1 \neq \perp$,
  4. $st'_0 = st'_1$,
  5. $kw_0 = kw_1$, and
  6. $X_0 \neq X_1$;

  otherwise the experiment outputs 0.

Figure 6.3: Verifiability experiment for searchable encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{UserJoin}, \mathsf{Init}, \mathsf{Update}, \mathsf{Search}, \mathsf{Resolve}, \mathsf{Version})$ and adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$



Dictionary HT
Array A
Array D
$f_L(k_L, w_1)$
$f_K(k_K, w_1)\oplus$
$f_L(k_L, w_2)$
$f_K(k_K, w_2)\oplus$
$f_L(k_L, w_3)$
$f_K(k_K, w_3)\oplus$
Enc
NIL
Enc($\mathbb{A}(b)$&"Op:search"&"Wrd:$f_B(k_B, w_1)$", •)
Enc($\mathbb{A}(a)$&"Op:search"&"Wrd:$f_B(k_B, w_1)$", •)
Enc($\mathbb{A}(b)$&"Op:search"&"Wrd:$f_B(k_B, w_2)$", •)
Enc($\mathbb{A}(a)$&"Op:search"&"Wrd:$f_B(k_B, w_3)$", •)
⟨D:2, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩

Figure 6.4: The index structure used in the SEAC scheme, computed from our example document collection from Page 38. Data structures HT and D are as in the CGKO scheme, although lists in array D are potentially shorter in the SEAC scheme. Lists in array A control access to D lists and employ attribute-based encryption for this purpose. Policies used for attribute-based encryption depend on the policy shared by documents in the D list being pointed to, as well as the keyword that is shared by all documents in all D lists that the A list points to.

6.2 Static Searchable Encryption

Our first construction of a searchable encryption scheme with access control, SEAC, targets the static case of searchable encryption. As a consequence, SEAC's Update protocol is executed at most once. SEAC is an adaption of the CGKO scheme [Cur+11], c.f. Section 5.4, which results in SEAC only satisfying the security notions of data confidentiality, and (trivially) forward privacy and fork consistency, but not the notion of verifiability.

Since SEAC is a static scheme, its practical applications are rather limited. One potential use may be for data archives, which do not get updated after the initial data has been processed. However, SEAC finds applications in the construction of dynamic searchable encryption, as demonstrated in Section 6.3. Note that, since SEAC is static, it does not consider documents' write and ownership access policies, but only documents' read policies.

We now turn to discussing the ideas behind SEAC, and later on shift towards technical details and proofs of security.

6.2.1 SEAC, Conceptually

As stated, the SEAC scheme is an adaption of the CGKO searchable symmetric encryption scheme. By enhancing the index structure of Curtmola et al.'s scheme [Cur+11], the CGKO scheme is adapted into the multi-user setting with access control. Particularly, the encrypted linked list employed by Curtmola et al. to represent pre-computed search results for each keyword are split into sub-lists. Documents in the same sub-list share an access policy. The sub-lists for each keyword are still grouped together in an encrypted linked list. Hence, SEAC uses a two level hierarchy of linked lists. The upper level (A lists), which is reached from a dictionary HT, as in the CGKO scheme, lists all access policies associated with a keyword. A list entry in the upper level for a given access policy points to a list on the lower level of the hierarchy (D lists) that contains all documents with the given access policy that contain the relevant keyword. The data structures are shown in Figure 6.4.

With the given index structure, a search query contains two pseudorandom strings generated from the searched keyword, just like search queries are formed in the CGKO scheme. Additionally, in SEAC a search query contains parts of a user's decryption key for attribute-based encryption (ABE). That key is used for decrypting ABE ciphertexts in A lists. Only parts of the user's key are given to the server in order to prevent it from decrypting document ciphertexts, which are ABE encrypted for access control enforcement. Document ciphertexts are stored in data structures separate from the index structure.

We introduce namespaces for attributes in order to separate attributes by the roles they have in SEAC. Namespaces simply prefix the attribute names. We have three namespaces. Namespace Usr for general purpose attributes of users, for example, attributes describing users by their roles. Attributes from namespace Op describe operations users are allowed to perform; in SEAC we use attribute "Op:search" for policies in A list nodes and "Op:read" for policies of documents. Namespace Wrd prefixes attribute names that are derived from searchable keywords; they are used for policies in A list nodes.

Attributes from namespace Usr serve access control purposes, preventing users from accessing documents they are not allowed to access. Attributes from namespaces Op and Wrd prevent the server from accessing any document plaintexts — despite getting attributes as part of search queries — and from using these attributes for accessing search results different from the pre-computed search result for the keyword that the query asks for. The former property is achieved by not including "Op:read" in search queries, but requiring it in the policies for documents. The latter property is achieved by keyword-specific attributes in conjunction with ABE's collusion resistance, which prevents the server from combining the (general purpose) attribute from one query with the keyword-specific attribute from another query. The restriction of users' ABE keys to certain attributes is done via key delegation and the reason why, in Section 3.2.3, ABE schemes are required to feature key delegation.

6.2.2 SEAC, Formally

We now formally present the SEAC scheme, implementing the ideas laid out above. As building blocks, SEAC uses pseudorandom functions (PRFs) (Definition 3), symmetric encryption (Defintion 5; implicitly: PRF output $\oplus$ plaintext, and explicitly: scheme Sym), and ABE (Definition 12). SEAC uses PRFs and symmetric encryption similarly to the CGKO scheme, c.f. Section 5.4, for labels and values used in dictionary HT and for encryption of A and D lists. An additional PRF is used to derive keyword dependent attributes for namespace Wrd from plaintext keywords.

Construction 32. *The* SEAC *scheme consists of algorithms and protocols* (Setup, UserJoin, Init, Update, Search, Resolve, Version) *shown in Figures 6.5–6.7.*

Essentially, what the algorithms and protocols of the SEAC scheme do is the following. Setup computes the PRF keys and sets up an ABE scheme. Private keys form SEAC's master secret; the ABE scheme's parameters form SEAC's public parameters.

UserJoin has the user send her desired (general purpose) attributes to the key issuer. The key issuer assigns a random certificate (number) to the user and then computes an ABE key for the user, giving her keys for the requested attributes (in namespace Usr), and additionally giving the user keys for attributes "Op:search" and "Op:read". The user also gets attribute keys for all keywords $kw \in words$; the set *words* is assumed to be public knowledge. The user's key is composed of the computed ABE key and of the PRF keys sampled during Setup. The key is given to the user, who also defines her state.

Init effectively does nothing beyond incrementing a counter in the key issuer's state. The protocol is only present for SEAC to satisfy the definition of a searchable encryption scheme with access control.

```
Setup($1^\lambda$)
• $k_B \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_L \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $(pp, msk') \leftarrow \mathsf{ABE.Setup}(1^\lambda)$
• $msk \leftarrow (msk', k_B, k_K, k_L)$
• $jsk \leftarrow \perp$
• $st_{\text{judge}} \leftarrow \perp$
• $st_{\text{issuer}} \leftarrow 0$
• output $(pp, msk, jvk, st_{\text{judge}}, st_{\text{issuer}})$

UserJoin($pp, U | pp, msk, st_{\text{issuer}}$)
U: send $U$ to key issuer
KI: if $st_{\text{issuer}} < 2$: send “no document collection” to user and abort protocol
KI: $usk' \leftarrow \mathsf{ABE.KeyGen}(msk', \{\text{“Usr:}u\text{”}\}_{u \in U} \cup \{\text{“Op:read”}, \text{“Op:search”}\} \cup \{\text{“Wrd:}\mathsf{PRF.E}$
KI: $crt \leftarrow_{\$} \{0,1\}^\lambda$
KI: $usk \leftarrow (usk', k_B, k_K, k_L, crt)$
KI: publish $crt$
KI: send $usk$ to user
U: if received “no document collection:” abort protocol
U: $st_{\text{user}} \leftarrow -1$
U: output $(usk, st_{\text{user}})$ to user

Init($pp, msk, st_{\text{issuer}} | pp, st_{\text{server}}$)
KI: if $st_{\text{issuer}} > 0$: abort protocol
KI: send “initialize” to server
S: if $st_{\text{server}} \neq \perp$: send “already initialized” to key issuer and abort protocol
S: $st_{\text{server}} \leftarrow \emptyset$
S: send “initialized” to key issuer
S: output $st_{\text{server}}$
KI: if received message “already initialized:” abort protocol
KI: $st_{\text{issuer}} \leftarrow 1$
KI: output $st_{\text{issuer}}$ to key issuer
```

Figure 6.5: Algorithm Setup, and protocols UserJoin and Init of the SEAC scheme

```
$\underline{\mathsf{Update}(pp,msk,st_{\text{issuer}},DC|pp,st_{\text{server}})}$
KI: if $st_{\text{issuer}}>1$: abort protocol
KI: $K,R\leftarrow\emptyset$
KI: $\texttt{CT},\texttt{HT}\leftarrow$ empty dictionaries
KI: for each $doc=(d,v,r,w,o,k)\in DC$:
  • $R\leftarrow R\cup\{r\}$
  • for each $kw\in k$:
     – $K\leftarrow K\cup\{kw\}$
     – if $L[kw,r]$ is undefined: $L[kw,r]\leftarrow\emptyset$; else: $L[kw,r]\leftarrow L[kw,r]\cup\{(d,v)\}$
  • $\texttt{CT}.\mathsf{Append}(\langle d,v\rangle,\mathsf{ABE.Enc}(pp,r\&\text{“Op:read”},k))$
KI: $s\leftarrow\sum_{(d,v,r,w,o,k)\in DC}\sum_{kw\in k}|kw|/\min\{|kw|:(d,v,,r,w,o,k)\in DC\wedge kw\in k\}$
KI: $\texttt{A}\leftarrow$ empty memory array with $|R|\cdot|K|$ cells marked free
KI: $\texttt{D}\leftarrow$ empty memory array with $s$ cells marked free
KI: for each $(kw,r)\in K\times R$: // compute D lists
  • if $L[kw,r]$ is defined:
     – $sp\leftarrow$ NIL
     – while $L[kw,r]\neq\emptyset$:
          ∗ pick $(d,v)$ uniformly at random from $L[kw,r]$
          ∗ $L[kw,r]\leftarrow L[kw,r]\setminus\{(d,v)\}$
          ∗ pick index $i$ uniformly at random from the indices of cells of D marked free
          ∗ remove free mark from cell $i$ of D
          ∗ $nk\leftarrow\mathsf{Sym.KeyGen}(1^\lambda)$
          ∗ $\texttt{D}[i]\leftarrow\mathsf{Sym.Enc}(nk,\langle sp,\langle d,v\rangle\rangle)$ // apply padding
          ∗ $sp\leftarrow\langle i,nk\rangle$
     – if $L[kw]$ is undefined: $L[kw]\leftarrow\emptyset$; else: $L[kw]\leftarrow L[kw]\cup\{(r,sp)\}$
KI: for each $kw\in K$: // compute A lists
  • $sp\leftarrow$ NIL
  • while $L[kw]\neq\emptyset$:
     – pick $(r,lp)$ uniformly at random from $L[kw]$
     – $L[kw]\leftarrow L[kw]\setminus\{(r,lp)\}$
     – pick index $i$ uniformly at random from the indices of cells of A marked free
     – remove free mark from cell $i$ of A
     – $nk\leftarrow\mathsf{Sym.KeyGen}(1^\lambda)$
     – $inner\leftarrow\mathsf{ABE.Enc}(pp,r\&\text{“Op:search”}\&\text{“Wrd:}\mathsf{PRF.Eval}(k_B,kw)\text{”},lp)$
     – $\texttt{A}[i]\leftarrow\mathsf{Sym.Enc}(nk,\langle sp,inner\rangle)$ // apply padding
     – $sp\leftarrow\langle i,nk\rangle$
  • $label\leftarrow\mathsf{PRF.Eval}(k_L,kw)$
  • $pointer\leftarrow\mathsf{PRF.Eval}(k_K,kw)\oplus sp$ // apply padding
  • $\texttt{HT}.\mathsf{Append}(label,pointer)$
KI: for each cell $i$ of A marked free: $\texttt{A}[i]\leftarrow\mathsf{Sym.Enc}(\mathsf{Sym.KeyGen}(1^\lambda),0)$ // apply padding
KI: for each cell $i$ of D marked free: $\texttt{D}[i]\leftarrow\mathsf{Sym.Enc}(\mathsf{Sym.KeyGen}(1^\lambda),0)$ // apply padding
KI: send (HT,A,D,CT) to server
S: if $st_{\text{server}}=\perp$ or $st_{\text{server}}\neq\emptyset$: send “operation disallowed” to key issuer and abort protocol
S: send “data stored” to key issuer
S: output $st_{\text{server}}\leftarrow(\texttt{HT},\texttt{A},\texttt{D},\texttt{CT})$ to server
KI: if received message “operation disallowed:” abort protocol
KI: output $st_{\text{issuer}}\leftarrow 2$ to key issuer
```

Figure 6.6: Protocol Update of the SEAC scheme

```
Search($pp$,$usk$,$st_{\text{user}}$,$kw$|$pp$,$st_{\text{server}}$)
U: $HTlabel \leftarrow \mathsf{PRF.Eval}(k_L, kw)$
U: $HTkey \leftarrow \mathsf{PRF.Eval}(k_K, kw)$
U: let $usk_{\text{restricted}}$ be the restriction of $usk$ to attributes from namespace Usr, and
   additionally the attributes "Op:search" and "Wrd:$\mathsf{PRF.Eval}(k_B, kw)$", computed via
   ABE.Delegate
U: send $query = (HTlabel, HTkey, usk_{restricted})$ to server
S: if $st_{\text{server}} = \perp$: send "not initialized" to user and abort protocol
S: if $st_{\text{server}} = \emptyset$: send "no data stored" to user and abort protocol
S: if $\mathtt{HT}.\mathsf{Lookup}(HTlabel)$ is undefined: send $\emptyset$ to user and abort protocol
S: $rslt \leftarrow \emptyset$
S: $\langle sp, lk \rangle \leftarrow \mathtt{HT}.\mathsf{Lookup}(HTLabel) \oplus HTkey$
S: while $p = \langle sp, lk \rangle \neq$ NIL:
   • $(p, inner) \leftarrow \mathsf{Sym.Dec}(lk, \mathtt{A}[sp])$
   • $lp \leftarrow \mathsf{ABE.Dec}(pp, usk_{\text{restricted}}, inner)$
   • if $lp = \perp$: continue to next iteration
   • while $lp = \langle np, nk \rangle \neq$ NIL:
       – $(lp, \langle d, v \rangle) \leftarrow \mathsf{Sym.Dec}(nk, \mathtt{D}[np])$
       – $rslt \leftarrow rslt \cup \{\mathtt{CT}.\mathsf{Lookup}(\langle d, v \rangle)\}$
S: send $rslt$ to user
U: if received message "not initialized" or "no data stored:" abort protocol
U: $st_{\text{user}} \leftarrow 0$
U: $X \leftarrow \emptyset$
U: for each $ct \in rslt$:
   • $X \leftarrow X \cup \{\mathsf{ABE.Dec}(pp, usk, ct)\}$
U: output $X$ and $st_{\text{user}}$

Resolve($pp$,$st_{\text{server}}$|$pp$,$jsk$,$st_{\text{judge}}$)
S&J: abort protocol

Version($pp$,$st_{\text{user}}$)
• output $st_{\text{user}}$
```

Figure 6.7: Protocols Search and Resolve, and algorithm Version of the SEAC scheme

Update computes document ciphertexts from all documents in the document collection; remember that the multi-set k from a document's tuple in a document collection represents the plaintext document as a multi-set of keywords contained in the document, c.f. explanation on document collections on page 38. The user proceeds to compute the index structure, ensuring that the order of documents in pre-computed search results is independent from their order in the document collection. As in the CGKO scheme, the user ensures that all symmetric ciphertexts in HT, A, and D, respectively, have the same length, and that arrays A and D contain dummy entries. Then, the user sends her computed data to the server, which stores the data as its state.

For Search, the user computes a subset of her key and sends the subset as her query. Given the partial key, the server recovers some pre-computed search results, which identify relevant document ciphertexts. The identified ciphertexts are sent to the user, who decrypts them and outputs the resulting plaintexts.

Resolve and Version effectively do nothing. The protocol and algorithm are present for SEAC to satisfy the definition of a searchable encryption scheme with access control.

Correctness. From the intuition of protocols Update and Search, we can immediately see, that SEAC is correct in terms of Conditions 2–4 of the result correctness notion of searchable encryption with access control: the Update protocol computes an index structure such that the Search protocol results in the user outputting the set of documents accessible to the user that contain the searched keyword. Conditions 1 and 5 are trivially satisfied, because the Search protocol never outputs $\perp$ to the user and because SEAC does not make user of notifications. The additional conditions due to state correctness are also satisfied.

Efficiency. A drawback in SEAC is its sensitivity towards the number of keyword–policy pairs in its document collection and the influence of that sensitivity on the efficiency of SEAC. The sensitivity can immediately be seen from the size of memory array A, whose size is an upper bound on the number of keyword–policy pairs. The number of keyword–policy pairs also influences the number of ABE encryptions that need to be performed when generating the index structure, particularly then computing memory array A. Similarly, during search, for each policy the searched keyword is associated with an ABE decryption, or at least a test whether decryption is possible, needs to take place. All in all, the efficiency of operations not only depends on the size of a document collection, but also on its inner structure.

Leakage. In order for us to prove the security of the SEAC scheme, we need to analyze the scheme's leakage to an adversary that controls the server and dishonest users. To that end, we now discuss the leakage functions involved in SEAC. This analysis assumes PRF to be a pseudorandom function, Sym to be an eavesdropping-secure symmetric encryption scheme, and ABE to be a chosen plaintext attack (CPA)-secure ABE scheme. Note that every keyword is associated with an identifier allowing us to distinguish keywords, hat are otherwise indistinguishable in terms of leakage functions, for example, because they occur in the same documents. Identifiers of keywords come in the form of keyword dependent keys, labels, and attributes. From a keyword, we can always infer the respective identifier, but the reverse is not necessarily true.

The first leakage function to discuss is $\mathcal{L}_{\text{HonestJoin}}$, which is the leakage from enrolling honest users in the system. In this scenario, the adversary only gets to observe the public outputs of the UserJoin protocol. This means, the adversary only sees the user certificate *crt* output by the key issuer. Hence, $\mathcal{L}_{\text{HonestJoin}}(U) = crt$.

The adversary gains significantly more knowledge on the plaintext document collection DC when enrolling dishonest users. The additional knowledge is due to the user keys that the adversary obtains from the process and the things that the server can do using these keys, for instance, decrypt document ciphertexts. The server learns the user's certificate crt and attributes U. From data structure $\mathtt{CT}$, the server learns the document names and version numbers, the read access policies, and the contents of documents that the user is allowed to access due to her attributes. We denote this as

$$DC(U) = \{(d, v, \mathbb{A}_{\text{read}}, k) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC \wedge U \text{ satisfies } \mathbb{A}_{\text{read}}\}.$$

Due to the adversary's (implicitly assumed) knowledge of all keywords in the document collection and the adversary's knowledge of a user's key, the adversary can compute $\mathtt{HT}$ labels and keys for all keywords, granting the adversary access to all $\mathtt{A}$ lists. From the $\mathtt{A}$ lists, the adversary learns all keyword–access policy lists in the document collection DC. We denote this as

$$KP(DC) = \{(kw, \mathbb{A}_{\text{read}}) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC \wedge kw, \in k\}.$$

Therefore,

$$\mathcal{L}_{\text{CorruptJoin}}() = (crt, U, DC(U), KP(DC)).$$

When corrupting a once honest user, the adversary obtains the same information as if the user were enrolled as a dishonest user; except for the user's certificate crt, which the adversary has already learned before. Hence,

$$\mathcal{L}_{\text{Corruption}}(crt) = (crt, U, DC(U), KP(DC)).$$

No information can be learned from the server initialization, so $\mathcal{L}_{\text{Init}}(crt) = \perp$

Since SEAC is static, users cannot perform updates. Instead, the single update is performed by the key issuer. Users do not even exist at the time the update operation is performed. The update operation therefore leaks nothing beyond the sizes of data structures $\mathtt{HT}$, $\mathtt{A}$ and $\mathtt{D}$, and some information on document ciphertexts. The size of $\mathtt{HT}$ is determined by the number of distinct keywords in the document collection, so

$$|\mathtt{HT}| = |\{kw : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC \wedge kw \in k\}|.$$

The size of arrays $\mathtt{A}$ and $\mathtt{D}$ is explicitly computed in algorithm Update as an upper bound on the number of keyword–access policy pairs in the document collection and as the quotient of the total size of document plaintext and the length of the shortest keyword, which is a trivial upper bound on the average number of keywords per document from document collection DC, so

$$|\mathtt{A}| = |\{(kw, \mathbb{A}_{\text{read}}) : (d, v, r, w, o, k) \in DC \wedge kw \in k \wedge (d', v', \mathbb{A}_{\text{read}}, o', k') \in DC\}|$$

and

$$|\mathtt{D}| = \frac{\sum_{(d,v,r,w,o,k) \in DC} \sum_{kw \in k} |kw|}{\min\{|kw| : (d, v, r, w, o, k) \in DC \wedge kw \in k\}}.$$

Finally, from data structure $\mathtt{CT}$ that stores document ciphertexts, the adversary leans the set $\mathtt{CT}(DC)$ of document names, version numbers, read access policies, and sizes of documents in DC, so

$$\mathtt{CT}(DC) = \{(d, v, \mathbb{A}_{\text{read}}, |k|) : (d, v, \mathbb{A}_{\text{read}}, o, w, k) \in DC\}.$$

All in all,

$$\mathcal{L}_{\text{Update}}(DC) = (|\mathtt{HT}|, |\mathtt{A}|, |\mathtt{D}|, \mathtt{CT}(DC)).$$

Whenever an honest user with user certificate *crt* issues a search query for any keyword *kw*, the adversary learns the user's attribute set $U(crt)$, and identifiers of *kw*, denoted *id*(*kw*), particularly, keyword dependent keys and labels from the query. The adversary also learns the read policies that occur in conjunction with the keyword *kw* in the searchable document collection DC. This is learned from the cells of array `A` that can be decrypted using the decryption key from the query. We denote set of read policies as

$$\mathbb{A}_{\text{read}}(DC, kw) = \{\mathbb{A}_{\text{read}} : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC \land kw \in k\}.$$

From `D` lists accesses during search, the adversary learns the document names and version numbers of documents that both, contain the keyword *kw* and are accessible to the user. We denote this as

$$doc(U(crt), kw) = \{(d, v) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC \land kw \in k \land U(crt) \text{ satisfies } \mathbb{A}_{\text{read}}\}.$$

Therefore,

$$\mathcal{L}_{\text{Search}}(crt, kw) = (U(crt), id(kw), \mathbb{A}_{\text{read}}(DC, kw), doc(U(crt), kw)).$$

Since protocol Resolve does nothing, nothing can be learned from executing it, so $\mathcal{L}_{\text{Resolve}}() = \perp$.

Security. We now come to the analysis of SEAC's security properties. As stated before, SEAC provides data confidentiality, and, trivially, forward privacy and fork consistency. We restate these claims as Lemmas 33–35 and proof them individually.

Lemma 33. *If* PRF *is modelled as a random oracle,* Sym *is an eavesdropping-secure symmetric encryption scheme, and* ABE *is a* CPA*-secure non-committing (hybrid)* ABE *scheme, then* SEAC *provides data confidentiality (Definition 28) relative to the above leakage functions* ($\mathcal{L}_{HonestJoin}, \mathcal{L}_{CorruptJoin}, \mathcal{L}_{Corruption}, \mathcal{L}_{Init}, \mathcal{L}_{Update}, \mathcal{L}_{Search}, \mathcal{L}_{Resolve}$) *in the random oracle model.*

Proof. Given the lemma's prerequisites, as argued above, SEAC's leakage functions take the described form. For the proof, we use the fact that ABE is non-committing. Hence, random oracles solve the challenge of performing the Update operation without actually knowing the underlying document collection. At a later time, when (parts of) the underlying document collection become known, for instance, due to the adversary corrupting a user or from search results, the simulator patches a random oracle, and the data stored at the server becomes dependent on the revealed parts of the document collection.

In the random oracle model, we not only have non-committing ABE, but also non-committing symmetric encryption. Our proof also uses random oracles to model the PRFs of the SEAC scheme. As implied by these statements, we make heavy use of random oracles in our proof and, particularly, the definition of the simulator for experiment $\mathbf{Exp}^{\text{SE-conf-sim}}$. The simulator's behavior is defined in Figures 6.8 and 6.9.

Due to the simulator's ability to patch the random oracles, all data structures in $\mathbf{Exp}^{\text{SE-conf-sim}}_{\mathcal{A},\mathcal{S},\mathsf{SEAC}}$ behave just like their counterparts in $\mathbf{Exp}^{\text{SE-conf-real}}_{\mathcal{A},\mathsf{SEAC}}$. This is the conclusion we draw from two main observations. First, the adversary $\mathcal{A}$ only can make meaningful queries to the random oracles modelling the PRFs after a user has been corrupted. This is because the adversary has no knowledge of the PRF keys before it has corrupted any user.

$\mathcal{O}_{\text{Setup}}(1^\lambda)$

- $(pp, msk, jsk, st_{\text{issuer}}) \leftarrow \mathsf{Setup}(1^\lambda)$
- remember msk, st_{issuer}
- output pp

$\mathcal{O}_{\text{HonestJoin}}(\mathcal{L}_{\text{HonestJoin}}(U))$

- execute protocol $\mathsf{UserJoin}(pp, \emptyset | pp, msk, st_{\text{issuer}})$; remember the published user certificate *crt*

$\mathcal{O}_{\text{CorruptJoin}}(\mathcal{L}_{\text{CorruptJoin}}())$

- if *crt* from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following
- based on received leakage $\{(d, v, \mathbb{A}_{\text{read}}, k)\}$ that has not occurred before:
  - patch random oracle for non-committing ABE, so the `CT` entry for document identifier and version (d, v) with policy $\mathbb{A}_{\text{read}}$ contains plaintext document k
  - compute non-encrypted linked `D` lists as in protocol Update; patch the random oracle for non-committing symmetric encryption in order to embed the lists in array `D`
- based on received leakage $\{(kw, \mathbb{A}_{\text{read}})\}$ that has not occurred before:
  - compute non-encrypted linked `A` lists as in protocol Update, except the ABE ciphertexts inside the list nodes are dummy ciphertexts, unless a `D` list for $(kw, \mathbb{A}_{\text{read}})$ has previously been computed, in which case the ABE ciphertext is an encryption of the pointer to the `D` list
  - patch random oracles for PRFs so the `HT` entry for keyword *kw* points to `A` list for *kw*
- if any of the above oracle patching fails: output $\perp$ and abort the simulation
- execute $\mathsf{UserJoin}$ with key issuer's input $(pp, msk, st_{\text{issuer}})$; interact with $\mathcal{A}$, which plays the user; remember the published user certificate *crt* and mark it as corrupt

$\mathcal{O}_{\text{Corruption}}(\mathcal{L}_{\text{Corruption}}(crt))$

- if *crt* from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following
- compute data and patch oracles as in $\mathcal{O}_{\text{CorruptJoin}}$
- if no user key *usk* and state st_{user} for *crt* have been remembered:
  - based on received leakage U: execute $(usk, st_{\text{user}}) \leftarrow \mathsf{UserJoin}(pp, U | pp, msk, st_{\text{issuer}})$ playing the user and key issuer, but do not publish resulting certificate *crt*
  - remember *usk* as *crt*'s user key and st_{user} as *crt*'s state
- mark *crt* as corrupt
- output (usk, st_{user}) to $\mathcal{A}$

$\mathcal{O}_{\text{Init}}(\mathcal{L}_{\text{Init}}(crt))$

- execute protocol Init with key issuer's input $(pp, msk, st_{\text{issuer}})$; interact with $\mathcal{A}$; remember updated key issuer's state

Figure 6.8: Behavior of the simulator $\mathcal{S}$ for SEAC's data confidentiality game against adversary $\mathcal{A}$: Reactions to setup, enrollment, corruption, and initialization leakage

$\mathcal{O}_{\text{Update}}(\mathcal{L}_{\text{Update}}(crt,DC))$

- if $st_{\text{issuer}} \neq 1$: do nothing of the following
- based on received leakage on set sizes:
  - compute dictionary `HT` containing dummy entries (labels and values chosen uniformly at random from appropriate domains); the number of entries is indicated by the leakage
  - compute memory arrays `A` and `D` containing dummy cells; the number of entries is indicated by the leakage
- based on received leakage on `CT`: compute dummy `CT` entries for document identifiers and version numbers with access policies and plaintext sizes as indicated by the leakage
- if any of the above oracle patching fails: output $\perp$ and abort the simulation
- send (`HT`,`A`,`D`,`CT`) to $\mathcal{A}$ and behave like the key issuer in protocol Update in the ensuing interaction with the server (played by $\mathcal{A}$); remember the updated key issuer's state

$\mathcal{O}_{\text{Search}}(\mathcal{L}_{\text{Search}}(crt,kw))$

- if *crt* from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following
- if the identifier of *kw* has been observed in previous search leakage: skip to the last point
- else: based on received leakage $\{(d,v)\}$ that has not occurred before:
  - compute non-encrypted linked `D` lists as in protocol Update; patch the random oracle for non-committing symmetric encryption in order to embed the lists in array `D`
- based on received leakage $\{\mathbb{A}_{\text{read}}\}$ that has not occurred before:
  - compute non-encrypted linked `A` list as in protocol Update, except ABE ciphertexts inside the list nodes are dummy ciphertexts, unless a `D` list for $(kw,\mathbb{A}_{\text{read}})$ has previously been computed, in which case the ABE ciphertext is an encryption of the pointer to the `D` list
  - patch random oracles for PRFs so the `HT` entry for keyword *kw* points to the `A` list for *kw*
- if any of the above oracle patching fails: output $\perp$ and abort the simulation
- if no user key *usk* and state st_{user} for *crt* have been remembered:
  - based on received leakage U: execute $(usk,st_{\text{user}}) \leftarrow \mathsf{UserJoin}(pp,U|pp,msk,st_{\text{issuer}})$ playing the user and key issuer, but do not publish the resulting certificate *crt*
  - remember *usk* as *crt*'s user key and st_{user} as *crt*'s state
- behave like user *crt* in protocol Search and interact with the server (played by $\mathcal{A}$); use PRF images to construct the query that recover lists given in the leakage (the lists have been created before)

$\mathcal{O}_{\text{Resolve}}(\mathcal{L}_{\text{Resolve}}())$

- do nothing

Figure 6.9: Behavior of the simulator $\mathcal{S}$ for SEAC's data confidentiality game against adversary $\mathcal{A}$: Reactions to update, search, and resolution leakage

Once a user gets corrupted, the incurred leakage ensures that the simulated data structures are as they are in experiment $\mathbf{Exp}^{\text{SE-conf-real}}_{\mathcal{A},\mathsf{SEAC}}$, at least as far as those parts of the data structures `HT`, `A`, `D` and `CT` are concerned that the adversary can decrypt. Those parts of the data structure that the adversary is unable to decrypt, due to our use of secure encryption and the adversary's lack of decryption keys, are covered by our second observation: Since we use eavesdropping secure symmetric encryption with a fresh key for each ciphertext and CPA-secure ABE, ciphertexts on plaintexts of equal length are indistinguishable from each other. Particularly, dummy ciphertexts in `A`, `D`, and `CT` cannot be distinguished from not yet decrypted lists nodes and documents.

As a consequence of the data structures behaving identically in both experiments, the adversary cannot use the data structures to distinguish between the experiments. Our simulator is designed so that its output behavior in experiment $\mathbf{Exp}^{\text{SE-conf-sim}}$ is exactly the same as the oracles' output behavior in experiment $\mathbf{Exp}^{\text{SE-conf-real}}$ with the notable exception that the simulator outputs $\perp$ if it fails to patch one of its oracles. However, patching the oracle only fails with probability negligible in the security parameter. $\square$

Lemma 34. *The* SEAC *scheme is fork consistent (Definition 29).*

Proof. In order for an adversary to break SEAC's fork consistency, an adversary needs to make two honest users with identical initial state st_0 transition to state st^*. In the process, one user adopts state st'_0 and the other adopts state st'_1. These states must satisfy $\mathsf{Version}(pp, st_0) < \mathsf{Version}(pp, st'_0) = \mathsf{Version}(pp, st'_1) < \mathsf{Version}(pp, st^*)$. However, in SEAC, honest users' states make Version output either -1 or 0, so the above conditions can never be satisfied. $\square$

Lemma 35. *If* PRF *is a pseudorandom function,* Sym *is an eavesdropping-secure symmetric encryption scheme, and* ABE *is a* CPA*-secure* ABE *scheme,* SEAC *is forward private (Definition 31).*

Proof. As implied by our proof of SEAC's fork consistency, the notion of forks is irrelevant for SEAC. Furthermore, due to the order in which operations must occur in a static searchable encryption scheme with access control, c.f. Definition 27, there are no dishonest users at the time SEAC's update leakage $\mathcal{L}_{\text{Update}}$ occurs. Therefore, in SEAC, update leakage always occurs in the absence of forks and dishonest users. This is reflected in SEAC's leakage function $\mathcal{L}_{\text{Update}}$, which is as presented above due to the lemma's prerequisites, and which contains no information on keywords beyond an upper bound on the number of distinct keywords affected by the Update operation, and information that can trivially be derived from this upper bound. Examples of information that can trivially be derived, are an upper bound on the number of keyword–policy pairs in the document collection and an upper bound on the average number of keywords per document. The derived information is independent of the actual keywords.

The upper bound on the number of distinct keywords in the update comes in the form of the number of entries in dictionary `HT`. The bound and information trivially derived from it is the only information that the Update leakage contains on keywords if no user is corrupt. Hence, SEAC is forward private. $\square$

6.3 Dynamic Searchable Encryption

An obvious drawback of the searchable encryption scheme with access control from the previous section is that the SEAC scheme is static. In this section, we expand upon SEAC and construct dSEAC, which is dynamic.

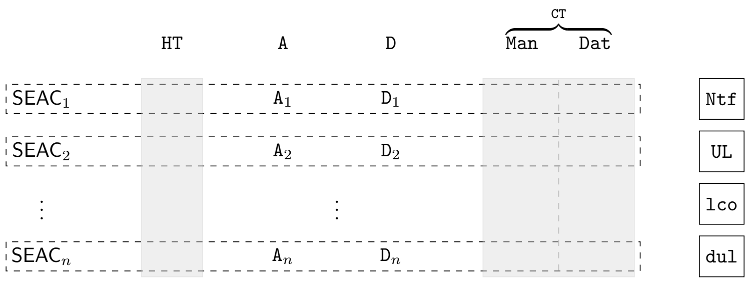


Figure 6.10: Organization of multiple (extended) SEAC instances in dSEAC: arrays A and D remain separate for each SEAC instance, but dictionary HT is shared among instances; SEAC's dictionary CT is split into two dictionaries Man and Dat that are shared between instances; additional data structures Ntf, UL, lco, and dul are stored at the server

The dynamic aspect of dSEAC is achieved by combining multiple SEAC instances, one for each update, in a specific manner, such that dSEAC achieves data confidentiality, forward privacy, fork consistency, and verifiability. At the same time, we keep users' keys and public parameters short — not much longer than in a single SEAC instance. As a consequence of the short key requirement, we cannot achieve dynamic searchable encryption from a straightforward generic combination of SEAC instances. Such a combination either does not have short keys, or is not forward private, because key reuse occurs and the server learns when two document collections have a keyword in common, because then the resulting SEAC index structures have HT labels in common. The SEAC scheme lacks the desired feature of verifiability. Additionally, in SEAC, there is only one update to the searchable document collection and it is performed by the key issuer.

In contrast, our dSEAC scheme allows all users to update the searchable document collection. Updates to the collection respect write access restrictions set by previous updates. At the same time, the dSEAC scheme is fork consistent and forward private. Search respects read access restrictions, while search results are verifiable.

As we can see, we need to make a number of changes to SEAC in order to use it in our construction of dSEAC. In the upcoming subsections, we present the necessary changes on a conceptual level, present additional measures we need to take in order make dSEAC viable, and then present and discuss the complete dSEAC scheme.

6.3.1 dSEAC, conceptually

The dSEAC scheme employs multiple SEAC instances in order to achieve dynamic searchable encryption with access control. As a consequence, the data structures employed in SEAC have counterparts in dSEAC. Particularly, the dictionary HT from SEAC becomes a dictionary shared between SEAC instances in dSEAC. The arrays A and D from SEAC remain separate. The dictionary CT, used to store document ciphertexts in SEAC, is split into two dictionaries Man and Dat, which are shared between SEAC instances. The split of CT is a consequence of dSEAC considering read *and* write access rights to data, rather than only considering read access as in SEAC. As we explain further below, Man primarily handles aspects of write access, whereas Dat primarily handles aspects of read access.

The organization of these data structures is shown in Figure 6.10. The figure also shows four additional data structures not present in SEAC, namely, data structures Ntf, UL, lco, and dul. Ntf is a collection of notifications that originate from conflict resolution, the process by which an honest server can blame a malicious user's actions on the user.

Thus, `Ntf` plays a role in dSEAC achieving correctness and verifiability. `UL` is an update log. It is an AAD that essentially serves as a catalog of all updates ever performed, and plays a role in conflict resolution. `lco` represents the last completed (write) operation. It is a digest of the system state, serves as a *discrete clock* and plays an important role in achieving verifiability. Lastly, `dul` establishes a relation between `lco` and `UL`, and plays an important role in achieving fork consistency.

However, the new data structures are insufficient for achieving verifiability and forward privacy. Achieving these notions requires modifications to internals of some of SEAC's original data structures, while retaining their conceptual purpose, c.f. Section 6.2.1. In the upcoming paragraphs, we discuss the changes to SEAC's data structures and additional data structures introduced by dSEAC. The discussion assumes users to hold a signing–verification key pair (sk, vk) of a signature scheme with delayed verifiability, c.f. Definition 22, in addition to their SEAC key. Users obtain the key pair during the UserJoin operation. In dSEAC, the certificate published during the UserJoin operation is a certificate on the user's verification key and also contains a list of the (names of) attributes the user holds. In order for the key issuer to create such certificates, the key issuer holds a singing key. The corresponding verification key is part of the system parameters pp. Similarly, in dSEAC the judge holds a signing key, and the corresponding verification key is again contained in pp.

Managing access rights. In our notion of document collections, c.f. page 38, we have three types of access rights: rights to *read* a document, rights to *write* a document, and rights to *own* a document. Write access to a document allows a user to create new versions of the document and to determine that new version's read policy. Document ownership allows a user to determine future document versions' write and ownership policies. Our SEAC construction only considers read access and ignores restrictions to write and ownership access, because these restrictions only apply to future document versions, but documents are only written once and thus have only a single version. In dSEAC, however, documents can be written multiple times, so restrictions to write access and ownership apply.

As stated before, dSEAC stores documents using two dictionaries `Man` and `Dat`, which replace SEAC's `CT` data structure. Particularly, `Man` and `Dat` jointly store documents, but differ in what part of the document they store. The distinction is due to the access rights necessary to write the respective part of the document. `Man` stores the part of documents that requires ownership access to be written, particularly, the document's ownership and read policies. `Dat` stores the part of documents that requires write access to be written, particularly , the documents' read policy and contents. We call the label–value pairs stored in `Man` and `Dat` management and data pairs, respectively.

Let (d, v, r, w, o, k) be a document from a document collection, with document identifier d, version number v, read access policy r, write access policy w, ownership policy o and keyword set/contents k. The management pair for this document stores $((d, v), (w, o))$. The corresponding data pair stores $((d, v), (r, k))$. Both pairs use (d, v) as the label for their respective dictionary entry.

Management and data pairs enforce the three types of access rights by means of ABE in combination with signatures (with delayed verifiability). This combination of encryption and signatures for enforcing (write) access restrictions is inspired by the Sharoes system [SL08]. Technically, delayed verifiability of signatures is not required here, but it is used for consistency. dSEAC implements the pairs described above as follows.

A management pair for document (d, v, r, w, o, k) consists of label (d, v) and value

$$\begin{pmatrix} v_{dp}, \mathsf{ABE.Enc}(o, dsk_{\text{owner}}), \mathsf{ABE.Enc}(w, dsk_{\text{writer}}), \\ dvk_{\text{owner}}, dvk_{\text{writer}}, crt_{\text{creator}}, \sigma_{\text{owner}}, \sigma_{\text{creator}} \end{pmatrix}.$$

Component v_{dp} establishes relations between management and data pairs for the same document; it is discussed further below. The above tuple contains two ABE ciphertexts, one each under the document's ownership and write policy. As messages, the ciphertexts contain document owner's signing key dsk_{owner} and document writer's signing key dsk_{writer}, respectively. The corresponding signature verification keys dvk_{owner} and dvk_{writer} are contained in the above tuple. The tuple also contains the certificate crt_{creator} of the user who has created the tuple. Finally, the tuple contains a signature σ_{owner} on (d, v) and tuple components v_{dp} to crt_{creator} under the document owner's signing key dsk_{owner}, and a signature σ_{creator} on (d, v) and tuple components v_{dp} to σ_{owner} under the tuple creator's signing key. The certificate crt_{creator} is a certificate on the verification key that corresponds to that signing key.

A data pair for document (d, v, r, w, o, k) consists of label (d, v) and value

$$(v_{mp}, \mathsf{ABE.Enc}(r, k), crt_{\text{creator}}, \sigma_{\text{writer}}, \sigma_{\text{creator}}).$$

Component v_{mp} is discussed further below. The tuple contains an encryption of the document's content under the read access policy, the certificate crt_{creator} of the user who has created the tuple, and two signatures σ_{writer} and σ_{creator}. Signature σ_{writer} is a signature on (d, v) and tuple components v_{mp} to crt_{creator} under the document writer's signing key from the management pair that is relevant to this data pair. Signature σ_{creator} is a signature on (d, v) and tuple components v_{mp} to σ_{writer} under the signing key of the tuple's creator.

As we can see, there is a connection between management and data pairs in the form of the document writer's signing key, which is contained in a management pair, but is used to create a signature in a data pair. The σ_{writer} signature from the data pair can be used to check that the creator of the tuple holds the necessary write access rights by verifying σ_{writer} relative to the verification key from the management pair relevant to the data pair. If the creator does not hold the necessary rights, she is unable to obtain the document writer's signing key from the management pair's ABE ciphertext. As an alternative to holding ABE keys that enable a creator to decrypt the ABE ciphertext in the management pair, the creator of a new data pair could also be the original creator of the management tuple, and thus remember the document writer's singing key. Therefore, when checking that a creator holds the necessary access rights, relying on signatures is insufficient. We additionally check that the user is able to decrypt the ABE ciphertext that contains the writer's signing key by considering the attributes held by the creator. The set of attributes held by the creator is contained in the creator's certificate crt_{creator} as contained in the new data pair.

The above mechanism enforces write access restrictions to data pairs. With slight adaptions, the mechanism can also be used to enforce write restrictions to management pairs. For that, we replace the document writer's key from the above description by the document owner's key from the management tuple. The ABE ciphertext on the owner's key together with the tuple creator's certificate, as before, ensure that the creator of a management tuple for a new document version holds the rights that allow her to create the new management tuple.

It is important to note that the write and ownership access policies for document version v are contained in the management tuple for document version $v - 1$. As a consequence, there are no write restrictions for version 1 of documents. As another consequence of write and ownership policies for version v being contained in the management pair for version $v - 1$, a user holding a document's ownership rights, but lacking the document's write rights, seems unable to create a new version of the document. Due to our mechanism for access rights enforcement, the user can create a management tuple for the new version, because her access rights grant her access to the document owner's signing key from the previous version's management tuple, but the lack of write access rights prevents the user from creating a data tuple for the new document version: Write access, and thus, access to the document writer's signing key, is required, because the data pair contains a signature under that signing key on, among other things, the document version number, which changes when compared to the data pair of the current version. A similar problem occurs when a user with write access, but without ownership access, tries to create a new version.

Tuple components v_{dp} and v_{mp} from management and data pairs, respectively, solve this problem. They allow the creation of a new document version by only creating one new pair — the one that the user is able to create — and adopting an existing pair for the pair that the user is unable to create. Then, components v_{dp} and v_{mp} reference the version number of the adopted pair. If a user holds both, ownership and write access rights, she creates a management pair and a data pair for that document version, and tuple components v_{dp} and v_{mp} take the version number as their value. If a user is only able to create either the management pair or the data pair for a new document version, then the v_{dp} or v_{mp} component of the new pair is the same as the v_{dp} or v_{mp} component from the previous pair of the same type.

For example, consider our example document collection from Page 38. The collection contains document $d_{1,1} = (\mathrm{D}{:}1, \mathrm{V}{:}1, \mathrm{R}{:}\mathbb{A}(b), \mathrm{W}{:}\mathbb{A}(a), \mathrm{O}{:}\mathbb{A}(b), \mathrm{K}{:}\{w_1, w_2\})$ that later gets updated to a new version $d_{1,2} = (\mathrm{D}{:}1, \mathrm{V}{:}2, \mathrm{R}{:}\mathbb{A}(c), \mathrm{W}{:}\mathbb{A}(a), \mathrm{O}{:}\mathbb{A}(b), \mathrm{K}{:}\{w_2, w_3\})$ and there are three users holding access rights $\{a\}$, $\{b\}$, and $\{b, c\}$, where access policy $\mathbb{A}(i)$ is satisfied by access right i for $i \in \{a, b, c\}$. As we can see, the contents of (entry K of) the document tuple changes, so the user performing the update needs write access to the document. Only one user holds these access rights and that user does not have ownership access to the document, which also is not required for the update, because the update does not modify the document's write and ownership policies. The management and data pairs for the document then look as follows. With ABE scheme ABE and signature scheme Σ, before the update, we have management pair

$$\left((1,1), \begin{pmatrix} 1, \mathsf{ABE.Enc}(\mathbb{A}(b), \mathit{dsk}_{\text{owner}}), \mathsf{ABE.Enc}(\mathbb{A}(a), \mathit{dsk}_{\text{writer}}), \\ \mathit{dvk}_{\text{owner}}, \mathit{dvk}_{\text{writer}}, \mathit{crt}_{\text{initial creator}}, \\ \Sigma.\mathsf{Sign}(\mathit{dsk}_{\text{owner}}, \dots), \Sigma.\mathsf{Sign}(\mathit{usk}_{\text{initial creator}}, \dots) \end{pmatrix}\right)$$

and data pair

$$\left((1,1), \begin{pmatrix} 1, \mathsf{ABE.Enc}(\mathbb{A}(b), \{w_1, w_2\}), \mathit{crt}_{\text{initial creator}}, \\ \Sigma.\mathsf{Sign}(\mathit{dsk}_{\text{writer}}, \dots), \Sigma.\mathsf{Sign}(\mathit{usk}_{\text{initial creator}}, \dots) \end{pmatrix}\right)$$

stored at the server. After the update, a new data pair

$$\left((1,2), \begin{pmatrix} 1, \mathsf{ABE.Enc}(\mathbb{A}(c), \{w_2, w_3\}), \mathit{crt}_{\text{user } \{a\}}, \\ \Sigma.\mathsf{Sign}(\mathit{dsk}_{\text{writer}}, \dots), \Sigma.\mathsf{Sign}(\mathit{usk}_{\text{user } \{a\}}, \dots) \end{pmatrix}\right)$$

exists at the server.

Given these pairs, both the server and users are given two means to *verify the pairs*, particularly, to verify the pairs' signatures. The first means is given by the signatures under document owners' and writers' keys. The second means is given by the access policies used in the encryption of the document owners' and writers' keys together with user attributes from the certificate contained in management and data pairs, and the pairs' signatures that are verified relative to the signature verification key from that certificate: The certificate's attributes must satisfy the respective policies and the creator's signature must be valid under the certificate's verification key.

Updates and time dependence. In dSEAC, the various SEAC instances share a common dictionary HT. Due to the short key requirement that we impose on dSEAC, dSEAC user keys should be about as long as SEAC user keys. Thus, we cannot have separate user keys for each of the SEAC instances. Therefore, we must reuse SEAC keys for multiple SEAC instances, but this opens up the possibility for the server to take a search query it has seen before and apply it to a new SEAC instance. The server having this option means that this naïve approach of key reuse is not forward private. We thus augment the inner workings of HT to consider creation times to rule out the outlined application of an old query to a new SEAC instance. Naturally, queries then must consider time as well. As a consequence of queries considering time, a challenge from SEAC re-occurs, namely, that the server may use time dependent bits from one search query with time-independent bits (particularly, attributes) from another search query for the same keyword.

The solution to this challenge is essentially the same as in SEAC: we introduce time dependent attributes that we include in access policies for ABE ciphertexts in array A. This involves a new namespace Tme for these time dependent attributes. Just like with keyword dependent attributes (namespace Wrd), users obtain all time dependent attributes during operation UserJoin.[2]

For making HT entries time dependent, we adopt Bost's technique, c.f. Section 5.5, from the Σοφος scheme [Bos16]. Specifically, we derive labels and decryption keys for HT entries via a trapdoor permutation. In Σοφος, the number of applications of the trapdoor permutation, for each keyword, equals the number of documents the keyword occurs in. For dSEAC, this means users need to know occurrence counters for each keyword, and these counters need to be shared between users. The need to maintain a consistent view on the counter values makes this approach infeasible in dSEAC, specifically because users and the server may behave maliciously, and thus there is no reason for users to believe in the correctness of received counter values. Therefore, in our adaption of Bost's technique, the number of applications of the trapdoor permutation is independent of occurrence counters, and instead uses the discrete system time, that is the number of Update operations that have been performed at and accepted by the server. Hence, the i-th SEAC instance uses i applications of the trapdoor permutation.

The application of Bost's technique and time dependent attributes to our example document collection is shown in Figure 6.11. As can be seen from the figure, the labels and decryption keys for HT entries are derived, via PRF f_D, from keyword dependent PRF keys and seeds for the trapdoor permutations. We derive the keyword dependent PRF keys and the keyword dependent seed for the trapdoor function from the relevant keyword. For the derivation process, we use three PRFs, denoted f_K, f_L, and f_S in the figure, with keys k_K, k_L, and k_S, respectively. Users get these PRF keys, as well as the secret key sk_T for the trapdoor permutation π_T as part of their secret key. The public key pk_T for the trapdoor permutation is part of dSEAC's public parameters pp.

Verifiability, user states, and last completed operations. For the dSEAC scheme to be verifiable, we rely on techniques from the literature on authenticated data structures and combine these techniques with signatures. Particularly, we rely on AADs (Definition 24), signatures with delayed verifiability (Definition 22), and the concept of "last completed

[2]These may be a lot of attributes, but satisfy the short key requirement in that users get one attribute per SEAC instance rather than one attribute per keyword per SEAC instance. Using appropriate techniques, the number of time dependent attributes may be reduced to a number logarithmic in the number of SEAC instances — by bit decomposition of the bit string representing time. Then, ABE's key delegation and collusion resistance among delegated keys prevent the server from combining bits from multiple search queries.

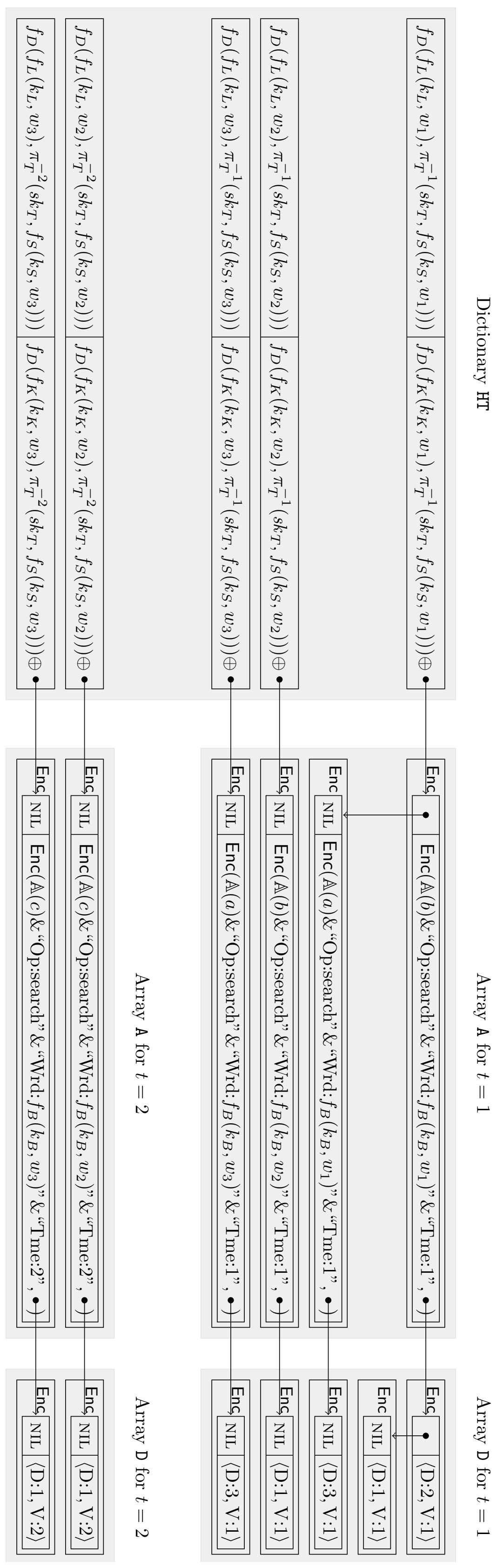

Dictionary HT
$f_D(f_L(k_L, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1)))$
$f_D(f_K(k_K, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1)))\oplus$
$f_D(f_L(k_L, w_2), \pi_T^{-1}(sk_T, f_S(k_S, w_2)))$
$f_D(f_K(k_K, w_2), \pi_T^{-1}(sk_T, f_S(k_S, w_2)))\oplus$
$f_D(f_L(k_L, w_3), \pi_T^{-1}(sk_T, f_S(k_S, w_3)))$
$f_D(f_K(k_K, w_3), \pi_T^{-1}(sk_T, f_S(k_S, w_3)))\oplus$
$f_D(f_L(k_L, w_2), \pi_T^{-2}(sk_T, f_S(k_S, w_2)))$
$f_D(f_K(k_K, w_2), \pi_T^{-2}(sk_T, f_S(k_S, w_2)))\oplus$
$f_D(f_L(k_L, w_3), \pi_T^{-2}(sk_T, f_S(k_S, w_3)))$
$f_D(f_K(k_K, w_3), \pi_T^{-2}(sk_T, f_S(k_S, w_3)))\oplus$
Array A for $t = 1$
Enc
Enc(𝔸(b)&"Op:search"&"Wrd:$f_B(k_B, w_1)$"&"Tme:1",
NIL
Enc(𝔸(a)&"Op:search"&"Wrd:$f_B(k_B, w_1)$"&"Tme:1",
Enc(𝔸(b)&"Op:search"&"Wrd:$f_B(k_B, w_2)$"&"Tme:1",
Enc(𝔸(a)&"Op:search"&"Wrd:$f_B(k_B, w_3)$"&"Tme:1",
Array A for $t = 2$
Enc(𝔸(c)&"Op:search"&"Wrd:$f_B(k_B, w_2)$"&"Tme:2",
Enc(𝔸(c)&"Op:search"&"Wrd:$f_B(k_B, w_3)$"&"Tme:2",
Array D for $t = 1$
⟨D:2, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩
Array D for $t = 2$
⟨D:1, V:2⟩
⟨D:1, V:2⟩

Figure 6.11: dSEAC's post-update index structure for our example document collection from Page 38, including applications of Bost's technique and time dependent attributes

operation" (adapted from Cachin and Geisler [CG09]) for users to share a consistent view on the system.

dSEAC inherits the partial pre-computed search results that SEAC stores in `D` and, indirectly, `A` lists. We apply signatures with delayed verifiability to these pre-computed search results. The signatures ensure verifiability of pre-computed results, so users can verify the integrity of partial search results presented to them. Furthermore, we use AADs to enable users to check that they have received all partial pre-computed search results relevant to their search queries. The dictionary digests necessary for verifying that all pre-computed search results have been received are stored as part of users' states. User states represent the views users have on the system, and these views must be synchronized between users. The last completed operation serves these synchronization purposes.

We now discuss the use and impact of signatures with delayed verifiability, AADs, and the last completed operation. First, we introduce the concept of *list authenticators*. A list authenticator is a signature with delayed verifiability on the linked lists that represent pre-computed search results, namely, `A` and `D` lists. However, we only sign the lists' contents, while we ignore administrative data used to maintain the list structure. A list authenticator on a list is stored alongside the pointer to the respective list. The list authenticator's open value is stored as part of the *encrypted* pointer to the respective list.

As a result, a list authenticator on a `D` list is a signature with delayed verifiability on the sequence of pointers (to documents) stored in the `D` list. The authenticator is stored in an `A` list node. Alongside the list authenticator on the `D` list, we explicitly store the read access policy shared by the documents in that `D` list.

Thus, each `A` list node stores a pointer to its successor node, a list authenticator on a `D` list, the access policy shared by the documents the `D` list points to, and an ABE ciphertext of a pointer to that `D` list. We compute the list authenticator on an `A` list as a signature with delayed verifiability on the sequence of pairs of list authenticators and access policies stored in that `A` list. The list authenticators on `A` lists are stored in `HT` entries pointing to the respective lists. An example of this arrangement can be seen in Figure 6.12.

The use of list authenticators is a first step towards verifiability of dSEAC. Particularly, during search, the server not only includes documents in search results, but also all list authenticators on `A` and `D` lists, policies from accessed `A` lists, and open values for list authenticators on accessed `A` and `D` lists. Then, using the list authenticators, the user who receives the result verifies the results as follows.

1. List authenticators on `A` list are used to check that all pairs of list authenticators and access policies from that `A` list are included in the search result (assuming the creator of the list was honest; if the creator did not behave honestly, the server engages in conflict resolution, see below). For this test, verification algorithm VSig of the signature scheme with delayed verifiability is applied to the list authenticator and the reported policy-authenticator pairs.

2. The reported policies from `A` list nodes are used to determine what existing `D` lists are accessible using the user's access rights. All such `D` lists should have been accessed and reported in the presented result, and thus, the list authenticators on the respective `D` lists should be verifiable in the next step.

3. List authenticators on accessible `D` lists are used to check that the server has accessed the respective lists and has included all document pointers from the respective `D` lists in the search result. For this test, for each of the accessible lists, verification algorithm VSig of the signature scheme with delayed verifiability is applied to the

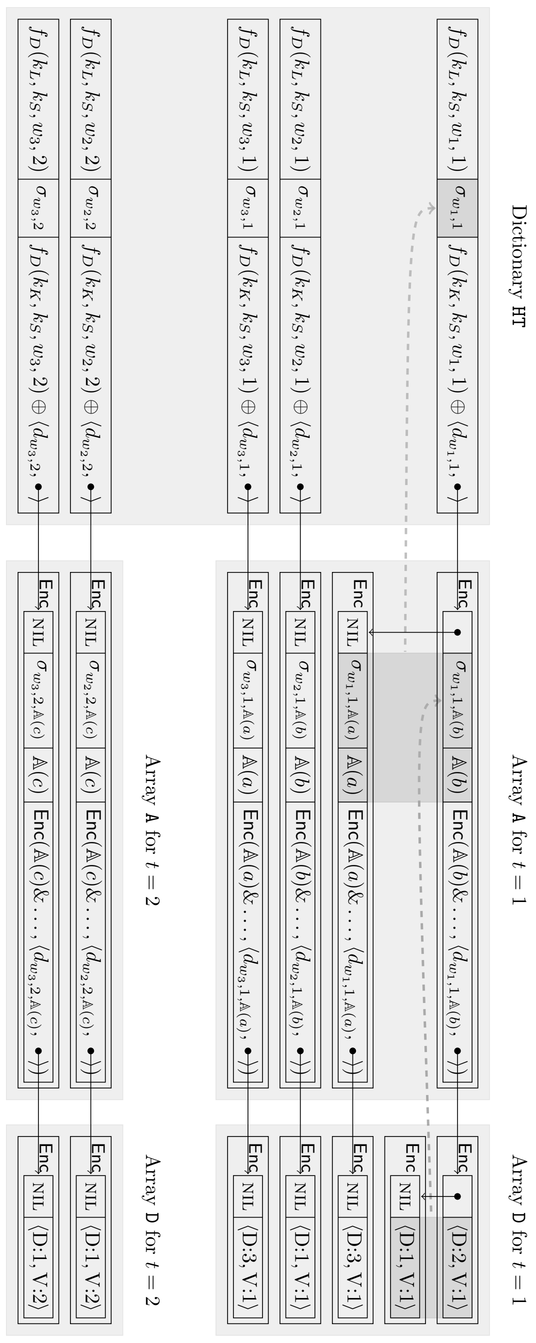

Dictionary HT
$f_D(k_L, k_S, w_1, 1)$
$\sigma_{w_1,1}$
$f_D(k_K, k_S, w_1, 1) \oplus \langle d_{w_1,1}, \bullet\rangle$
$f_D(k_L, k_S, w_2, 1)$
$\sigma_{w_2,1}$
$f_D(k_K, k_S, w_2, 1) \oplus \langle d_{w_2,1}, \bullet\rangle$
$f_D(k_L, k_S, w_3, 1)$
$\sigma_{w_3,1}$
$f_D(k_K, k_S, w_3, 1) \oplus \langle d_{w_3,1}, \bullet\rangle$
$f_D(k_L, k_S, w_2, 2)$
$\sigma_{w_2,2}$
$f_D(k_K, k_S, w_2, 2) \oplus \langle d_{w_2,2}, \bullet\rangle$
$f_D(k_L, k_S, w_3, 2)$
$\sigma_{w_3,2}$
$f_D(k_K, k_S, w_3, 2) \oplus \langle d_{w_3,2}, \bullet\rangle$
Array A for $t = 1$
Enc
NIL
$\sigma_{w_1,1,\mathbb{A}(b)}$
$\mathbb{A}(b)$
$\mathsf{Enc}(\mathbb{A}(b)\& \ldots, \langle d_{w_1,1,\mathbb{A}(b)}, \bullet\rangle)$
$\sigma_{w_1,1,\mathbb{A}(a)}$
$\mathbb{A}(a)$
$\mathsf{Enc}(\mathbb{A}(a)\& \ldots, \langle d_{w_1,1,\mathbb{A}(a)}, \bullet\rangle)$
$\sigma_{w_2,1,\mathbb{A}(b)}$
$\mathbb{A}(b)$
$\mathsf{Enc}(\mathbb{A}(b)\& \ldots, \langle d_{w_2,1,\mathbb{A}(b)}, \bullet\rangle)$
$\sigma_{w_3,1,\mathbb{A}(a)}$
$\mathbb{A}(a)$
$\mathsf{Enc}(\mathbb{A}(a)\& \ldots, \langle d_{w_3,1,\mathbb{A}(a)}, \bullet\rangle)$
Array A for $t = 2$
$\sigma_{w_2,2,\mathbb{A}(c)}$
$\mathbb{A}(c)$
$\mathsf{Enc}(\mathbb{A}(c)\& \ldots, \langle d_{w_2,2,\mathbb{A}(c)}, \bullet\rangle)$
$\sigma_{w_3,2,\mathbb{A}(c)}$
$\mathbb{A}(c)$
$\mathsf{Enc}(\mathbb{A}(c)\& \ldots, \langle d_{w_3,2,\mathbb{A}(c)}, \bullet\rangle)$
Array D for $t = 1$
⟨D:2, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩
⟨D:1, V:1⟩
⟨D:3, V:1⟩
Array D for $t = 2$
⟨D:1, V:2⟩
⟨D:1, V:2⟩

Figure 6.12: dSEAC's post-update index structure for our example document collection from Page 38; in extension to the time dependent mechanics introduced in Figure 6.11, this figure shows changes to the structure due to list authenticators; the data marked gray is authenticated by the list authenticators the respective arrows point to

authenticator on the respective list and the reported document pointers from that list.

4. Verification algorithm VOrig of the signature scheme with delayed verifiability is applied to the list authenticators on D lists not accessible using the user's access rights.

For all of the above signature verifications, the same signature verification key is used.

All in all, list authenticators enable verification of search results on a per-SEAC-instance basis. We build upon this preliminary result by specifically implementing the dictionary HT by an AAD. Using Bost's technique, necessitated by time dependence, during search, the server essentially computes search queries for each individual SEAC instance, which results in a Lookup query to HT for each SEAC instance. The answers to these queries contain list authenticators on A lists. By including proofs for the results of the Lookup operations in the search result, users can use the proofs and list authenticators on A lists to determine that the server has provided correct answers for each SEAC instance considered by the server. If users also knows how many SEAC instances the server needs to consider, they can also check that the server has considered all SEAC instances.

Before we handle this condition of users knowing the number of SEAC instances, we need to mention that the dictionaries Man and Dat used for enforcing access restrictions to documents are also implemented as AADs. Then, if all users know the digests of these dictionaries, documents can never be deleted or overridden. This condition, users knowing digests of dictionaries Man and Dat, can be addressed by the same mechanism as the requirement of users knowing the number of SEAC instances in the system.

Both issues are solved using the concept of *last completed operations*. A last completed operation is a digest of the server's state after completion of the most recent operation that has modified the server state. The server state is modified by Init, Update, and Resolve operations. The server stores the post-operation digest in its lco variable as part of its state and registers lco with the update log UL by appending lco to the AAD that implements UL. Appending lco changes UL's digest, and the updated digest and append-only proof are stored in variable dul.

Whenever a user interacts with the server, the user receives a copy of lco and dul. She then uses her user state, lco, dul, and additional data provided by the server to check that the server is in a state that can be reached form the state the server was in the last time the user performed such a check. If the check is successful, the user adopts the received lco and dul tuples as her new user state. As a consequence, the current lco and dul are used for the next check.

On a technical level, lco is a 5-tuple

$$(t, dig, ch, tvk, \sigma),$$

where t is the current discrete time stamp, counting the total number of Init, Update, and Resolve operations that have been performed so far. Component $dig = (\text{digest}(\texttt{HT}), \text{digest}(\texttt{Man}), \text{digest}(\texttt{Dat}), \text{digest}(\texttt{UL}))$ consists of dictionary digests for the four AADs HT, Man, Dat, and UL, respectively. Component σ of the lco tuple is a signature on components t to tvk; the respective signature verification key is provided as part of tvk. Depending on who has performed the operation, tvk is either the judge's signature verification key or the certificate of the user who has performed the operation. Tuple component ch is a summary of the changes made to the server state during the operation that resulted in lco being the current server state's digest; this summary is left empty for Init operations, it is a user's signature (with delayed verifiability) on the SEAC instance computed as part of the operation for Update operations (in this case the open value is provided as part of the signature), and for Resolve operations, *ch* is the judge's signature from a notification that resulted from the respective Resolve operation (see below for notifications and Resolve operations). While component *ch* is largely irrelevant to users, it plays a role in conflict resolution.

Given the structure of `lco`, we can determine what checks users perform to conclude that the server's claimed state (in the form of a new `lco` and `dul`) can be reached from a previously verified state (in the form of the old `lco` stored as the user's state). The following conditions must be met for a user to adopt `lco` as her new state.

1. The signature σ from `lco` must be valid.
2. `lco` must have been registered with `UL` (relative to `UL`'s digest from `dul`)
3. `UL`'s digest from `dul` must be an immediate successor of `UL`'s digest from `lco` (according to the append-only proof from `dul`).
4. The old user state must contain the only last completed operation `UL` associates with the state's time stamp (relative to `lco`'s `UL` digest).
5. Digests of AADs `HT`, `Man` and `Dat` from `lco` must be successors of the corresponding digests from the old user state.

These conditions are dropped or tweaked if the user holds no previous state due to having been enrolled only recently, if the server has not been initialized yet, or if `lco` originates from server initialization.

Intuitively, `dul` tuples represent the difference between pre-operation and post-operation `UL` digests. To that end, the `dul` tuple that corresponds to an `lco` tuple is a 3-tuple. It contains the digest of dictionary `UL` after `lco` has been added, an append-only proof linking the `UL` digests from `lco` and `dul`, and a proof that the digest from `dul` represents a dictionary that contains `lco`.

Relying on the `lco` tuple, users, during search, know how many SEAC instances the server stores at most. Thus, users know for how many instances the server has to report search results. Particularly, for each SEAC instance, the server has to report the result for one Lookup query on dictionary `HT`. This then allows users to determine that the server has considered all SEAC instance when computing the result. Since the answers to Lookup queries also contain list authenticators on `A` lists for each considered SEAC instance, the per-SEAC-instance search results are indirectly authenticated by the `HT` digest from `lco`.

Notifications and conflict resolution. The goal of conflict resolution is to enable the server to blame a malicious user's actions on that user. For example, a malicious user may submit data as part of an Update operation that is flawed, but cannot be detected as being flawed during the operation. Examples of flaws include NIL pointers, pointers to non-existing addresses, and wrong list decryption keys in `HT` entries or inside the ABE ciphertexts of `A` list nodes, as well as list authenticators that can be origin verified, but not signature verified relative to the data the authenticators are supposed to authenticate. In all of these examples, the flaws in the data are hidden by layers of encryption.

As a result, the server can detect the flaws only when the data gets decrypted during Search operations. If the server finds data to be flawed, it engages in conflict resolution with the judge. During conflict resolution, the server convinces the judge that the flawed data originates from a successful Update operation. For that, the server sends the SEAC instance that contains the flawed data and the search query that has lead to the detection of the flaw to the judge. Alongside, the server sends proofs that the presented SEAC instance occurs in the history of Update operations that has lead to the current server state. This includes sending the current lco and dul tuples to the judge. The judge verifies the server's claims against her own state and then verifies the proofs and re-performs the Search operation that resulted in the flaw being detected. The judge determines the SEAC instance to be flawed if the re-performed operation triggers conflict resolution. If the judge finds the SEAC instance to be flawed, it issues a notification, publicly certifying the flawed nature of the instance.

For notifications to serve their purpose, they take the following form. A notification is a tuple

$$(\mathtt{lco}, t_{\text{affected}}, \sigma),$$

where lco is the lco tuple the server has sent to the judge, t_{affected} is the time stamp of the investigated SEAC instance, and σ is a signature on tuple components lco and t_{affected} under the judges signing key.

After the judge has concluded her investigation, she advances time by essentially performing an Update with an empty document collection, registering the notification with dictionaries HT and UL in the process. This process also updates the judge's state.

When users obtain a notification as part of a search result, they need to *verify the notification,* because users verify search results, and the notification replaces a search result, so it gets treated identically. For verifying notifications, users verify the notifications' signature σ relative to the judges verification key and verify via AAD.VAppend that the notification's lco occurs in the history of updates that lead to the current user state; the server provides the necessary proofs.

We stress that neither the server nor the judge attempt to fix a flawed SEAC instance. After all, the flawed instance may originate from a malicious user. Instead of attempting to fix the instance, the instance is effectively discarded. However, the instance cannot be erased from history, because that would require deleting an entry from UL, which is not possible in an AAD. So instead, the judge issues a notification, essentially ordering users to ignore the flawed SEAC instance.

Additional extensions. For our security proofs, we need to make one additional change to the internal structure of HT entries. Specifically, we include an additional layer of encryption in HT entries' values. Above, HT entries are shown to take the form

$$(\ell_{kw,t}, (\sigma_{kw,t}, k_{kw,t} \oplus \langle d_{kw,t}, p_{kw,t} \rangle)),$$

where $\ell_{kw,t}$ is the label of the HT entry for keyword kw and time step t, $\sigma_{kw,t}$ is the list authenticator of the A list the HT entry points to, $d_{kw,t}$ is the corresponding open value, $p_{kw,t}$ is the pointer to the A list, and $k_{kw,t}$ is an encryption key that depends on kw and t, and is used to mask the open value and the pointer. Our additional layer of encryption affects the pointer and the open value, so we use symmetric encryption with a fresh key for encrypting the open value and the pointer, while masking the fresh key by $k_{kw,t}$. As a result, HT entries take the form

$$(\ell_{kw,t}, (\sigma_{kw,t}, k_{kw,t} \oplus k', \mathsf{Sym.Enc}(k', \langle d_{kw,t}, p_{kw,t} \rangle))),$$

where k' is freshly sampled for every HT entry.

This extension is required in our proof of dSEAC's data confidentiality to deal with adversaries causing forks in the system. Particularly, tuple component $k_{kw,t}$ deterministically depends on kw and t and thus may occur on two forks. Our extension then ensures that the pointer and list authenticator pairs on the forks are encrypted under independent symmetric keys, rather than identical symmetric keys.

6.3.2 dSEAC, Formally

Implementing the ideas laid out above, we construct the dSEAC scheme. Naturally, dSEAC's algorithms and protocols resemble SEAC's, so in our description of the algorithms and protocols we focus on computations not present in SEAC. Note that our presentation features a helper operation LCO-Update that updates users' states to the server's `lco` and `dul` as described before. Our presentation also features a complementary helper operation LCO-Registration for registering a new `lco` tuple with the server's update log `UL`. The helper operations are used as sub-routines in various other operation.

Construction 36. *The* dSEAC *scheme consists of algorithms and protocols* (Setup, UserJoin, Init, Update, Search, Resolve, Version) *shown in Figures 6.13–6.18.*

We now briefly discuss dSEAC's operations. The Setup operation is executed by a trusted party and computes PRF keys, keys to a trapdoor permutation, sets up ABE and three AADs, parameters for commitment scheme Γ, and computes signing keys for the key issuer and the judge. As signature scheme Σ, we use a signature scheme with delayed verifiability (following the hash–then–commit approach, c.f. Figure 3.20) and externally supplied commitment parameters, meaning that the scheme's KeyGen algorithm takes the public parameters of a commitment scheme as input, and these parameters become part of the verification key output by KeyGen, c.f. Section 3.4.

During the UserJoin operation, the user computes her singing key (for signature scheme with delayed verifiability Σ) and requests her (extended) SEAC key from the key issuer. The key issuer provides the user with the requested key, which contains the requested attributes, attributes for operations and keywords (as in SEAC), and attributes for all time stamps that will ever be used, and publishes a certificate on the user's requested attributes and signature verification key. Details on algorithms and protocols Setup, UserJoin are shown in Figure 6.13.

The LCO-Update operation is executed as a sub-routine in protocols Update, Search and Resolve, and is executed between the user or judge, and the server. The operation outputs 0 to the user or judge if the any of the checks fail outputs 1 to the user if the server reports a concurrent Update or Resolve operation, and otherwise outputs (`lco`, `dul`) to the user or judge. Details on protocol LCO-Update are shown in Figure 6.14.

Operation LCO-Registration is a helper protocol executed by the server, and the user or judge during Init, Update, and Resolve operations. The operation registers the most recent `lco` tuple with the update log `UL`, computes the `dul` value for the registration process, and sends `dul` to the user or judge.

During operation Init, the server initializes four empty AADs and sends the corresponding dictionary digests to the user, who then computes the first last completed operation (`lco`) and registers it with the server. The user adopts `lco` and the corresponding `dul` as her state. The server initializes additional data structures, and stores the dictionaries, additional data structures and `lco` as its state. Details on protocols LCO-Registration and Init are shown in Figure 6.15.

For operation Update, the user first updates her user state to the current `lco` and `dul`, and then determines whether she is able to perform the update relative to her rights and the write access restrictions imposed by previous versions of the document she wants to write. If the user is able to perform the update, she computes the necessary management and data tuples (as a replacement for SEAC's `CT`), and computes the index structures `HT`,

```
$\underline{\mathsf{Setup}(1^\lambda)}$
• $k_B \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_L \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_S \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $(sk_T, pk_T) \leftarrow \mathsf{TDP.KeyGen}(1^\lambda)$
• $(pp', msk') \leftarrow \mathsf{ABE.Setup}(1^\lambda)$
• $cpp \leftarrow \mathsf{\Gamma.Setup}(1^\lambda)$
• $(isk, ivk) \leftarrow \mathsf{\Sigma.KeyGen}(1^\lambda, cpp)$
• $(j\varsigma k, jvk) \leftarrow \mathsf{\Sigma.KeyGen}(1^\lambda, cpp)$
• $(pp_{\mathtt{HT}}, vk_{\mathtt{HT}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda, \beta(\lambda))$
• $(pp_{\mathtt{Man}}, vk_{\mathtt{Man}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda, \beta(\lambda))$
• $(pp_{\mathtt{Dat}}, vk_{\mathtt{Dat}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda, \beta(\lambda))$
• $(pp_{\mathtt{UL}}, vk_{\mathtt{UL}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda, \beta(\lambda))$
• $msk \leftarrow (msk', isk, k_B, k_K, k_L, k_S, sk_T)$
• $pp \leftarrow (pp', cpp, ivk, jvk, pp_{\mathtt{HT}}, vk_{\mathtt{HT}}, pp_{\mathtt{Man}}, vk_{\mathtt{Man}}, pp_{\mathtt{Dat}}, vk_{\mathtt{Dat}}, pp_{\mathtt{UL}}, vk_{\mathtt{UL}})$
• $jsk \leftarrow (j\varsigma k, k_L, sk_T)$
• $st_{\text{judge}} \leftarrow \perp$
• output $(pp, msk, jsk, st_{\text{judge}})$

$\underline{\mathsf{UserJoin}(pp, U | pp, msk)}$
U: $(u\varsigma k, uvk) \leftarrow \mathsf{\Sigma.KeyGen}(1^\lambda, cpp)$
U: $\sigma \leftarrow \mathsf{\Sigma.Sign}(u\varsigma k, \langle U, uvk \rangle)$
U: send $U, uvk, \sigma$ to key issuer
KI: if $\mathsf{\Sigma.VSig}(uvk, \langle U, uvk \rangle, \sigma) \neq 1$: abort protocol
KI: $uak \leftarrow \mathsf{ABE.KeyGen}(msk', \{\text{"Usr:}u\text{"}\}_{u \in U} \cup \{\text{"Op:read"}, \text{"Op:search"}\} \cup \{\text{"Wrd:}\mathsf{PRF.E}$
    $\{\text{"Tme:}t\text{"}\}_{t \in timestamps}$
KI: $\sigma' \leftarrow \mathsf{\Sigma.Sign}(isk, \langle uvk, U \rangle)$
KI: $crt \leftarrow (uvk, U, \sigma')$
KI: $usk' \leftarrow (uak, k_B, k_K, k_L, k_S, sk_T,)$
KI: publish $crt$
KI: send $(usk, crt)$ to user
U: $usk \leftarrow (usk', u\varsigma k, crt)$
U: $st_{\text{user}} \leftarrow \perp$
U: output $(usk, st_{\text{user}})$ to user
```

Figure 6.13: Algorithm Setup and protocol UserJoin of the dSEAC scheme

```
$\underline{\mathsf{LCO\text{-}Update}(pp,st_{\text{user}}|pp,st_{\text{server}})}$
U: send “LCO-Update” to server
S: if $st_{\text{server}} = \bot$: send “not initialized” to user and abort protocol
S: if an Update or Resolve operation is being performed: send “concurrent operation” to
   user and abort protocol
S: send ($\texttt{lco}$,$\texttt{dul}$) to user
U: if received “not initialized” and $st_{\text{user}} \neq \bot$: output 0 to user and abort protocol
U: if received “not initialized” or “concurrent operation:” output 1 to user and abort
   protocol
U: if $\texttt{lco} = st_{\text{user}}.\texttt{lco}$ and $\texttt{dul} = st_{\text{user}}.\texttt{dul}$:
   • send “user initialized” to server
   • output ($\texttt{lco}$,$\texttt{dul}$) to user
   • abort protocol
U: if $\Sigma.\mathsf{VSig}(\texttt{lco}.tvk,\langle\texttt{lco}.t,\texttt{lco}.dig,\texttt{lco}.ch,\texttt{lco}.tvk\rangle,\texttt{lco}.\sigma) \neq 1$ or $\mathsf{AAD.VAppend}(vk_{\texttt{UL}},$
   $\texttt{lco}.\text{digest}(\texttt{UL}),\texttt{dul}.\text{digest}(\texttt{UL}),\texttt{dul}.\pi_{\mathsf{Append}}) \neq 1$ or $\mathsf{AAD.Version}(\texttt{dul}.\text{digest}(UL)) \neq$
   $\mathsf{AAD.Version}(\texttt{lco}.dig.\text{digest}(UL))+1 \neq \texttt{lco}.t+1$ or $\mathsf{AAD.VResult}(vk_{\texttt{UL}},\texttt{dul}.\text{digest}(UL),$
   $\texttt{lco}.t,\{\texttt{lco}\},\pi_{\mathsf{Lookup}}) \neq 1$ or $v_{\texttt{lco}} \neq \{\texttt{lco}\}$ : output 0 to user and abort protocol
U: if $\texttt{lco}.t = 0$:
   • if $\texttt{lco}.ch \neq \emptyset$ or $\mathsf{AAD.Version}(\texttt{lco}.dig.\text{digest}(\texttt{HT})) \neq 0$ or $\mathsf{AAD.Version}(\texttt{lco}.dig.\text{digest}(\texttt{M}$
     0 or $\mathsf{AAD.Version}(\texttt{lco}.dig.\text{digest}(\texttt{Dat})) \neq 0$ or $\mathsf{AAD.Version}(\texttt{lco}.dig.\text{digest}(\texttt{UL})) \neq 0$:
     output 0 to user and abort protocol
U: if $st_{\text{user}} = \bot$:
   • send “user initialized” to server
   • output ($\texttt{lco}$,$\texttt{dul}$) to user
   • abort protocol
U: if $\texttt{lco}.t < st_{\text{user}}.\texttt{lco}.t$: output 0 to user and abort protocol
U: if $\texttt{lco}.ch \neq \emptyset$ and $\Sigma.\mathsf{VOrig}(\texttt{lco}.tvk,\texttt{lco}.ch) \neq 1$: output 0 to user and abort protocol
U: send $st_{\text{user}}.\texttt{lco}.t$ to server
S: if received “user initialized:” output $\bot$ to server and abort protocol
S: determine dictionaries $\texttt{HT}'$,$\texttt{Man}'$,$\texttt{Dat}'$ for time stamp $st_{\text{user}}.\texttt{lco}.t$ from $\texttt{UL}$
S: determine dictionaries $\texttt{UL}'$,$\texttt{UL}^*$ for time stamps $st_{\text{user}}.\texttt{lco}.t+1$ (as recorded by digest
   in $\texttt{dul}$ tuple for $st_{\text{user}}.\texttt{lco}$) and $st_{\text{server}}.\texttt{lco}.t$ from $\texttt{UL}$
S: $(v_t,\pi_t) \leftarrow \mathsf{AAD.Lookup}(pp_{\texttt{UL}},\texttt{UL}^*,t)$
S: $\pi_{\texttt{HT}} \leftarrow \mathsf{AAD.PAppend}(pp_{\texttt{HT}},\texttt{HT}',\texttt{HT})$
S: $\pi_{\texttt{Man}} \leftarrow \mathsf{AAD.PAppend}(pp_{\texttt{Man}},\texttt{Man}',\texttt{Man})$
S: $\pi_{\texttt{Dat}} \leftarrow \mathsf{AAD.PAppend}(pp_{\texttt{Dat}},\texttt{Dat}',\texttt{Dat})$
S: $\pi_{\texttt{UL}} \leftarrow \mathsf{AAD.PAppend}(pp_{\texttt{UL}},\texttt{UL}',\texttt{UL}^*)$
S: send $(\pi_t,\pi_{\texttt{HT}},\pi_{\texttt{Man}},\pi_{\texttt{Dat}},\pi_{\texttt{UL}})$ to user
U: if $\mathsf{AAD.VResult}(vk_{\texttt{UL}},\texttt{lco}.dig.\text{digest}(\texttt{UL}),t,\{st_{\text{user}}.\texttt{lco}\},\pi_t) \neq 1$ or $\mathsf{AAD.VAppend}(vk_{\texttt{HT}},$
   $st_{\text{user}}.dig.\text{digest}(HT),\texttt{lco}.dig.\text{digest}(HT),\pi_{\texttt{HT}}) \neq 1$ or $\mathsf{AAD.VAppend}(vk_{\texttt{Man}},st_{\text{user}}.dig.\text{dig}$
   $\texttt{lco}.dig.\text{digest}(\texttt{Man}),\pi_{\texttt{Man}}) \neq 1$ or $\mathsf{AAD.VAppend}(vk_{\texttt{Dat}},st_{\text{user}}.dig.\text{digest}(\texttt{Dat}),\texttt{lco}.dig,$
   $\text{digest}(\texttt{Dat}),\pi_{\texttt{Dat}}) \neq 1$ or $\mathsf{AAD.VAppend}(vk_{\texttt{UL}},st_{\text{user}}.\texttt{dul}.\text{digest}(\texttt{UL}),\texttt{lco}.dig.\text{digest}(\texttt{UL}),$
   $\pi_{\texttt{UL}}) \neq 1$: output 0 to user and abort protocol
U: output ($\texttt{lco}$,$\texttt{dul}$) to user
```

Figure 6.14: Helper operation LCO-Update of the SEAC scheme; the helper operation is used as part of protocols Update, Search, and Resolve

```
$\underline{\mathsf{LCO\text{-}Registration}(pp, st_{\text{server}}, \texttt{lco} | pp, st_{\text{user}}, \texttt{lco})}$
S: $\texttt{UL}' \leftarrow \texttt{UL}$
S: $(\texttt{UL}, \text{digest}(\texttt{UL})) \leftarrow \mathsf{AAD.Append}(pp_{\texttt{UL}}, \texttt{UL}, \{(\texttt{lco}.t, \texttt{lco})\})$
S: $\pi_{\mathsf{Append}} \leftarrow \mathsf{AAD.PAppend}(pp_{\texttt{UL}}, \texttt{UL}', \texttt{UL})$
S: $(v_{\mathsf{Lookup}}, \pi_{\mathsf{Lookup}}) \leftarrow \mathsf{AAD.Lookup}(pp_{\texttt{UL}}, \texttt{UL}', \texttt{lco}.t)$
S: $\texttt{dul} \leftarrow (\text{digest}(\texttt{UL}), \pi_{\mathsf{Lookup}})$
S: send $\texttt{dul}$ to user
S: output $(\texttt{Ntf}, \texttt{lco}, \texttt{dul}, \texttt{HT}, \texttt{Man}, \texttt{Dat}, \texttt{UL})$ to server
U: if $\mathsf{VAppend}(vk_{\texttt{UL}}, \texttt{lco}.\text{digest}(\texttt{UL}), \texttt{dul}.\text{digest}(\texttt{UL}), \texttt{dul}.\pi_{\mathsf{Append}}) = 0$ or $\mathsf{VResult}(vk_{\texttt{UL}}, \texttt{dul}.$di
    0: output 0 to user and abort protocol
U: output $(\texttt{lco}, \texttt{dul})$ to user

$\underline{\mathsf{Init}(pp, usk, st_{\text{user}} | pp, st_{\text{server}})}$
U: if $st_{\text{user}} \neq \perp$: abort protocol
U: send "initialize" to server
S: if $st_{\text{server}} \neq \perp$: send "already initialized" to user and abort protocol
S: initialize empty AADs $\texttt{HT}$, $\texttt{Man}$, and $\texttt{Dat}$ with digests $\text{digest}(\texttt{HT})_0$, $\text{digest}(\texttt{Man})_0$, digest(Dat
    and $\text{digest}(\texttt{UL})_0$, respectively, from AAD parameters and verification keys from $pp$
S: send $dig = (\text{digest}(\texttt{HT})_0, \text{digest}(\texttt{Man})_0, \text{digest}(Dat)_0, \text{digest}(\texttt{UL})_0)$ to user
U: if received message "already initialized:" abort protocol
U: $\sigma \leftarrow \Sigma.\mathsf{Sign}(u\varsigma k, \langle 0, dig, \emptyset, crt \rangle)$
U: $\texttt{lco} \leftarrow (0, dig, \emptyset, crt, \sigma)$
U: send $\texttt{lco}$ to server
S: if $\texttt{lco}$ is malformed or $\Sigma.\mathsf{VSig}(crt.uvk, \langle 0, dig, \emptyset, crt \rangle, \texttt{lco}.\sigma) \neq 1$ or $\Sigma.\mathsf{VSig}(ivk, \langle crt.uvk,$
    $crt.U \rangle, crt.\sigma) \neq 1$: abort protocol
S: create empty data structure for storing $\texttt{Ntf}$
S: create empty data structures $ASet$ and $DSet$ for storing SEAC instances' $\texttt{A}$ and $\texttt{D}$
    memory arrays
S&U: $(s|u) \leftarrow \mathsf{LCO\text{-}Registration}(pp, st_{\text{server}}, \texttt{lco} | pp, st_{\text{user}}, \texttt{lco})$
S: output $st_{\text{server}} \leftarrow s$ to server
U: if $u = 0$: raise alarm and abort protocol
U: output $st_{\text{user}} = u$ to user
```

Figure 6.15: Helper operation LCO-Registration and protocol Init of the dSEAC scheme; the helper operation is used as part of protocols Init, Update, and Resolve

A, and D as in SEAC, including our extensions to HT entries and A list nodes, particularly, list authenticators and time dependence. The user sends the data to the server, which appends the dictionary entries to the respective AADs, resulting in new dictionary digests, which are sent back to the user. The user uses the new digests to compute a new lco, registers lco with the server, and adopts lco and the corresponding dul as her new state. Details on protocol Update can be found in Figure 6.16.

During operation Search, the user first updates her user state to the current lco and dul, and then computes a master query, which is sent to the server. From the master query, the server derives search queries for the individual SEAC instances, to which the queries are applied. The search result consists of the instance-specific results, including verification data like membership proofs for HT entries, and list authenticators and their open values. The user uses verification data to verify the result's completeness, and finally outputs document plaintexts that correspond to data pairs from the search result. Details on protocol Search can be found in Figure 6.17.

For protocol Resolve, the judge first updates her state to the current lco and dul. Then the server sends data to the judge, such that the judge can determine that a SEAC instance is indeed flawed, that the instance occurs in a history of updates that lead to lco, and that the flaw was detected when serving a recent search query. If the SEAC instance is found to be flawed, the judge computes a notification on the instance, registers the notification with dictionary HT, creates a new lco for registering the notification with the update log UL, registers lco with the server, and adopts lco and the corresponding dul as her new state.

Algorithm Version outputs a vector that consists of the current time stamp and the version numbers of the four dictionary digests. Details on operations Resolve and Version can be found in Figure 6.18.

Correctness. The dSEAC scheme is correct in that its Search operation either outputs $\perp$ (Condition 1) or numerous document plaintexts that satisfy Conditions 2–4 of the result correctness definition for searchable encryption schemes; this property is inherited from SEAC's search results. However, dSEAC can exclude documents from individual SEAC instances from the search result by including a notification on the respective SEAC instance in the search result instead (Condition 5); the judge's signature in notifications makes sure notifications can only result from executions of protocol Resolve.

State correctness also holds for dSEAC. Condition 1 holds, because time stamps in lco increase with every new lco tuple created, and dictionary digest's version numbers cannot decrease. Our Search operation ensures that a user performing search, after search, holds the lco and dul tuples as her user state that resulted from the most recently performed Update operation, and that lco and dul tuples are the state of the user that performed the Update operation, so Condition 2 of the state correctness definition also holds.

Efficiency. Due to dSEAC's reliance on SEAC, dSEAC inherits SEAC's sensitivity towards the inner structure of document collections. As in SEAC, the size of memory array A, for each internal SEAC instance, depends on the number of keyword–read policy pairs. Similarly, the number of ABE encryptions during generation of A depends on the number of keyword–read policy pairs. In contrast to SEAC, the number of read policies associated with a searched keyword not only influences the number of ABE decryptions, but, due to our verification mechanism for search results, also the size of search results. Particularly, even an empty search result can have arbitrary size, for example, if the searched keyword is associated with many policies, but the user on whose behalf search is performed does not hold access rights that satisfy any of the policies.

```
$\underline{\mathsf{Update}(pp,usk,st_{\text{user}},DC|pp,st_{\text{server}})}$
U&S: engage in protocol $(o|\bot)\leftarrow\mathsf{LCO\text{-}Update}(pp,st_{\text{user}}|pp,st_{\text{server}})$
U: if $o=0$: raise alarm and abort protocol
U: if $o=1$: abort protocol
U: $st_{\text{user}}\leftarrow o.\mathtt{lco}$ // $o=(\mathtt{lco},\mathtt{dul})$
U: $T\leftarrow\{d:(d,v,r,w,o,k)\in DC\}$ // compute management and data pairs
U: send $T$ to server
S: determine highest version numbers for data and management pairs for document
   identifiers from $T$
S: send version numbers and proofs (from AAD.Lookup) of maximality to user; this
   includes data and management pairs
U: verify that version numbers for all document identifiers in $T$ have been provided, and
   that the version numbers are maximal
U: determine what management and data pairs need to be and can be created given
   user's access rights
U: if a pair needs to be created, but the user is unable to: abort protocol
U: let $MP$ be the set of necessary, newly computed management pairs
U: let $DP$ be the set of necessary, newly computed data pairs
U: compute HT entries (as $H$) and memory arrays A and D as in SEAC's Update operation,
   but make entries in HT and A time dependent, lists in A and D authenticated, and add
   the additional layer of encryption to HT entries
U: $I\leftarrow(H,\mathtt{A},\mathtt{D},(MP,DP))$
U: $\sigma_I\leftarrow\Sigma.\mathsf{Sign}(u\varsigma k,I)$
U: send $I$ to server
U: verify that dictionary entries in $I$ share the same origin ($\Sigma$.VOrig), and that the user
   hold the access rights necessary to create new document version; if verification fails:
   send "invalid operation" to user and abort protocol
S: AAD.Append dictionary entries from $I$ to respective dictionaries
S: send new dictionary digests $dig$ and append-only proofs to user
U: verify received proofs for HT,Man,Dat and that UL's digest in $dig$ matches UL's digest
   in dul (from LCO-Update); if verification fails: raise alarm and abort protocol
U: $\sigma^*\leftarrow\Sigma.\mathsf{Sign}(u\varsigma k,\langle st_{\text{user}}.t+1,dig,\sigma_I,crt\rangle)$
U: $\mathtt{lco}\leftarrow(st_{\text{user}}.t+1,dig,\sigma_I,crt,\sigma^*)$
U: send lco to server
S: verify lco, by verifying signatures and checking that data from lco matches its
   expected value; if verification fails: send "invalid operation" to user, roll back changes
   made to dictionaries, and abort protocol
S: add A and D from $I$ to $ASet$ and $DSet$, respectively
S&U: $(s|u)\leftarrow\mathsf{LCO\text{-}Registration}(pp,st_{\text{server}},\mathtt{lco}|pp,st_{\text{user}},,\mathtt{lco})$
S: output $st_{\text{server}}=s$ to user
U: if $u=0$: raise alarm and abort protocol
U: output $st_{\text{user}}=u$ to user
```

Figure 6.16: Protocol Update of the dSEAC scheme

```
$\underline{\mathsf{Search}(pp,usk,st_{\text{user}},kw|pp,st_{\text{server}})}$
U&S: engage in protocol $(o|\bot)\leftarrow\mathsf{LCO\text{-}Update}(pp,st_{\text{user}}|pp,st_{\text{server}})$
U: if $o=0$: raise alarm and abort protocol
U: if $o=1$: output $\bot$ and abort protocol
U: $st_{\text{user}}\leftarrow o$ // $o=(\texttt{lco},\texttt{dul})$
U: $ldk\leftarrow\mathsf{PRF.Eval}(k_L,kw)$
U: $kdk\leftarrow\mathsf{PRF.Eval}(k_K,kw)$
U: $seed\leftarrow\mathsf{PRF.Eval}(k_S,kw)$
U: for $i=1$ to $st_{\text{user}}.t$: $seed\leftarrow\mathsf{TDP.Inv}(sk_T,seed)$
U: let $usk_{\text{restricted}}$ be the restriction of $usk$ to attributes from namespace Usr, and
   additionally the attributes "Op:search", "Wrd:$\mathsf{PRF.Eval}(k_B,kw)$", and "Tme:$st_{\text{user}}.t$"
U: $\sigma_Q\leftarrow\Sigma.\mathsf{Sign}(u\varsigma k,\langle ldk,kdk,seed,u,crt,st_{\text{user}}\rangle)$
U: $Q\leftarrow(ldk,kdk,seed,u,crt,\sigma_Q)$
U: send $Q$ to server
S: if $\Sigma.\mathsf{Verify}(crt.uvk,\langle ldk,kdk,seed,u,crt,\texttt{lco}\rangle,\sigma_Q)\neq 1$: send "malformed query" to user
   and abort protocol
S: $rslt\leftarrow\emptyset$
S: for $i=\texttt{lco}.t$ down to 1:
   • $q\leftarrow(\mathsf{PRF.Eval}(ldk,seed),\mathsf{PRF.Eval}(kdk,seed,u)$
   • perform search as in SEAC with query $q$ and result $X$, but include membership
     proofs, list authenticators and authenticated data in $X$; if a notification for the SEAC
     instance from time step $i$ exists in Ntf, let $X$ be that notification; if an error occurs
     during search: send "malformed data detected" to user, abort the protocol, and
     engage in conflict resolution
   • $rslt\leftarrow X$
   • $seed\leftarrow\mathsf{TDP.Eval}(pk_T,seed)$
S: send $rslt$ to user
U: if received "malformed query" or "malformed data detected:" output $\bot$ and abort
   protocol
U: recompute the server's (internal) SEAC queries and check that $rslt$ contains (non-
   )membership proofs for all of these queries, or a notification for the query's time steps
U: verify all notifications
U: verify list authenticators of A lists from all membership proofs ($\Sigma.\mathsf{VSig}$)
U: verify list authenticators of D list from all accessed A lists ($\Sigma.\mathsf{VSig}$ for accessed D lists,
   $\Sigma.\mathsf{VOrig}$ for D lists not accessed)
U: verify that all D lists for access polices satisfied by the user's attributes are present in $rslt$
U: verify all management and data pairs from $rslt$ and that their creators were allowed
   to create them
U: if any of the above checks fails: raise alarm and abort protocol
U: decrypt all document ciphertexts from data pairs from $rslt$
U: output the set of resulting plaintexts and $st_{\text{user}}$ to user
```

Figure 6.17: Protocol Search of the dSEAC scheme

```
Resolve($pp, st_{\text{server}} | pp, jvk, st_{\text{judge}}$)
S: send "conflict resolution" to judge
J&S: $(o|\bot) \leftarrow$ LCO-Update($pp, st_{\text{judge}} | pp, st_{\text{server}}$)
S: let $Q = (ldk, kdk, seed, u, crt, \sigma_Q)$ be the user's query that led to error detection
S: let $t'$ be the time stamp of the flawed SEAC instance
S: let $I_{t'}$ be the flawed SEAC instance
S: $r \leftarrow$ (AAD.Lookup($pp_{\texttt{UL}}$,UL,$t'$),AAD.Lookup($pp_{\texttt{UL}}$,UL,lco.$t$))
S: determine dictionaries HT$'$, Man$'$, Dat$'$, UL$'$ for time stamp $t'$ and dictionary HT$^*$ for
   time stamp lco.$t$
S: send $(r, Q, t', I_{t'}$, UL[$t'$], (AAD.PAppend($pp_{\texttt{HT}}$, HT$'$, HT), AAD.PAppend($pp_{\texttt{Man}}$, Man$'$, Man),
   AAD.PAppend($pp_{\texttt{Dat}}$,Dat,Dat)))
J: verify $t'$ is the time stamp occurring in UL[$t'$]
J: verify received AAD proofs relative to dictionary digests from UL[$t'$]
J: verify UL[$t'$] and UL[$t'$].$ch$ is a signature on $I_{t'}$
J: verify $\sigma_Q$ is a signature on $Q$ and lco
J: if any of the above checks fails: raise alarm and abort protocol
J: compute the SEAC search query $q$ for time step $t'$
J: perform search as in SEAC on instance $I_t'$ with query $q$; if no error occurs during search:
   raise alarm and abort protocol
J: $\sigma_{ntf} = (\sigma'_{ntf}, d_{ntf}) \leftarrow \Sigma$.Sign($jsk, \langle$lco$, t'\rangle$)
J: $ntf \leftarrow$ (lco,$t'$,$\sigma_{ntf}$)
J: $H \leftarrow \{$(PRF.Eval(PRF.Eval($k_L, kw$),TDP.Inv$^{\texttt{lco}.t+1}$(PRF.Eval($k_S, kw$))),$\sigma'_{ntf}$)$\}_{kw in words}$
J: send $H$ to server
S: AAD.Append dictionary entries from $H$ to HT
S: send new dictionary digest for HT and corresponding append-only proof to judge
J: if verification of append-only proof fails: raise alarm and abort protocol
J: replace UL's digest in lco by dul.digest($UL$)
J: replace HT's digest in lco by the received HT digest
J: $\sigma \leftarrow \Sigma$.Sign($jsk, \langle$lco.$t+1$,lco.$dig, \sigma_{ntf}, jvk\rangle$)
J: lco$' \leftarrow$ (lco.$t+1$,lco.$dig, \sigma_{ntf}, jvk, \sigma$)
J: send ($ntf$,lco$'$) to server
J: publish $ntf$
S: add $ntf$ to Ntf
S&J: $(s|u) \leftarrow$ LCO-Registration($pp, st_{\text{server}}$,lco$'$|$pp, st_{\text{judge}}$,lco$'$)
S: output $st_{\text{server}} = s$ to server
J: if $u = 0$: raise alarm and abort protocol
J: output $st_{\text{judge}} = u$ to judge

Version($pp, st_{\text{user}}$)
• if $st_{\text{user}} = \bot$: output $(-1, -1, -1, -1, -1)$
• otherwise: output ($st_{\text{user}}$.lco.$t$,AAD.Version($st_{\text{user}}$.lco.$dig$.digest(HT)),AAD.Version($st_{\text{us}}$
  AAD.Version($st_{\text{user}}$.lco.$dig$.digest(Dat)),AAD.Version($st_{\text{user}}$.lco.$dig$.digest(UL)))
```

Figure 6.18: Protocol Resolve and algorithm Version of the dSEAC scheme

Leakage. Due to dSEAC allowing users to update the searchable document collection and the possibility of many updates, we have to consider that the adversary may attempt to fork the system. The system is forked if two users perform updates on the same system state. dSEAC is fork consistent, see below, but fork consistency prevents forks from being merged, not from being created. Therefore, we have to consider forks when analyzing dSEAC's leakage. Particularly, even though honest users only operate on a single fork, their actions may impact all forks. For example, a search query may be applied to all forks, which is possible because HT labels and keys are deterministically computed from keywords. Therefore, the following leakage functions occasionally mention the fork that the leakage occurs on.

Apart from forks, the dSEAC scheme's leakage is comparable to the leakage of the SEAC scheme, though distributed over time for the various SEAC instances used internally by dSEAC. Note however, that in dSEAC some of the leaked data takes a different form than in SEAC. For example, the user certificate in dSEAC contains a signature verification key and an attribute set, rather than a meaningless random number. As a result, whenever a user certificate *crt* occurs in leakage, we can infer the corresponding attribute set. As in SEAC, keywords are associated with identifiers which can be inferred from keywords, but keywords cannot be inferred from keyword identifiers; this allows us to distinguish keywords that occur in the same documents.

Also as in SEAC, for our analysis of dSEAC's leakage, we assume PRF to be a pseudorandom function, Sym to be an eavesdropping secure encryption scheme, and ABE to be a CPA-secure ABE scheme. We additionally assume TDP to be a trapdoor permutation.

The leakage incurred by enrolling and corrupting users remains conceptually the same as in SEAC, but in the case of corruption reveals details of all SEAC instances, rather than a single one. As a result, the leakage from enrolling an honest user is still only the user's certificate (now including the user's attributes) as published by the key issuer during the UserJoin operation. Therefore, $\mathcal{L}_{\text{HonestJoin}}(U) = (crt, U)$

When a dishonest user is enrolled, the leakage consists of the new user's certificate *crt*, including the user's attribute set U, as well as (plaintext) documents and their identifiers, version numbers, access policies, and the system forks these documents appear on. Document plaintexts are included in the leakage, because these documents are accessible due to the new user's access rights. We denote the document collection from the Update operation at time t on fork f as $DC_{f,t}$ and denote the set of documents from $DC_{f,t}$ accessible to the newly enrolled user as

$$DC_{f,t}(U) = \{(f, t, d, v, \mathbb{A}_{\text{read}}, k) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{f,t} \wedge U \text{ satisfies } \mathbb{A}_{\text{read}}\}.$$

The leakage also contains all keyword–policy pairs present in the searchable document collection $DC_{f,t}$ for all system forks and time steps prior to enrollment. This information can be extracted from the various memory arrays A of SEAC instances, which are accessible from the dictionary HT using the new user's key and the user's (implicitly assumed) knowledge of the set *words* of possible keywords. We denote this part of the leakage as

$$KP(DC_{f,t}) = \{(f, t, kw, \mathbb{A}_{\text{read}}) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{f,t} \wedge kw \in k\}.$$

All in all,

$$\mathcal{L}_{\text{CorruptJoin}}() = (crt, U, \forall(f,t) : DC_{f,t}(U), \forall(f,t) : KP(DC_{f,t})\}.$$

As in SEAC, when corrupting a once honest user identified by certificate *crt*, the leakage is as if that user were enrolled as a corrupt user, so

$$\mathcal{L}_{\text{Corruption}}(crt) = (crt, U, \forall(f,t) : DC_{f,t}(U), \forall(f,t) : KP(DC_{f,t})\}.$$

In contrast to SEAC, server initialization in dSEAC depends on the secret key of the user that triggered the initialization operation. Particularly, this dependence comes in the form of the user certificate *crt* contained in the `lco` tuple created during the operation. The certificate also contains the attribute set U of the user identified by *crt* Therefore, $\mathcal{L}_{\text{Init}}(crt) = (crt, U)$.

As a consequence of users being able to perform updates, for update leakage, we need to distinguish whether or not a corrupt user exists in the system at the time of the respective Update operation, and what attributes the corrupt user holds. The existence of a corrupt user is denoted as $\exists$ corrupt U below, and means that one of the corrupt users holds attribute set U. We also need to consider whether an honest user's Update for the same time step, but different system fork, has occurred, because then `HT` labels and keys occur on both forks, specifically for keywords shared by the respective document collections.

Let DC_{new} denote the new document collection. In dSEAC, every update operation leaks the certificate *crt* and attribute set U of the user who performs the operation. The operation also leaks the number of new `HT` entries, sizes of memory arrays `A` and `D` associated with the new SEAC instance. These numbers are given by

$$|\mathtt{HT}_{\text{new}}| = |\{kw : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{\text{new}} \land kw \in k\},$$

$$|\mathtt{A}| = [\{(kw, \mathbb{A}_{\text{read}}) : (d, v, r, w, o, k) \in DC_{\text{new}} \land kw \in k \land (d', v', \mathbb{A}_{\text{read}}, w', o', k') \in DC_{\text{new}}\},$$

and

$$|\mathtt{D}| = \frac{\sum_{(d,v,r,w,o,k)\in DC_{\text{new}}} \sum_{kw\in k} |kw|}{\min\{|kw| : (d, v, r, w, o, k) \in DC_{\text{new}} \land kw \in k\}}.$$

An Update operation also leaks the identifiers, version numbers, access policies, and sizes of new and modified documents. This information can be extracted from new management and data tuples. We denote this information as

$$\mathtt{CT}(DC_{\text{new}}) = \{(d, v, \mathbb{A}_{\text{read}}, \mathbb{A}_{\text{write}}, \mathbb{A}_{\text{own}}, |k|) : (d, v, \mathbb{A}_{\text{read}}, \mathbb{A}_{\text{write}}, \mathbb{A}_{\text{own}}, k) \in DC_{\text{new}}\}.$$

If there is an Update operation for the same time step t, but on a different fork f, and the other Update operation used document collection $DC_{f,t}$, then the leakage also contains the identifiers $id(kw, t)$ of all keywords that $DC_{f,t}$ and DC_{new} have in common. Specifically, the dictionaries `HT` computed from the document collections share keyword and time dependent labels and keys. We denote these identifiers as

$$KW(DC_{\text{new}}, DC_{f,t}) = \left\{\begin{array}{r} id(kw,t) : (d, v, r, w, o, k) \in DC_{\text{new}} \\ \land (d', v', r', w', o', k') \in DC_{f,t} \\ \land kw \in k \cap k' \end{array}\right\}.$$

If there is a corrupt user, the leakage also includes plaintexts of documents accessible by the corrupt user, so the leakage is as if the user was corrupted after the update operation occurred. This additional leakage is denoted

$$DC_{\text{new}}(U') = \{(d, v, \mathbb{A}_{\text{read}}, k) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{\text{new}} \land U' \text{ satisfies } \mathbb{A}_{\text{read}}\}$$

and

$$KP(DC_{\text{new}}) = \left\{\begin{array}{r} (kw, \mathbb{A}_{\text{read}}) : (d, v, r, w, o, k) \in DC_{\text{new}} \land kw \in k \\ \land (d', v', \mathbb{A}_{\text{read}}, o', k') \in DC_{\text{new}} \end{array}\right\}.$$

As a result,

$$\mathcal{L}_{\text{Update}}(crt, DC_{\text{new}}) = \left(\begin{array}{r} crt, U, |HT_{\text{new}}|, |\mathtt{A}|, |\mathtt{D}|, \mathtt{CT}(DC_{\text{new}}), KW(DC_{\text{new}}, DC_{f,t}), \\ \exists \text{ corrupt } U' : KP(DC_{\text{new}}), \forall \text{ corrupt } U' : DC_{\text{new}}(U') \end{array}\right).$$

Honest users' search requests for some keyword kw leak the certificate crt and attribute set U of the user on whose behalf search is performed. The request also leaks identifiers $id(kw, t)$ of the searched keyword for every time step $t \le t_{crt}$ so far on the fork of the user identified by crt. The read policies that occur in conjunction with kw in the document collections $DC_{f,t}$ are leaked from accessed `A` lists of the various system forks f for SEAC instance t, denoted

$$\mathbb{A}_{\text{read}}(DC_{f,t}, kw) = \{\mathbb{A}_{\text{read}} : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{f,t} \wedge kw \in k\}.$$

Accessed `D` lists of all system forks f and for all SEAC instances t leak the identifiers and version numbers of documents that contain kw and are accessible to the user on whose behalf search is performed. We denote this as

$$doc(DC_{f,t}, U, kw) = \{(d, v) : (d, v, \mathbb{A}_{\text{read}}, w, o, k) \in DC_{f,t} \wedge kw \in k \wedge U \text{ satisfies } \mathbb{A}_{\text{read}}\}.$$

Given document identifiers, version numbers, and forks, the corresponding documents' access policies and potentially contents can be inferred from other (previous) leakage. Therefore,

$$\mathcal{L}_{\text{Search}}(crt, kw) = \begin{pmatrix} crt, U, id(kw, t), \forall (f, t)_{t \le t_{crt}} : \mathbb{A}_{\text{read}}(DC_{f,t}, kw), \\ \forall (f, t)_{t \le t_{crt}} : doc(DC_{f,t}, U, kw) \end{pmatrix}.$$

When the resolve protocol is executed, no information is leaked to the server, so $\mathcal{L}_{\text{Resolve}}() = \perp$.

Security. We now turn to dSEAC's security properties. Particularly, we prove dSEAC's data confidentiality, fork consistency, verifiability, and forward privacy.

Lemma 37. *If* PRF *is modelled as a random oracle,* TDP *is a trapdoor permutation,* Sym *is an eavesdropping-secure symmetric encryption scheme,* ABE *is a CPA-secure non-committing (hybrid) ABE scheme, and* Σ *is a secure non-committing signature scheme with delayed verifiability,* dSEAC *provides data confidentiality relative to leakage functions* ($\mathcal{L}_{HonestJoin}$, $\mathcal{L}_{CorruptJoin}$, $\mathcal{L}_{Corruption}$, $\mathcal{L}_{Init}$, $\mathcal{L}_{Update}$, $\mathcal{L}_{Search}$, $\mathcal{L}_{Resolve}$) *in the random oracle model.*

Proof. As in the proof of SEAC's data confidentiality (Lemma 33), this lemma's prerequisites ensure that dSEAC's leakage is as discussed above. Equally, as in the proof of Lemma 33, non-committing encryption and patching oracles play significant roles in proving dSEAC's data confidentiality. Additionally, we rely on the random oracle for our signatures with delayed verifiability to be non-committing. Our simulator for the data confidentiality experiment can be found in Figures 6.19 and 6.20.

Given the simulator's ability to patch its random oracles, the simulator is able to create data structures in experiment $\mathbf{Exp}^{\text{SE-conf-sim}}_{\mathcal{A},\mathcal{S},\mathsf{dSEAC}}$ that the adversary cannot distinguish from the respective data structures in experiment $\mathbf{Exp}^{\text{SE-conf-real}}_{\mathcal{A},\mathsf{dSEAC}}$. For that, we make three key observations.

First, list authenticators can be computed from dummy data and later on, by patching a random oracle, the authenticator can be turned into a signature on any given list. Therefore, list authenticators are independent of the lists that they supposedly are signatures on; dependence is only established once the dependence is registered with the random oracle by patching the oracle. The dependence is only registered with the oracle once the authenticated data actually becomes known. This is the same reasoning we applied to SEAC, and that applies here as well for the indistinguishability of ciphertexts: the

$\underline{\mathcal{O}_{\text{Setup}}(1^\lambda)}$

- $(pp,msk,jsk,st_{\text{judge}}) \leftarrow \mathsf{Setup}(1^\lambda)$
- remember $(msk,jsk,st_{\text{judge}})$
- output pp

$\underline{\mathcal{O}_{\text{HonestJoin}}(\mathcal{L}_{\text{HonestJoin}}(U))}$

- execute protocol $(usk,st_{\text{user}}|crt) \leftarrow \mathsf{UserJoin}(pp,U|pp,msk)$; remember $(crt,usk,st_{\text{user}})$

$\underline{\mathcal{O}_{\text{CorruptJoin}}(\mathcal{L}_{\text{CorruptJoin}}())}$

- if crt from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following
- based on received leakage $\{(f,d,v,\mathbb{A}_{\text{read}},k)\}$ that has not occurred before:
  - patch random oracle for non-committing ABE, so `Dat` entry for (d,v) with policy $\mathbb{A}_{\text{read}}$ contains plaintext document k
  - compute non-encrypted linked `D` lists as in protocol Update; patch random oracle for non-committing symmetric encryption in order to embed the lists in array `D` for fork f and time steps in which (d,v) from leakage were added to DC
- based on received leakage $\{(f,t,kw,\mathbb{A}_{\text{read}})\}$ that has not occurred before:
  - compute non-encrypted linked `A` as in protocol Update, except the ABE ciphertexts inside the list nodes are dummy ciphertexts, unless a `D` list for $(kw,t,\mathbb{A}_{\text{read}})$ has previously been computed, in which case the ABE ciphertext is an encryption of the pointer to the `D` list (for the fork and time step from the leakage)
  - patch random oracles for PRFs so `HT` entry for keyword kw points to `A` list for kw and the fork and time step from the leakage
- if any of the above oracle patching fails: output $\perp$ and abort the simulation
- execute $\mathsf{UserJoin}$ with key issuer's input $(pp,msk$; interact with $\mathcal{A}$, which plays the user; remember the published user certificate crt and mark it as corrupt

$\underline{\mathcal{O}_{\text{Corruption}}(\mathcal{L}_{\text{Corruption}}(crt))}$

- if crt from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following; otherwise: a tuple $(crt,usk,st_{\text{user}})$ has been remembered
- compute data and patch oracles as in $\mathcal{O}_{\text{CorruptJoin}}$
- mark crt as corrupt
- output (usk,st_{user}) from remembered tuple to $\mathcal{A}$

$\underline{\mathcal{O}_{\text{Init}}(\mathcal{L}_{\text{Init}}(crt))}$

- if crt from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following; otherwise: a tuple $(crt,usk,st_{\text{user}})$ has been remembered
- execute protocol Init with user's input $(pp, usk, st_{\text{user}})$; interact with $\mathcal{A}$; remember updated user state

Figure 6.19: Behavior of the simulator $\mathcal{S}$ for dSEAC's data confidentiality game against adversary $\mathcal{A}$: Reactions to setup, enrollment, corruption and initialization leakage

$\mathcal{O}_{\text{Update}}(\mathcal{L}_{\text{Update}}(crt,DC))$

- if crt from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following; otherwise: a tuple $(crt,usk,st_{\text{user}})$ has been remembered
- execute protocol Update with user's input $(pp,usk,st_{\text{user}})$, but without a document collection; instead, compute data from received leakage on set sizes and document properties as follows:
  - compute `Man` entries as in the protocol
  - compute `Dat` entries as in the protocol, using plaintexts from the leakage if available, and dummy document plaintexts if real plaintexts are not available.
  - compute entries for dictionary `HT` containing dummy data (labels and values chosen from appropriate domains); respect occurrence of keyword in multiple forks; the number of entries is indicated by the leakage
  - compute memory arrays `A` and `D` containing dummy cells; the number of entries is indicated by the leakage
  - if any of the implicit oracle patching above fails: output $\perp$ and abort the simulation
  - remember updated user state

$\mathcal{O}_{\text{Search}}(\mathcal{L}_{\text{Search}}(crt,kw))$

- if crt from leakage has not been remembered or is marked corrupt: output $\perp$ to $\mathcal{A}$ and do nothing of the following; otherwise: a tuple $(crt,usk,st_{\text{user}})$ has been remembered
- based on received leakage $\{(f,d,v)\}$ that has not occurred before (for the identifier of kw from the leakage):
  - compute non-encrypted linked `D` lists as in protocol Update (for all relevant forks and time steps); patch random oracle for non-committing symmetric encryption in order to embed the lists in arrays `D` (for the relevant forks and time steps)
- based on received leakage $\{(f,t,\mathbb{A}_{\text{read}})\}$ that has not occurred before:
  - compute non-encrypted linked `A` list as in protocol Update, except ABE ciphertexts inside the list nodes are dummy ciphertexts, unless a `D` list for $(kw,t,\mathbb{A}_{\text{read}})$ has previously been computed, in which case the ABE ciphertext is an encryption of the pointer to the `D` list for fork f and time step t
  - patch random oracles for PRFs so `HT` entries for keyword kw in fork f point to `A` lists for kw
- if any of the above oracle patching fails: output $\perp$ and abort the simulation
- execute protocol Search with user's input $(pp,usk,st_{\text{user}})$, but with dummy keyword; internally, use PRF images to construct the query that recover lists given in the leakage (the lists have been created before); remember updated user state

$\mathcal{O}_{\text{Resolve}}(\mathcal{L}_{\text{Resolve}}())$

- execute protocol Resolve planing the judge with input $(pp,jsk,st_{\text{judge}})$; remember updated st_{judge}

Figure 6.20: Behavior of the simulator $\mathcal{S}$ for dSEAC's data confidentiality game against adversary $\mathcal{A}$: Reactions to update, search and resolution leakage

adversary lacks information to make meaningful queries to the oracles, and once the adversary requests the information necessary for making meaningful queries, the oracles are patched so ciphertexts contain the expected plaintexts and signatures authenticate the expected data. Unfortunately, this line of reasoning assumes that the system has not been forked, because relations between forks may cause oracle patching to fail, and that patching oracles rarely fails due to other reasons.

Second, for all uses of random oracles apart from uses in the creation of HT entries, inputs to the oracle include a random bit string, for example, the randomness of list authenticators' open values and symmetric keys used only once. This ensures that patching oracles for symmetric and hybrid encryption, and signatures with delayed verifiability will fail only with negligible probability, independent of system forks.

Third, the additional layer of encryption that we have introduced for the values in HT entries ensures that, even if two system forks have two keywords in common for any time step t, the entries' pointers to A lists and the lists' authenticator's open values are symmetrically encrypted under independent keys. Although the independent keys are masked by the same value, and thus, the difference between these keys is known to the adversary, there are still exponentially many pairs of keys (in the security parameter) that have that distance. Then, with non-committing encryption in the random oracle model, the values encrypted under these keys — list pointers and authenticators' open values — remain unpredictable. As a consequence, even though all forks rely on the same random oracles, forks can not be used reliably to force oracle patching to fail.

Due to Observations 2 and 3, restrictions on Observation 1 are not applicable. Hence, we conclude that simulating the data structures fails only with negligible probability, namely, when an oracle patch fails. We further conclude that if simulating the data structures does not fail, the simulated data structures in experiment $\mathbf{Exp}^{\text{SE-conf-sim}}_{\mathcal{A},\mathcal{S},\mathsf{dSEAC}}$ are indistinguishable from the data structures in experiment $\mathbf{Exp}^{\text{SE-conf-real}}_{\mathcal{A},\mathsf{dSEAC}}$. Furthermore, if none of the oracle patches fail, the input and output behavior of our simulator in experiment $\mathbf{Exp}^{\text{SE-conf-sim}}$ is the same as the behavior of the oracles in experiment $\mathbf{Exp}^{\text{SE-conf-real}}$, so the adversary $\mathcal{A}$ can distinguish between the two experiments only with probability negligible in the security parameter. ☐

Lemma 38. *If* dSEAC *is instantiated with a secure signature scheme with delayed verifiability and a secure AAD,* dSEAC *is fork consistent.*

Proof. Assume to the contrary of the lemma that an adversary $\mathcal{A}$ makes experiment $\mathbf{Exp}^{\text{SE-fc}}_{\mathcal{A},\mathsf{dSEAC}}$ output 1. This requires the adversary to cause two honest users 0 and 1 to adopt two distinct user states st'_0 and st'_1 for the same version, and later on make both users adopt the same state st^*. Note that the operations that cause a user to adopt a state are not necessarily identical to the operations that the adversary from the fork consistency experiment points out. We now consider the operations of users 0 and 1 that led to them adopting st^*.

First, we consider the case of at least one of these operations being an Init operation. Without loss of generality, the Init operation was performed by user 0. Then we know that user 0 has not held a state previously, thus $st'_0 = \bot$; otherwise, user 0 had aborted the Init operation and $st'_0 = st^*$, which would violate condition $\mathsf{Version}(st'_0) < \mathsf{Version}(st^*)$ from the fork consistency experiment and the experiment would not have output 1. The fact that the experiment has output 1 also implies that $\mathsf{Version}(st'_1) = \mathsf{Version}(st'_0) = (-1,-1,-1,-1,-1)$. The tests performed by operation LCO-Update during Update and Search operations imply that the first component of the version vector of any state adopted due to operation LCO-Update is at least 0; otherwise, such a state is not adopted. Therefore, user 1 cannot hold state st^* as a result of an Update or Search operation. The user also cannot hold state st^* as a result of a successful Init operation, because a user that holds a state immediately aborts the Init operation, meaning that, for user 1 to not abort the Init operation $st'_1 = \perp$. Then, however, $st'_0 = st'_1$ and the experiment would not have output 1. This only leaves an unsuccessful Init operation as the cause for user 1 to hold state st^*, meaning that the Init operation has not caused the user to change her state, so st^* has already been held before the operation, meaning $st^* = st'_1$, and thus, $\mathsf{Version}(st'_1) = \mathsf{Version}(st^*)$, so the fork consistency experiment would not have output 1.

Since none of the operations that caused the honest users to adopt state st^* is an Init operation, we now focus on the case of at least one of the operations being an Update operation. Without loss of generality, assume that user 0 adopted state st^* as a result of an Update operation. We need to consider three sub-scenarios, namely, whether the Update operation changed the user's state at all, which depends on the outcome of the operation's call to $\mathsf{LCO\text{-}Update}$, and whether or not the user's update was accepted or rejected by the server. If operation $\mathsf{LCO\text{-}Update}$ has returned 0 or 1, user 0 did not change her state, and thus, has held st^* before the operation. This implies $st^* = st'_0$, which would have caused the fork consistency experiment to output 0. So, the $\mathsf{LCO\text{-}Update}$ operation returned `lco` and `dul` tuples. If the server had rejected user 0's Update operation or had not sent a `dul` value for the post-update `lco`, the user again did not change her state, so $st^* = st'_0$ and the experiment would have output 0. Therefore, for user 0 to adopt state $st^* \neq st'_0$ during an Update operation, the last completed operation that st^* represents was performed by user 0, and we need to consider whether user 1 adopted st^* during an Update or Search operation.

The above reasoning also applies to user 1's adoption of state st^* during an Update operation, and we conclude that, for the fork consistency experiment to output 1, the user's operation must have been successful. This requires that the operation has not been rejected by the server and that the server has sent a `dul` tuple for the operation's `lco` tuple. The structure of user states then implies that, for both user's to hold st^* after their respective Update operations, the dictionary digests inside the states must be identical, the user certificates must be identical, and both user's signatures on their respective states must be identical. User certificates being identical requires users 0 and 1 to have identical signature verification keys and requires both certificates to contain the same signature under the key issuers' signing key. Users' or key issuer's signatures are secure signatures with delayed verifiability. Such signatures being identical is unlikely as a consequence of these signatures satisfying the hiding property, which requires the signing process to be probabilistic. Note that, above, an event is called unlikely if it happens with probability negligible in the security parameter.

So, except with negligible probability, user 1 adopted state st^* during a Search operation. As with Update operations, the Search operation's internal call to operation $\mathsf{LCO\text{-}Update}$ must have returned tuples `lco` and `dul`, because otherwise, as before, the fork consistency experiment would not have output 1. However, both users have seen sequences of append-only proofs for the digests of dictionary `UL`, linking the respective digests from the `lco` components of states st'_0 and st'_1 to the `UL` digest from st^*'s `dul`. Such a sequence is a violation of `UL`'s AAD fork consistency, and thus the probability for the server to output such a sequence is negligible in the security parameter. However, the server outputting such a sequence is necessary for the fork consistency experiment to output 1.

The same reasoning applies if both users 0 and 1 have adopted state st^* during a Search operation. All in all, every scenario that results in the fork consistency experiment outputting 1 occurs with probability negligible in the security parameter. □

Lemma 39. *If* dSEAC *is instantiated with a secure signature scheme with delayed verifiability and a secure AAD,* dSEAC *is verifiable.*

Proof. Assume to the contrary of the lemma that an adversary $\mathcal{A}$ makes experiment $\mathbf{Exp}^{\text{SE-vrfy}}_{\mathcal{A},\mathsf{dSEAC}}(\lambda)$ output 1, meaning that the adversary makes two honest users holding identical attribute sets accept two distinct search results for the same keyword, despite the user verifying their respective search results relative to identical states

Since the experiment outputs 1, we know that neither search result has been rejected. Therefore, we have $X_0 \neq \perp \neq X_1$. Then, because both users hold identical states, both users have obtained partial search results for each time step, as indicated by their state. Each partial result either consists of a SEAC search result together with an AAD membership proof and list authenticators, or is a notification. Particularly, twice the number of notifications plus the number of SEAC search results must be the number of time steps from the users' states. Notifications count twice, once for the SEAC instance they replace, and once for the time steps during which they were issued.

If the raw search results (that the users decrypt to X_0 and X_1) differed in the number of notifications they contained, due to notifications counting twice, one of the raw results needed to contain two SEAC search results that are replaced by a notification in the other raw result. Particularly, there must be a SEAC instance for the time stamp from the `lco` tuple that registers the notification with the update log. A partial search result for that SEAC instance gets presented to the user that is not presented with the notification. Most notably, for that SEAC instance, there is a membership proof for dictionary `HT` and the keyword *kw*. However, since dSEAC registers the notification with dictionary `HT` *for every keyword*, either the membership proof presented to the user contains a reference to the notification, or it does not. In the former case, the honest user would have rejected the search result and output $\perp$, resulting in the verifiability experiment outputting 0. In the latter case, the adversary would have broken `HT`'s AAD membership security, which only happens with probability negligible in security parameter λ. The same reasoning applies to the case of the raw search results containing differing sets of notifications.

The reasoning also applies to adversaries presenting the users with differing results to the AAD.Lookup queries that are made as part of the internal SEAC search queries. Hence, for each of the internal SEAC search queries, the adversary reports the same status (membership or non-membership), and in the case of membership, the reported list authenticators (on `A` lists) are identical. Then however, for the verifiability experiment to output 1, both users must have verified their respective list authenticator for the SEAC instance successfully, which in turn means that both users' raw search results contain the same `A` list for the list authenticator. Otherwise, the list authenticator, which is a signature with delayed verifiability, is a signature on two distinct messages. Due to the signature scheme's unforgeability properties, the adversary can find such messages only with probability negligible in the security parameter.

The list authenticators (on `A` lists) are signatures on lists of pairs of access policies and list authenticators (on `D` lists). Since both users hold the same set of attributes and are presented with identical sets of access policies, they agree on which of the presented policies their attributes satisfy, and thus the users agree on what `D` lists they expect to see.

Hence, as a consequence of the users holding identical lists authenticators on `A` lists, the users also obtain identical policy–authenticator pairs in their raw search results. The same reasoning also applies to list authenticators on `D` lists and the document identifier–version number pairs from these `D` lists. This in turn means that both users are presented with results to identical internal AAD.Lookup queries to dictionaries `Man` and `Dat`. Then, due to AAD membership security, except with negligible probability, both users are presented with identical results to these queries.

Since the results to Dat queries contain the ABE ciphertexts that the users decrypt and output (as X_0 and X_1, respectively) and ABE decryption only fails with negligible probability, we have that, except with probability negligible in security parameter λ, if $X_0 \neq \perp \neq X_1$, then $X_0 = X_1$. Thus, dSEAC is verifiable. $\square$

Lemma 40. *If* PRF *is a pseudorandom function,* Sym *is an eavesdropping secure symmetric encryption scheme,* ABE *is a CPA-secure ABE scheme, and* Σ *is a secure signature scheme with delayed verifiability,* dSEAC *is forward private.*

Proof. As before, the lemma's prerequisites ensure that dSEAC's leakage functions are as discussed above. In the absence of forks and dishonest users, dSEAC's Update operation leaks no information on keywords beyond an upper bound on the number of distinct keywords affected by the operation, as well as information that can trivially be derived from the upper bound, for example, upper bounds on the number of keyword–policy pairs in the update and the average number of keywords per documents. Particularly, the actual keywords remain hidden.

The upper bounds can be inferred from the number of new HT entries due to the Update operation, as well as the sizes of data structures A and D sent to the server during the operation. All in all, dSEAC is forward private. $\square$

7 Leakage in Searchable Encryption with Access Control

Our SEAC scheme and its dynamic counterpart dSEAC may be categorized as L_1 schemes in Cash et al.'s leakage classification framework [Cas+15], c.f. Section 5.2. This categorization is due to the index structure we use; particularly, this classification is inherited from the CGKO scheme [Cur+11], c.f. Section 5.4.

However, the categorization of our SEAC and dSEAC schemes ignores our schemes' access control mechanisms and what the mechanisms reveal, particularly if we want search results to be verifiable. Ideally, whenever Alice searches for a keyword kw, Alice only learns the documents that contain kw and are accessible to Alice. However, verifiability generally requires Alice to be able to check that documents containing kw not presented to her as part of the search result are not accessible to her. Hence, Alice at least learns the access policies of such documents and, particularly, the fact that documents that contain kw exist despite Alice not being able to access them.

Our motivation for integrating access control with searchable encryption was to avoid such leakage to the greatest extent possible. One may ask whether the leakage that dSEAC incurs is necessary if the searchable encryption scheme with access control is to be verifiable. As we demonstrate in this chapter, the answer to this question is no. However, to get this answer, we restrict the type of access policies to hierarchical access levels, and a single data owner. This results in the reduced-leakage scheme RLS. The RLS scheme provides data confidentiality and is forward private, fork consistent, and verifiable.

The RLS scheme integrates dSEAC's `HT`, `A` and `D` data structure into a single data structure, so it is natural to ask whether a similar reduction in leakage can also be achieved by combining only some of dSEAC's data structures, particularly `HT` and `A`, albeit using the more general access control policies from ABE. With our dictionary-integrated access control scheme DIA, we explore one such combination. It turns out that the leakage of DIA is essentially the same as the leakage of dSEAC.

7.1 Reduced-Leakage Scheme: RLS

Our reduced-leakage searchable encryption scheme RLS provides dynamic searchable encryption in scenarios that feature a single data owner and many data users. Users and documents are assigned access levels, and users are allowed to access all documents whose access levels are less than or equal to the user's access level.

Our RLS scheme is an adaption of Bost's Σοφος scheme [Bos16], c.f. Section 5.5. In contrast to Σοφος, our RLS scheme also handles document ciphertexts. RLS does so by storing symmetrically encrypted documents in a dictionary CT, similar to CT's role in SEAC, but implementing the dictionary as an AAD for verifiability purposes. Also unlike Σοφος, RLS works in a multi-reader scenario. The RLS scheme is inspired by the searchable encryption scheme of Alderman et al. [AMR17], although RLS barely resembles their scheme: we mainly adopt their concept of hierarchical access levels, essentially imposing a total order on the set of users, but we use Bost's technique to realize the levels.

Construction 41. *The* RLS *scheme consists of algorithms and protocols* (Setup, UserJoin, Init, Update, Search, Resolve, Version) *shown in Figures 7.1–7.3.*

For notational purposes, we assume the key issuer to also be the single data owner. The data owner's user state is denoted as st_{issuer}. As before in dSEAC, PRF denotes a PRF, TDP is a trapdoor permutation, AAD denotes an AAD scheme, Σ is a signature scheme with delayed verifiability, and Sym is a symmetric encryption scheme.

The index structure that the RLS scheme maintains is conceptually identical to the one of Σοφος, c.f.Figure 5.3, but with two applications of Bost's technique, rather than one, for both the labels and the decryption keys. For the collection of ciphertexts stored in RLS, we employ Bost's technique to derive symmetric encryption keys for each of the access levels.

As can be seen from the figure, the RLS scheme also makes use of some of our extensions (to SEAC) presented in Section 6.3.1, in order to make RLS verifiable. Particularly, we employ the concept of last completed operations.

Properties of the RLS scheme. We now briefly discuss the properties of the RLS scheme. First, the scheme is *correct.* This property is inherited from Bost's Σοφος scheme.

The leakage involved in the RLS scheme is fairly low. This is mainly due to the fact that the scheme only supports a hierarchy of access levels, and, as a consequence, has simpler data structures. For honest user enrollment, only the new user's certificate *crt* is leaked, so $\mathcal{L}_{\text{HonestJoin}}(a) = crt$.

When a dishonest user joins the system, leakage consists of the user's certificate *crt* and access level a. For the document collection DC_t from the t-th Update operation, enrollment of a dishonest user with access level a leaks the document names and version numbers, access levels and contents of all documents accessible to the user. We denote this partial leakage as

$$DC_t(a) = \{(t, d, v, a', k) : (d, v, a', w, o, k) \in DC_t \wedge a \geq a'\}.$$

Thus,

$$\mathcal{L}_{\text{CorruptJoin}}() = (crt, a, \forall t : DC_t(a)).$$

As with SEAC and dSEAC, it does not matter whether a user joins the scheme as a corrupt user or gets corrupted at some point after joining. Therefore,

$$\mathcal{L}_{\text{Corruption}}(crt) = (crt, a, \forall t : DC_t(a)).$$

The Init operation in RLS does not leak any information. Hence, $\mathcal{L}_{\text{Init}}(\perp) = \perp$.

The update leakage in RLS consists of the document names and version numbers, access levels, and sizes of all documents in the new document collection DC_{new}, denoted as

$$\texttt{CT}(DC_{\text{new}}) = \{(d, v, a, |k|) : (d, v, a, w, o, k) \in DC_{\text{new}}\}.$$

During an update operation, the number of new HT entries can be extracted from the data sent by the data owner. The number $|\texttt{HT}_{\text{new}}|$ of new HT entries is given by

$$|\texttt{HT}_{\text{new}}| = |\{(kw, a) : (d, v, a', w, o, k) \in DC \wedge kw \in k \wedge a \leq a'\}|.$$

```
$\underline{\mathsf{Setup}(1^\lambda)}$
• $k_D \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $k_S \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$
• $(sk_A, pk_A) \leftarrow \mathsf{TDP.KeyGen}(1^\lambda)$
• $(sk_T, pk_T) \leftarrow \mathsf{TDP.KeyGen}(1^\lambda)$
• $(isk, ivk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
• $(pp_{\mathtt{HT}}, vk_{\mathtt{HT}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda)$
• $(pp_{\mathtt{CT}}, vk_{\mathtt{CT}}) \leftarrow \mathsf{AAD.Setup}(1^\lambda)$
• $t \leftarrow 0$
• $dig \leftarrow \emptyset$
• $\sigma \leftarrow \Sigma.\mathsf{Sign}(isk, \langle t, dig \rangle)$
• $st_{\text{issuer}} \leftarrow (t, dig, \sigma)$
• $s_a \leftarrow \mathsf{TDP.Sample}()$
• $msk \leftarrow (k_D, k_K, k_S, sk_A, sk_T, isk, s_A)$
• $(jsk, jvk, st_{\text{judge}}) \leftarrow (\perp, \perp, \perp)$
• $pp \leftarrow (pk_A, pk_T, ivk, pp_{\mathtt{HT}}, vk_{\mathtt{HT}}, jvk)$
• output $(pp, msk, jsk, st_{\text{judge}}, st_{\text{issuer}})$

$\underline{\mathsf{UserJoin}(pp, a | pp, msk, st_{\text{issuer}})}$
U: send $a$ to key issuer
KI: $crt \leftarrow_{\$} \{0,1\}^\lambda$
KI: $s_a \leftarrow \mathsf{TDP.Inv}^a(sk_A, s_A)$
KI: $usk_a \leftarrow (a, s_a, k_K, k_S, sk_T, \{ldk_b = \mathsf{PRF.Eval}(k_D, \mathsf{TDP.Inv}^b(sk_A, s_A))\}_{b \le a}, crt)$
KI: publish $crt$
KI: send $(usk_a, st_{\text{issuer}})$ to user
U: output $usk_a$ and $st_{\text{user}} = st_{\text{issuer}}$ to user

$\underline{\mathsf{Init}(pp, msk, st_{\text{issuer}} | pp, st_{\text{server}})}$
KI: if $st_{\text{issuer}}.t > 0$: abort protocol
KI: send "initialize" to server
S: initialize empty AADs $\mathtt{HT}$ and $\mathtt{CT}$ with digests digest($HT$) and digest($\mathtt{CT}$)
S: send $dig = (\text{digest}(\mathtt{HT}), \text{digest}(\mathtt{CT}))$ to key issuer
KI: $\sigma \leftarrow \Sigma.\mathsf{Sign}(isk, \langle st_{\text{issuer}}.t+1, dig \rangle)$
KI: $\mathtt{lco} \leftarrow (st_{\text{issuer}}.t+1, dig, \sigma)$
KI: send $\mathtt{lco}$ to server
KI: output $st_{\text{issuer}} = \mathtt{lco}$ to key issuer
S: output $st_{\text{server}} = (\mathtt{HT}, \mathtt{CT}, \mathtt{lco})$ to server
```

Figure 7.1: Algorithms Setup and protocols UserJoin and Init of the RLS scheme

```
$\underline{\mathsf{Update}(pp,msk,st_{\text{issuer}},DC|pp,st_{\text{server}})}$
**KI:** $t' \leftarrow t+1$
**KI:** $H \leftarrow \emptyset$
**KI:** $C \leftarrow \emptyset$
**KI:** for each $(d,v,r,w,o,k) \in DC$:
  • for each $kw \in k$:
    – $k_{kw} \leftarrow \mathsf{PRF.Eval}(k_L,kw)$
    – $k_{kw,a} \leftarrow \mathsf{PRF.Eval}(k_{kw},\mathsf{TDP.Inv}^a(sk_A,,s_A))$
    – $(\sigma_{d,v,t,kw},d'_{d,v,t,kw}) \leftarrow \Sigma.\mathsf{Sign}(sk_\Sigma,\langle d,v,t,k_{kw,a}\rangle)$
    – $s_{kw} \leftarrow \mathsf{PRF.Eval}(k_S,kw)$
    – $v_{kw,t'} \leftarrow \mathsf{TDP.Inv}^t(sk_T,s_{kw})$
    – $\ell_{kw,a,t'} \leftarrow \mathsf{PRF.Eval}(k_{kw,a},\langle v_{kw,t'},1\rangle)$
    – $k_{kw,a,t'} \leftarrow \mathsf{PRF.Eval}(k_{kw,a},\langle v_{kw,t'},0\rangle)$
    – $e_{kw,a,t'} \leftarrow \langle d,v,d'_{d,v,t,kw}\rangle \oplus k_{kw,a,t'}$
    – $H \leftarrow H \cup \{(\ell_{kw,a,t'},(\sigma_{d,v,t,kw},e_{kw,a,t'}))\}$
  • $ct \leftarrow \mathsf{Sym.Enc}(\mathsf{PRF.Eval}(k_D,\mathsf{TDP.Inv}(sk_A,s_A)),k)$
  • $C \leftarrow C \cup \{((d,v),ct)\}$
**KI:** send $(H,C)$ to server
**S:** $\mathtt{HT}' \leftarrow \mathtt{HT}$
**S:** $\mathtt{CT}' \leftarrow \mathtt{CT}$
**S:** $(\mathtt{HT},\text{digest}(\mathtt{HT})) \leftarrow \mathsf{AAD.Append}(pp_{\mathtt{HT}},\mathtt{HT}',H)$
**S:** $(\mathtt{CT},\text{digest}(\mathtt{CT})) \leftarrow \mathsf{AAD.Append}(pp_{\mathtt{CT}},\mathtt{CT}',C)$
**S:** $\pi_{\mathtt{HT}} \leftarrow \mathsf{AAD.PAppend}(pp_{\mathtt{HT}},\mathtt{HT}',\mathtt{HT})$
**S:** $\pi_{\mathtt{CT}} \leftarrow \mathsf{AAD.PAppend}(pp_{\mathtt{CT}},\mathtt{CT}',\mathtt{CT})$
**S:** send $(\text{digest}(\mathtt{HT}),\text{digest}(CT),\pi_{\mathtt{HT}},\pi_{\mathtt{CT}})$ to key issuer
**KI:** if verification of append-only proofs fails: raise alarm and abort protocol
**KI:** $dig \leftarrow (\text{digest}(\mathtt{HT}),\text{digest}(\mathtt{CT}))$
**KI:** $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk_\Sigma,\langle t',dig\rangle)$
**KI:** $\mathtt{lco} \leftarrow (t',dig,\sigma)$
**KI:** send $\mathtt{lco}$ to server
**KI:** output $st_{\text{issuer}} = \mathtt{lco}$ to key issuer
**S:** output $st_{\text{server}} = (\mathtt{HT},\mathtt{CT},\mathtt{lco})$ to server
```

Figure 7.2: Protocol Update of the RLS scheme

```
$\underline{\mathsf{Search}(pp, usk_a, st_{\text{user}}, kw | pp, st_{\text{server}})}$
U: send "search" to server
S: send lco to user
U: if $\Sigma.\mathsf{Verify}(ivk, \langle \mathtt{lco}.t, \mathtt{lco}.dig\rangle, \mathtt{lco}.\sigma) \neq 1$: raise alarm and abort protocol
U: send $st_{\text{user}}.t$ to server
S: determine dictionaries $\mathtt{HT}'$ and $\mathtt{CT}'$ for time step $st_{\text{user}}.t$
S: $\pi_{\mathtt{HT}} \leftarrow \mathsf{AAD.PAppend}(pp_{\mathtt{HT}}, \mathtt{HT}', \mathtt{HT})$
S: $\pi_{\mathtt{CT}} \leftarrow \mathsf{AAD.PAppend}(pp_{\mathtt{CT}}, \mathtt{CT}', \mathtt{CT})$
S: send $(\pi_{\mathtt{HT}}, \pi_{\mathtt{CT}})$ to user
U: if verification of append-only proofs fails: raise alarm and abort protocol
U: $st_{\text{user}} \leftarrow \mathtt{lco}$
U: $k_{kw} \leftarrow \mathsf{PRF.Eval}(k_K, kw)$
U: $K \leftarrow \emptyset$
U: for $a > i \geq 0$:
    • $k_{kw,i} \leftarrow \mathsf{PRF.Eval}(k_{kw}, \mathsf{TDP.Eval}^{a-i}(pk_A, s_a))$
    • $K \leftarrow K \cup \{k_{kw,i}\}$
U: $s_{kw} \leftarrow \mathsf{PRF.Eval}(k_S, kw)$
U: $v_{kw,t} \leftarrow \mathsf{TDP.Inv}^t(sk_T, s_{kw})$
U: $query \leftarrow (K, v_{kw,t})$
U: send $query$ to server
S: $rslt \leftarrow \emptyset$
S: for each $k \in K$:
    • for $0 \leq i < t$:
        – $\ell \leftarrow \mathsf{PRF.Eval}(k, \langle \mathsf{TDP.Eval}^i(sk_T, v_{kw,t}), 0\rangle)$
        – $(\ell, \{e\}, \pi) \leftarrow \mathsf{AAD.Lookup}(pp, \mathtt{HT}, \ell)$
        – $\langle d, v, d'\rangle \leftarrow e \oplus \mathsf{PRF.Eval}(k, \langle \mathsf{TDP.Eval}^i(sk_T, v_{kw,t}), 1\rangle)$
        – $(\{ct\}, \pi') \leftarrow \mathsf{AAD.Lookup}(pp_{\mathtt{CT}}, \mathtt{CT}, (d, v))$
        – $rslt \leftarrow rslt \cup \{(k, \{e\}, \pi, ct, \pi')\}$
S: send $rslt$ to user
U: if verification of received membership proofs or signatures fails: raise alarm and abort
   protocol
U: decrypt all document ciphertexts that occur in $rslt$
U: output set of resulting plaintexts and $st_{\text{user}}$ to user

$\underline{\mathsf{Resolve}(pp, st_{\text{server}} | pp, jsk, st_{\text{judge}})}$
S&KI: abort protocol

$\underline{\mathsf{Version}(pp, st_{\text{user}})}$
• output $st_{\text{user}}.t$
```

Figure 7.3: Protocols Search and Resolve, and algorithm Version of the RLS scheme

If there is a corrupt user with access level a', denoted corrupt a' below, the leakage also includes the plaintexts/keyword sets of documents accessible to the corrupt user; we denote this as

$$DC_{\text{new}}(a') = \{(d, v, a, k) : (d, v, a, w, o, k) \in DC_{\text{new}} \land a' \geq a\}$$

Therefore,

$$\mathcal{L}_{\text{Update}}(\bot, DC_{\text{new}}) = (\texttt{CT}(DC_{\text{new}}), \forall \text{ corrupt } a' : DC_{\text{new}}(a')).$$

Whenever an honest user with access level a performs search, the RLS scheme for all document collections DC_t leaks the document names and version numbers of documents containing the searched keyword. We denote this as

$$doc(DC_t, a, kw) = \{(d, v) : (d, v, a', w, o, k) \in DC_t \land a \geq a' \land kw \in k\}.$$

From the query an identifier $id(kw)$ of the searched keyword and the user's access level can be extracted.

$$\mathcal{L}_{\text{Search}}(crt, kw) = (a, id(kw), \forall t : doc(DC_t, a, kw)).$$

Nothing can be learned from operation Resolve.

Given this leakage, it is fairly easy to see that, in the random oracle model, simulating the behavior of the real system based on leakage rather than real data is possible, and no PPT adversary can distinguish simulated behavior from the behavior of the real system. As a result, if RLS is instantiated with PRF PRF, trapdoor permutation TDP, secure AAD scheme AAD, secure signature scheme (with delayed verifiability) Σ, and CPA-secure symmetric encryption scheme Sym, RLS provides *data confidentiality.*

The RLS scheme is also *fork consistent* if it is instantiated with secure signature scheme Σ. Fork consistency is a consequence of the fact that user states need to be signed by the key issuer and the key issuer signs at most one user state for each version number. So, an adversary that manages to break RLS's fork consistency has forged a signature under the key issuer's signature verification key in order to cause users to adopt distinct states for the same version number.

The RLS scheme is also *verifiable* if it is instantiated with secure signature scheme Σ and secure AAD scheme AAD. This is because the users know how many results AAD.Lookup queries the server is to report in raw search results ($rslt$). Due to AAD membership security, the server has to report all relevant HT entries generated by the key issuer, and because all HT entries that originate from the key issuer are signed by the key issuer, the server also cannot report HT entries that do not originate from the key issuer without facing detection, unless the server is able to forge signature under the key issuer's singing key.

Finally, from the above update leakage, we can see that the RLS scheme is *forward private,* because the leakage, in the absence of dishonest users, does not leak any information on keywords beyond an upper bound the number of keywords affected by the update.

7.2 The DIA Scheme

With the Σοφος scheme [Bos16] and the RLS scheme above, we have seen that it is possible to integrate the dictionary HT with a list using Bost's technique. While Σοφος, as an evolution of the CGKO scheme [Cur+11], integrates HT with the D lists from the CGKO scheme, here, we take dSEAC as a basis and integrate its dictionary HT with A lists. This results in a scheme with dictionary-integrated access control, called DIA.

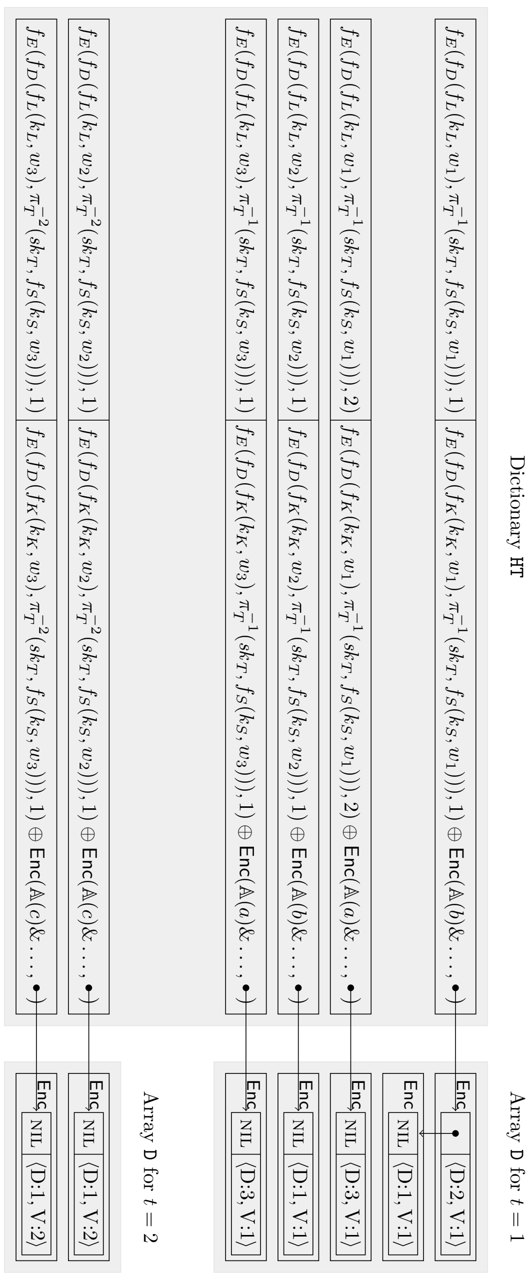

Dictionary HT
$f_E(f_D(f_L(k_L, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1))), 1)$
$f_E(f_D(f_K(k_K, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1))), 1) \oplus \mathsf{Enc}(\mathbb{A}(b)\&\ldots, \bullet)$
$f_E(f_D(f_L(k_L, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1))), 2)$
$f_E(f_D(f_K(k_K, w_1), \pi_T^{-1}(sk_T, f_S(k_S, w_1))), 2) \oplus \mathsf{Enc}(\mathbb{A}(a)\&\ldots, \bullet)$
$f_E(f_D(f_L(k_L, w_2), \pi_T^{-1}(sk_T, f_S(k_S, w_2))), 1)$
$f_E(f_D(f_K(k_K, w_2), \pi_T^{-1}(sk_T, f_S(k_S, w_2))), 1) \oplus \mathsf{Enc}(\mathbb{A}(b)\&\ldots, \bullet)$
$f_E(f_D(f_L(k_L, w_3), \pi_T^{-1}(sk_T, f_S(k_S, w_3))), 1)$
$f_E(f_D(f_K(k_K, w_3), \pi_T^{-1}(sk_T, f_S(k_S, w_3))), 1) \oplus \mathsf{Enc}(\mathbb{A}(a)\&\ldots, \bullet)$
$f_E(f_D(f_L(k_L, w_2), \pi_T^{-2}(sk_T, f_S(k_S, w_2))), 1)$
$f_E(f_D(f_K(k_K, w_2), \pi_T^{-2}(sk_T, f_S(k_S, w_2))), 1) \oplus \mathsf{Enc}(\mathbb{A}(c)\&\ldots, \bullet)$
$f_E(f_D(f_L(k_L, w_3), \pi_T^{-2}(sk_T, f_S(k_S, w_3))), 1)$
$f_E(f_D(f_K(k_K, w_3), \pi_T^{-2}(sk_T, f_S(k_S, w_3))), 1) \oplus \mathsf{Enc}(\mathbb{A}(c)\&\ldots, \bullet)$
Array D for $t = 1$
Enc ⟨D:2, V:1⟩
Enc NIL ⟨D:1, V:1⟩
Enc NIL ⟨D:3, V:1⟩
Enc NIL ⟨D:1, V:1⟩
Enc NIL ⟨D:3, V:1⟩
Array D for $t = 2$
Enc NIL ⟨D:1, V:2⟩
Enc NIL ⟨D:1, V:2⟩

Figure 7.4: DIA's post update index structure for our example document collection from Page 38; the figure includes the time-based extension from dSEAC and an additional PRF f_E to enumerate HT entries for the same keyword, but different access policies

An overview of the index structure of the DIA scheme can be found in Figure 7.4. Essentially, DIA integrates dictionary HT and A lists by enumerating the list nodes. We achieve this by applying an additional PRF f_E, using what was previously used as a label or encryption key as a PRF key, and using a counter value as the PRF's argument. We note that the D lists of the dSEAC scheme remain unchanged in DIA.

As shown in the figure, the DIA scheme is clearly compatible with the time-based extensions from dSEAC. However, applying dSEAC's mechanisms for verifiability, particularly list authenticators on D lists, is not possible in a straightforward manner. The reason is that an authenticator on a D list would also need to authenticate the access structure shared by the documents the D list points to.

To that end, we propose a slight adaption of our signatures with delayed verifiability: we allow some data to be verified immediately as part of algorithm VOrig, while the complete data is only verified by algorithm VSig. This adaption can be implemented in the same way that signatures with delayed verifiability can be implemented, namely, by following a commit–then–sign approach; however, in our adaption, the immediately verifiable data is signed together with the commitment. The commitment only considers the data that is to be verified only after the commitment's open value has been released to the public.

List authenticators generated from D lists and the access policies shared by the documents from the lists can then be included in HT entries in the same way we include them in dSEAC. By implementing HT as an AAD, we can also apply the concept of last completed operations to DIA.

Although we do not give pseudocode description of the DIA scheme, due to DIA's similarity to dSEAC, with the structure shown in Figure 7.4, the proposed changes to list authenticators, and the concept of last completed operations, the DIA scheme is a dynamic searchable encryption scheme with access control and satisfies our four security notions for such schemes. Proofs of DIA satisfying the security notions are rather similar to the proofs for dSEAC, due to the similarity of the schemes in terms of applied techniques.

During operation, the DIA scheme leaks essentially the same data as the dSEAC scheme. The only difference is the update leakage: DIA integrating A lists into dictionary HT changes the leakage on keywords. Particularly, dSEAC leaks the number of distinct keyword in the set of new and updated documents (number of new HT entries) and an *upper bound* on the number of keyword–policy pairs (size of memory array A). In contrast, DIA leaks the actual number of keyword–policy pairs instead, because that number is the number of new HT entries.

7.3 Comparison

We briefly compare our RLS, DIA, and dSEAC schemes. The Σοφος-based RLS scheme achieves dynamic searchable encryption with access control for the class of hierarchical access levels. The scheme is in the single writer–multi-reader setting, and achieves data confidentiality, fork consistency, verifiability, and forward privacy. While our DIA and dSEAC schemes satisfy the same security notions, they do so for a broader class of access policies, namely, the access policies supported by the underlying ABE scheme, and in the multi-writer–multi-reader setting. On the other hand, DIA and dSEAC leak significantly more information than the RLS scheme.

The differences in leakage can be mostly attributed to the simple class of access policies supported by RLS. RLS's restriction to the single writer setting only accounts for RLS leakage functions not considering different forks.

All in all, in this chapter, we have seen an example of the trade-offs in searchable encryption with access control. Particularly, we have seen the trade-off between the expressiveness of access policies supported by a (verifiable) scheme and the scheme's leakage.

8 Closing Remarks

We finish this part of the thesis by a brief discussion of means to shorten users' keys in our searchable encryption schemes, particularly SEAC, dSEAC and DIA, and discussing the revocation of users from our schemes.

Shortening user keys. All of our searchable encryption schemes that rely on ABE for enforcing access control have the drawback of requiring keyword-specific attributes for all possible keywords to be present in every users' secret key. We must therefore ask whether we can reduce the sizes of user keys. Indeed, the original presentation of our SEAC scheme in [Lök17] features short user keys by applying a particular type of ABE, namely, multi-authority attribute-based encryption (MA-ABE) [Cha07] with authority key customization [Lök17].

The idea behind MA-ABE is to allow policies to feature attributes managed by multiple attribute authorities that hand out users keys independently. Yet, the keys a user receives from the various attributes authorities are compatible amongst each other, so can be used in decryption, while being incompatible with keys of other users. Authority key customization then allows an attribute authority to give users all attributes managed by the authority in a concise way (called customized authority secret). As a consequence, users can generate actual user keys from the customized authority secret whenever the need arises.

The original SEAC employs one attribute authority for each of the attribute namespaces Usr, Op and Wrd, and gives a customized authority secret for namespace Wrd to each user as part of their secret keys. As a result, users can generate keyword-specific attribute keys themselves and do not need to obtain them explicitly as part of their secret key.

The same technique can be applied to our dSEAC and DIA schemes, and can also be extended to namespace Tme used by these schemes. The reason why we have not done so is due to the MA-ABE schemes from the literature. Those schemes either do not feature authority key customization, for example, Chase's scheme [Cha07] and the Lewko/Waters scheme [LW11], or are proven secure in a static setting that is incompatible with our dynamic schemes, for example, the Rouselakis/Waters scheme [RW15].

Therefore, we encourage research into fully (and dynamically) secure MA-ABE schemes with authority key customization. Applying such schemes in our generic constructions is an elegant way to reduce the sizes of user keys.

An alternative approach to shortening user keys is to make the formulation of search queries interactive between users and the key issuer, so whenever a user wants to search for some keyword, she has to ask the key issuer for the corresponding keyword-specific attribute.

However, the privacy implications as well as the need for additional communication make this solution undesirable.

Revoking users. Many searchable encryption schemes that address a multi-reader setting and achieve access control via ABE provide means for revoking user's access rights, for example, the schemes presented in [Yan+13; YJ12; YJR13; Yan+15; Yu+10; Sun+16]. In these schemes, user revocation is achieved by applying proxy re-encryption or by attribute version numbers. In the former case, whenever a user's access rights are revoked, the proxy stops re-encrypting ciphertext under the re-encryption key that complements the revoked user's key.

The latter option associates a version number with each attribute. Whenever an attribute's version number changes, user keys that contain the attribute are updated to the new version number by applying a dedicated update key. So when a user's access rights are revoked, all attributes that the user held get new version numbers, and the corresponding update keys are only distributed to the remaining users. This process also involves the re-encryption of all ciphertexts, to update attribute version numbers there as well. Since re-encrypting all ciphertexts is potentially very time consuming, it has been suggested to perform re-encryption spread out over time, namely, whenever a ciphertext is requested, rather than re-encrypting all ciphertexts immediately.

Both of the above approaches are quite costly in terms of computation, while the effectiveness of the mechanisms depends on whether or not the entity responsible for re-encryption cooperates. In the proxy re-encryption scenario, the proxy must refuse to apply the re-encryption key that corresponds to the revoked user's secret key. In the attribute version number setting, the entity serving users' requests for ciphertexts can only serve ciphertexts that have been re-encrypted. However, if the entity responsible for re-encryption is dishonest, it can choose to serve revoked users with ciphertexts that have been re-encrypted under the revoked user's re-encryption key or that have not been updated to new version numbers.

In our dSEAC and DIA schemes, users need to authenticate themselves by presenting a certificate of enrollment whenever they interact with the server. By applying standard certificate revocation mechanisms from the literature, we can effectively revoke users' access rights without the need of costly re-encryption. At the same time, our certificate-based approach to user revocation in secure is the same circumstances that the typical approaches are secure in, namely, if the server fully cooperates.

Part III

Reputation Systems

9 Introduction to Reputation Systems

Reputation systems allow customers to rate products and merchants, reflecting the reputation of the rated entity. In turn, potential customers use a merchant's or product's reputation in their decision making process whether or not to buy from that merchant or to buy the product. The underlying assumption is that past behavior or quality is a predictor of future behavior or quality [JG09]. Hence, reputation is conceived as a metric for business prospects. Consequentially, reputation systems have attracted research from the field of economy. Due to the perceived value of good reputation, reputation systems are also prone to attacks by malicious parties, and thus, such systems have also attracted security research.

While we mainly focus on security aspects of reputation systems, the economic aspect of reputation has an impact on the types of security threats that we need to consider. For example, consider a merchant trying to improve her reputation score by buying fake ratings. Obtaining such ratings is somewhat costly, so the merchant will not buy ratings unless the potential benefits of the ratings, like improved sales, outweigh the ratings' costs. This example illustrates why, in some instances, assuming a rational adversary is reasonable, and this should be reflected in security analyses of reputation systems. However, great care must be taken when deciding that it is reasonable to only consider rational adversaries, as a real-world example shows: A journalist has caused a non-existent restaurant to be among the best rated restaurants in London in the TripAdvisor reputation system [Hor17].

Personal reputation in crowd work. As stated in the Chapter 1, crowd work is a work model that will potentially have significant economic impact in the future. We consider reputation in the context of that work model. In crowd work, a requester seeks a solution to a task. Via an open call published online, a group, called crowd, is asked to work towards a solution in exchange for monetary rewards.

In practice, the open call in crowd work is typically communicated via dedicated platforms, which manage their respective crowd, arrange worker payments, and handle legal matters, for example, the rights to the results of the crowds' works. Established platforms often specialize in particular types of tasks. Types of tasks and platforms, that have been identified are (1) text creation ("marketplace platform"), (2) design and innovation, (3) micro tasks, (4) mobile tasks/crowd sensing, and (5) software development and testing.

In mobile and micro tasks, tasks are so simple that generally no expertise is required for solving the tasks. Software and development tasks may or may not require significant expertise. In creative tasks, particularly text creation, design, and innovation, great expertise is often required for solving tasks.

In our consideration of crowd work, we mainly focus on creative tasks, because platforms that specialize in creative tasks often make use of reputation systems in a way that workers' reputations influence their rewards. For example, reputable workers may be granted greater compensation directly, per solved task, or indirectly by being offered a larger number of tasks. The relation between reputation and compensation impacts workers' abilities to migrate to other platforms (voluntarily or involuntarily) or work for multiple platforms simultaneously. In the former case, reputation is lost, and in the second case, reputation is distributed and neither platform knows about the workers' true (combined) reputation. This situation, separate reputation systems for each platform, results in vendor lock-in to the crowd workers' disadvantage.

We address this issue by our proposal of *personal reputation systems*. In a personal reputation system, crowd workers' reputation is stored by the crowd workers themselves. As a result, our proposal mitigates vendor lock-in and loss of reputation due to unfortunate circumstances.

Our contribution. We make three contributions to the research on reputation systems, published at STM 2019 [BL19b]. First, we propose the novel notion of personal reputation systems as a particular type of reputation systems. Second, we propose the novel security notion of rater anonymity towards the general public. The notion is rooted in the crowd work setting, and applicable to reputation systems in general and personal reputation systems in particular. Third, we propose PRS, a personal reputation system that satisfies our rater anonymity notion, as well as other security notions typically imposed on reputation systems. Our scheme is particularly noteworthy due to its simplicity.

Overview of Part III. In the upcoming Chapter, we first discuss the role of platforms in crowd work and their impact on personal reputation systems. Building on that discussion, we formalize the notions of personal reputation systems and rater anonymity towards the general public. Eventually, we construct our personal reputation system PRS and discuss its properties. In Chapter 11, we discuss related work on reputation systems and some technologies we use in our construction.

10 Personal Reputation Systems

Our concept of personal reputation systems is derived from the crowd work scenario from the introduction. Therefore, personal reputation systems feature parties common in crowd work. Particularly, we have platforms, raters a.k.a. requesters, and ratees a.k.a. crowd workers. Since we envision personal reputation systems to run in parallel to the infrastructure for crowd work, there is potential for confusion, so we briefly discuss platforms' processes and reputation systems processes in order to separate them.

Platform processes. A platform organizes its work force and mediates between crowd workers and requesters. Crowd workers need to register with the platform in order to be able to solve tasks communicated via the platform. Registration is necessary to provide contact details so crowd workers can receive compensation for their work. Similarly, requesters need to register with the platform before they can communicate their tasks via the platform, so the platform can bill the requesters for its services and for crowd workers' compensation.

Since these registration processes are necessary from the platforms' point of view, platforms simply exist and do not need to register with anyone. Hence, we have two registration processes as part of the platforms' processes.

Reputation system processes. On the other hand, the processes of our personal reputation system are independent of a particular party. Within our system, ratees only communicate with raters, so we need a way for both types of parties to establish their respective (crpytographic) identities with each other. Therefore, we have a straightforward setup procedure for ratees. For raters, establishing their identity is more involved, and involves a platform.

Platforms play a pivotal role in achieving a secure personal reputation system. This is best shown in the involvement of platforms in establishing rater identities: A rater establishes an identity with a platform, and then the platform certifies that a particular ratee has provided a solution to one of the rater's tasks, which was communicated via the platform. We call that certificate platform transaction reference (PTR). Given the PTR, the rater can then contact the respective ratee to obtain a rating token, which, later on, allows the rater to rate the ratee.

Via the rating token, the PTR is included in the final rating. Intuitively, the PTR will give people confidence in the validity of ratings, particularly, that the ratings are from a real requester and are not made up by the ratee. We capture this in our security notion of rating unforgeability. Ratings are verified using a particular verification operation.

In order for platforms to certify relations between raters and ratees via PTRs, platforms need a setup procedure of their own. For technical reasons, personal reputation systems allow for a global setup procedure as well, particularly for setting up parameters shared by all parties and set up by a trusted entity.

Relations between processes. Our personal reputation system is intended to run alongside crowd work platforms, so there are relations between the two systems. After all, the whole point of our personal reputation system is for it to replace or enhance the platforms' internal reputation systems in order to better reflect crowd workers' reputation by considering work done on other platforms. As a consequence, personal reputation systems impact platform processes. The converse is also true. When platforms issue PTRs to raters, they include information on the relation between raters and ratees in the PTR. This takes the form of a transaction identifier, for example, an invoice number.

Outside parties and anonymity. We have briefly mentioned verifiers, parties verifying the ratings a ratee has obtained. Typically, raters want to remain anonymous from verifiers, for example, in order to keep trade secrets. As a consequence crowd workers on design and innovation platforms often have to sign non-disclosure agreements. We reflect this in one of our security notions, namely, rater anonymity against the general public.

Specifically, under that notion, raters are only anonymous towards the general public, but not necessarily towards the parties involved in the transaction that resulted in the rating, specifically the ratee and the platform. This is a weaker anonymity notion than the one typically imposed on reputation systems, which is full rater anonymity against all parties. However, our weaker notion is more sensible in our application scenario: Reasonable (textual) reviews must reference the criticized work. Such references allow crowd workers and platforms to identify the requester, because the criticized solution is uniquely tailored to the requester's task. Thus, in the crowd work scenario, platforms and ratees involved in a transaction can be assumed to able to identify the ratee. Threrefore, the notion of full rater anonymity is not useful in the crowd work context.

10.1 Preliminaries and Definitions

In this section we formalize the ideas laid out above, particularly, the notions of personal reputation systems, rating unforgeability, and rater anonymity towards the general public. We begin with a formalization of personal reputation systems.

Definition 42 ([BL19b]). *A* personal reputation system *consists of four PPT algorithms and three protocols.*

GlobalSetup *takes a security parameter 1^λ and outputs system parameters pp. The algorithm is executed by a trusted party.*

RateeSetup *takes in system parameters pp, and outputs private key usk and verification key uvk.*

PlatformSetup *takes in system parameters pp, and outputs private key psk and verification key pvk.*

IssuePTR *involves a rater and a platform. Both parties take in system parameters pp; the rater additionally takes in the platforms verification key pvk; the platform additionally takes in its private key psk, a helper string tid, and a ratee's verification key uvk. The rater outputs a PTR ptr.*

IssueRT *involves a rater and a ratee. Both parties take in system parameters pp; the rater additionally takes is the ratee's verification key uvk and her* PTR *ptr; the ratee additionally takes in her private key usk. The rater outputs a rating token rt.*

Rate *involves a rate and a ratee. Both parties take in system parameters pp; the rater additionally takes in the ratee's verification key uvk and her rating token rt and her (unprocessed) rating msg; the ratee additionally takes in her private key usk. Both parties output the (processed) rating r.*

Verify *takes in system parameters pp, a ratee's verification key uvk, a (processed) rating r and a platform's verification key pvk, and outputs a bit.*

We require, for $pp \leftarrow \mathsf{GlobalSetup}(1^\lambda)$*, key pairs* $(usk, uvk) \leftarrow \mathsf{RateeSetup}(pp)$ *and* $(psk, pvk) \leftarrow \mathsf{PlatformSetup}(pp)$*,* PTR $(ptr|\bot) \leftarrow \mathsf{IssuePTR}(pp, pvk|pp, tid, uvk)$*, rating token* $(rt|\bot) \leftarrow \mathsf{IssueRT}(pp, uvk, ptr|pp, usk)$*, and unprocessed rating msg, if* $(r|r) \leftarrow \mathsf{Rate}(pp, uvk, rt, msg|pp, usk)$*, then* $\mathsf{Verify}(pp, uvk, r, pvk) = 1$*.*

The correctness notion of personal reputation systems essentially requires ratings to be verified successfully.

The security notion of rating unforgeability is inspired by the signature unforgeability notion from Definition 20 with the Rate oracle replacing the Sign oracle. However, the rating unforgeability experiment $\mathbf{Exp}^{\text{PRS-forge}}$ is more complex due to the presence of additional oracles. The additional oracles are necessary to model the presence of many raters, ratees and platforms. Specifically, the experiment provides the adversary with oracles for setting up honest platforms and raters, for having honest raters obtain rating tokens, and for making honest raters rate a (potentially dishonest) ratee. Experiment $\mathbf{Exp}^{\text{PRS-forge}}$ is shown in Figure 10.1 and we define rating unforgeability relative to the experiment.

Throughout the experiment, we keep track of honest parties, their interactions, and the results of such interactions. The adversary's goal in the experiment is to output a ratee's verification key and a rating such that Verify accepts the rating under the given ratee's verification key and an honest platform's verification key, and no honest rater has created the output rating.

Definition 43 ([BL19b])**.** *A personal reputation system* $\Pi = (\mathsf{GlobalSetup}, \mathsf{RateeSetup}, \mathsf{PlatformSetup}, \mathsf{IssuePTR}, \mathsf{IssueRT}, \mathsf{Rate}, \mathsf{Verify})$ *is said to provide* rating unforgeability *if, for all* PPT *adversaries* $\mathcal{A}$*,*

$$\Pr[\mathbf{Exp}^{\textit{PRS-forge}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda).$$

A second security property of personal reputation systems is the notion of rater anonymity towards the general public motivated before. We model the notion by an indistinguishability experiment $\mathbf{Exp}^{\text{PRS-anon}}$. The experiment is presented in Figure 10.2 and relies on oracles behaving as shown in Figures 10.3 and 10.4. In contrast to the unforgeability experiment, honest parties in the anonymity experiment may interact with a broader range of dishonest parties. As a result, we have multiple variants for some of the oracles, particularly for oracles used to issue PTRs, issue rating tokens, and rate an honest ratee. Specifically, these oracles come in variants C and H. Variant C covers the case of a corrupt rater interacting with an honest platform or ratee. Variant H covers the case of an honest rater interacting with a dishonest platform or an honest or corrupt ratee, depending on the oracle. The oracle for issuing PTRs also features a Variant D, which covers the case of an honest rater interacting with an honest platform.

```
$\mathbf{Exp}^{\text{PRS-forge}}_{\mathcal{A},\Pi}(\lambda)$
• $P,R \leftarrow \emptyset$
• $pp \leftarrow \mathsf{GlobalSetup}(1^\lambda)$
• $(uvk^*,r^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\text{CreatePlatform}}(),\mathcal{O}_{\text{CreateRater}}(),\mathcal{O}_{\text{IssuePTR}}(\cdot,\cdot,\cdot),\mathcal{O}_{\text{IssueRT}}(\cdot),\mathcal{O}_{\text{Rate}}(\cdot,\cdot)}(pp)$, where the
  experiment reacts to oracle queries as follows:
  – $\mathcal{O}_{\text{CreatePlatform}}$ without arguments:
      ∗ $(psk,pvk) \leftarrow \mathsf{PlatformSetup}(pp)$
      ∗ $P \leftarrow P \cup \{(psk,pvk)\}$
      ∗ output $pvk$
  – $\mathcal{O}_{\text{CreateRater}}$ without arguments:
      ∗ $rid \leftarrow |R|$
      ∗ $R \leftarrow R \cup \{(rid,\emptyset)\}$
      ∗ output $rid$
  – $\mathcal{O}_{\text{IssuePTR}}$ with arguments $(pvk,rid,uvk)$:
      ∗ if $\exists(s,pvk) \in P$ for some $s$ and $\exists(rid,X) \in R$ for some $X$:
          · $tid \leftarrow_{\$} \{0,1\}^\lambda$
          · run protocol $\mathsf{IssuePTR}$ playing the rater and the platform, taking $(pp,$
            $pvk)$ as the rater's input and $(pp,s,tid,uvk)$ as the platform's input; let
            $ptr$ be the rater's output
          · if the protocol did not abort: $Y \leftarrow X \cup \{(tid,(uvk,pvk,ptr),\perp,0,\perp)\}$
          · else: $Y \leftarrow X \cup \{(tid,\perp,\perp,0,\perp)\}$
          · $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,Y)\}$
          · return $tid$
      ∗ return $\perp$
  – $\mathcal{O}_{\text{IssueRT}}$ with argument $rid,tid$:
      ∗ if $\exists(rid,X) \in R$ for some $X$ and $(tid,(u,v,p),\perp,0,\perp) \in X$ for some $u$, $v$ and $p$:
          · run protocol $\mathsf{IssueRT}$ playing the rater, taking $(pp,u,p)$ as input; the
            adversary plays the ratee's part; let $rt$ be the rater's output
          · it the protocol did not abort: $Y \leftarrow (X \setminus \{(tid,(u,v,p),\perp,0,\perp)\}) \cup \{(tid,(u,$
            $v,p),rt,0,\perp)\}$
          · else: $Y \leftarrow (X \setminus \{(tid,(u,v,p),\perp,0,\perp)\}) \cup \{(tid,(u,v,p),\mathsf{invalid},0,\perp)\}$
          · $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,Y)\}$
          · return 1
      ∗ else: return 0
  – $\mathcal{O}_{\text{Rate}}$ with arguments $(rid,tid,msg)$:
      ∗ if $\exists(rid,X) \in R$ for some X and $\exists(tid,(u,v,p),r',0,\perp) \in X$ for some $u,v$ and $p$
        and $r' \neq \mathsf{invalid}$:
          · run protocol $\mathsf{Rate}$ playing the rater, taking $(pp,u,r',msg)$ as input; the
            adversary plays the ratee's part; let $r$ be the rater's output
          · $Y \leftarrow (X \setminus \{(tid,(u,v,p),r',0,\perp)\}) \cup \{(tid,(u,v,p),r',r',1,r)\}$
          · $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,Y)\}$
          · return 1
      ∗ return 0
• The experiment outputs 1 if
  1. $\exists(s,pvk) \in P$ such that $\mathsf{Verify}(pp,uvk^*,r^*,pvk) = 1$ and,
  2. for all tuples $(rid,X) \in R$ and $(tid,(uvk^*,pvk,p),\cdot,\cdot,r) \in X$, we have $r \neq r^*$;
  otherwise the experiment outputs 0.
```

Figure 10.1: Rating unforgeability experiment for personal reputation system Π and adversary $\mathcal{A}$

$\underline{\mathbf{Exp}^{\text{PRS-anon}}_{\mathcal{A},\Pi}(\lambda)}$
- $P \leftarrow \emptyset$
- $R \leftarrow \emptyset$
- $pp \leftarrow \mathsf{GlobalSetup}(1^\lambda)$
- $(usk^*, uvk^*) \leftarrow \mathsf{RateeSetup}(pp)$
- $(st_\mathcal{A}, rid_0, rid_1, pvk^*, msg^*) \leftarrow \mathcal{A}_1^{\substack{\mathcal{O}_{\text{CreatePlatform}}(), \mathcal{O}_{\text{CreateRater}}(), \mathcal{O}_{\text{IssuePTR:}C}(\cdot,\cdot), \mathcal{O}_{\text{IssuePTR:}D}(\cdot,\cdot,\cdot), \\ \mathcal{O}_{\text{IssuePTR:}H}(\cdot,\cdot,\cdot), \mathcal{O}_{\text{IssueRT:}C}(\cdot), \mathcal{O}_{\text{IssueRT:}H}(\cdot), \mathcal{O}_{\text{Rate:}C}(\cdot,\cdot), \mathcal{O}_{\text{Rate:}H}(\cdot,\cdot)}}$
- If $(rid_0, X), (rid_1, Y) \notin R$ for any X and Y or $(psk, pvk^*) \notin P$ for any psk, then the experiment outputs -1 and aborts
- $b \leftarrow_\$ \{0,1\}^\lambda$
- Perform actions of oracle queries $tid \leftarrow \mathcal{O}_{\text{IssuePTR:}D}(pvk^*, rid_b, uvk^*)$, $\mathcal{O}_{\text{IssueRT:}H}(rid_b, tid)$, and $s \leftarrow \mathcal{O}_{\text{Rate:}H}(rid, tid, msg^*)$
- If $s = \perp$, the experiment outputs -1 and aborts
- $b' \leftarrow \mathcal{A}_2^{\substack{\mathcal{O}_{\text{CreatePlatform}}(), \mathcal{O}_{\text{CreateRater}}(), \mathcal{O}_{\text{IssuePTR:}C}(\cdot,\cdot), \mathcal{O}_{\text{IssuePTR:}D}(\cdot,\cdot,\cdot), \\ \mathcal{O}_{\text{IssuePTR:}H}(\cdot,\cdot,\cdot), \mathcal{O}_{\text{IssueRT:}C}(\cdot), \mathcal{O}_{\text{IssueRT:}H}(\cdot), \mathcal{O}_{\text{Rate:}C}(\cdot,\cdot), \mathcal{O}_{\text{Rate:}H}(\cdot,\cdot)}}(st_\mathcal{A})$
- The experiment outputs 1 if $b = b'$ and 0 otherwise.

Figure 10.2: Rater anonymity experiment for personal reputation system Π and adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$); oracles are shown in Figures 10.3 and 10.4

Definition 44 ([BL19b]). *A personal reputation system* $\Pi = (\mathsf{GlobalSetup}, \mathsf{RateeSetup}, \mathsf{PlatformSetup}, \mathsf{IssuePTR}, \mathsf{IssueRT}, \mathsf{Rate}, \mathsf{Verify})$ *is said to provide* rater anononymity towards the general public *if, for all* PPT *adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$),

$$|\Pr[\mathbf{Exp}^{\textit{PRS-anon}}_{\mathcal{A},\Pi}(\lambda) = 1] - 1/2| \leq \mathsf{negl}(\lambda).$$

10.2 The PRS Construction

In this section, we construct our PRS personal reputation system. To this end, we use a special building block not introduced among the general building blocks from Chapter 3. Namely, we use hash chains as described below. Combining hash chains with signature and commitment schemes, we then build our personal reputation system.

10.2.1 Hash Chains

A hash chain is a folklore example[3] of an authenticated data structure, c.f. Section 3.5 for an example of another authenticated data structure and some of the terminology. Essentially, a hash chain is a sequence of blocks of data chained together by a hash function. We want to restrict who can create blocks, so we have the owner of a hash chain sign the blocks. As a result, every block on a chain consists of three fields, (1) payload or data, (2) block hash, and (3) block signature. The block hash is a hash on the block's payload and a predecessor block. The block signature is a signature on the block hash.

Due to the block hash, we are able to verify the chain property. That is, given a set of blocks, we can sort the blocks using the predecessor relation as encoded in the block hash, and the chain property holds if and only if exactly one block has no predecessor in the set and exactly one block is not a predecessor of any other block from the set. Hence, a chain

[3]Similar concepts occur in symmetric encryption as the CBC mode of operation, in message authentication as CBC-MAC, and in the Merkle-Damgård transform.

```
• $\mathcal{O}_{\text{CreatePlatform}}$ without arguments:
  – $(psk,pvk) \leftarrow \mathsf{PlatformSetup}(pp)$
  – $P \leftarrow P \cup \{(psk,pvk)\}$
  – output $pvk$
• $\mathcal{O}_{\text{CreateRater}}$ without arguments:
  – $rid \leftarrow |R|$
  – $R \leftarrow R \cup \{(rid,\emptyset)\}$
  – output $rid$
• $\mathcal{O}_{\text{IssuePTR}:C}$ with arguments $(pvk,uvk)$:
  – $tid \leftarrow_{\$} \{0,1\}^{\lambda}$
  – run protocol $\mathsf{IssuePTR}$ playing the platform, taking $(pp,psk,uvk)$ as input; the
    adversary plays the rater's part
• $\mathcal{O}_{\text{IssuePTR}:D}$ with arguments $(pvk,rid,uvk)$:
  – if $\exists (s,pvk) \in P$ for some $s$ and $\exists (rid,X) \in R$ for some $X$:
      ∗ $tid \leftarrow_{\$} \{0,1\}^{\lambda}$
      ∗ run protocol $\mathsf{IssuePTR}$ playing the rater and the platform, taking $(pp,pvk)$ as
        the rater's input and $(pp,s,tid,uvk)$ as the platform's input; let $ptr$ be the
        rater's output
      ∗ if the protocol did not abort: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid, X \cup \{(tid,$
        $(uvk,pvk,ptr),\perp,0,\perp)\})\}$
      ∗ else: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,X \cup \{(tid,\perp,\perp,0,\perp)\})\}$
      ∗ return $tid$
  – return $\perp$
• $\mathcal{O}_{\text{IssuePTR}:H}$ with arguments $(pvk,rid,tid)$:
  – if $\exists (rid,X) \in R$ for some $X$ and $\forall (t,\cdot,\cdot,\cdot,\cdot) \in X : t \neq tid$:
      ∗ run protocol $\mathsf{IssuePTR}$ playing the rater, taking $(pp,pvk)$ as input; the adversary
        plays the platform's part; let $ptr$ be the rater's output
      ∗ if the protocol did not abort: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid, X \cup \{(tid,$
        $(uvk,pvk,ptr),\perp,0,\perp)\})\}$
      ∗ else: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,X \cup \{(tid,\perp,\perp,0,\perp)\})\}$
      ∗ return $tid$
  – return $\perp$
• $\mathcal{O}_{\text{IssueRT}:C}$ without arguments:
  – run protocol $\mathsf{IssueRT}$ playing the ratee, taking $(pp,usk^{*})$ as input; the adversary
    plays the rater's part
• $\mathcal{O}_{\text{IssueRT}:H}$ with argument $rid,tid$:
  – if $\exists (rid,X) \in R$ for some $X$ and $(tid,(u,v,p),\perp,0,\perp) \in X$ for some $u$, $v$ and $p$:
      ∗ run protocol $\mathsf{IssueRT}$ playing the rater, taking $(pp,u,p)$ as input; if $u = uvk^{*}$,
        also play the ratee on input $(pp,,usk^{*})$, otherwise the adversary plays the
        ratee's part; let $rt$ be the rater's output
      ∗ it the protocol did not abort: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,(X \setminus \{(tid,(u,v,p),$
        $\perp,0,\perp)\}) \cup \{(tid,(u,v,p),rt,0,\perp)\})\}$
      ∗ else: $R \leftarrow (R \setminus \{(rid,X)\}) \cup \{(rid,(X \setminus \{(tid,(u,v,p),\perp,0,\perp)\}) \cup \{(tid,(u,v,p),$
        $\mathsf{invalid},0,\perp)\})\}$
      ∗ return 1
  – else: return 0
```

Figure 10.3: Definitions of oracles for entity creation, and issuance of PTRs and rating tokens for experiment $\mathbf{Exp}^{\text{PRS-anon}}$

- $\mathcal{O}_{\text{Rate:}C}$ without arguments:
  - run protocol Rate playing the ratee on input (pp,usk^*); the adversary plays the rater's part
- $\mathcal{O}_{\text{Rate:}H}$ with arguments (rid,tid,msg):
  - if $\exists(rid,X)\in R$ for some X and $\exists(tid,(u,v,p),r',0,\bot)\in X$ for some u, v and p and $r'\neq\mathsf{invalid}$:
    - run protocol Rate playing the rater, taking (pp,u,r',msg) as input; the adversary plays the ratee's part; let r be the rater's output
    - $R\leftarrow(R\setminus\{(rid,X)\})\cup\{(rid,(X\setminus\{(tid,(u,v,p),r',0,\bot)\})\cup\{(tid,(u,v,p),r',r',1,r)\})\}$
    - return 1
  - return 0

Figure 10.4: Definitions of rating oracles for experiment $\mathbf{Exp}^{\text{PRS-anon}}$

of blocks is a hash chain if and only if the predecessor relation totally orders the set. If the hash function used to compute block hashes is collision resistant and the block hash of a new block is computed relative to the most recent block, then, except with negligible probability, the set of blocks is totally ordered.

As stated above, we use signatures to restrict who is able to create new blocks. For that, the first block in a chain contains a signature verification key as its data and omits the block hash, but retains the block signature. Thus, the first block is a self-signed certificate. Omitting the block hash gets rid of the need for a predecessor.

By requiring the first block of a chain to be a self-signed certificate, we can define the notion of *verification* of a set of blocks. Namely, we consider a set of blocks to be a *valid* hash chain if and only if

1. the set of blocks contains exactly one self-signed certificate,
2. the set is totally ordered by the predecessor relation, and
3. all blocks' signatures are valid under the signature verification key from the self-signed certificate.

We now define the Ξ hash chain scheme from hash function Hash and signature scheme Σ. Then, we discuss its algorithms and properties.

Construction 45. *The Ξ hash chain scheme consists of algorithms* (Setup, Init, Append, Verify *shown in Figure 10.5.*

The Setup algorithm chooses and outputs a hash function key. The Init algorithm establishes a new hash chain based on a hash function key output by Setup, so multiple chains can share the same hash function key; the algorithm outputs the self-signed certificate and the corresponding singing key. Algorithm Append computes a new block as a successor of the most recent block (denoted as $\max ESet$, where $ESet$ is a set of blocks). Algorithm Verify verifies the set $ESet$ of blocks as outlined above and outputs 1 if and only if verification finds $ESet$ to be a valid hash chain with self signed certificate $\min ESet$.

Hash chains have interesting security properties. Particularly, they satisfy the membership security, append-only security, and fork consistency properties for authenticated data structures, c.f. Section 3.5 for the terminology and definitions, albeit for a different data

```
$\underline{\mathsf{Setup}(1^\lambda)}$
• $hfk \leftarrow \mathsf{Hash.Gen}(1^\lambda)$
• output $hfk$

$\underline{\mathsf{Init}(hfk)}$
• $(sk,vk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
• $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk,\mathsf{Hash.Eval}(vk))$
• $ESet \leftarrow \{(vk,\sigma)\}$
• output $(sk,ESet)$

$\underline{\mathsf{Append}(hfk,sk,data,ESet)}$
• let $e \leftarrow \max ESet$
• $h \leftarrow \mathsf{Hash.Eval}(hfk,\langle e,data\rangle)$
• $\sigma \leftarrow \Sigma.\mathsf{Sign}(sk,h)$
• $ESet \leftarrow ESet \cup \{(data,h,\sigma)\}$
• output $ESet$

$\underline{\mathsf{Verify}(hfk,ESet)}$
• let $(vk,\sigma) \leftarrow \min ESet$
• if $\Sigma.\mathsf{Verify}(vk,\mathsf{Hash.Eval}(hfk,vk)),\sigma) \neq 1$: output 0 and quit
• for $e=(d,h,s) \in ESet \setminus \{(vk,\sigma)\}$ in ascending order:
    – let $e'$ be $e$'s predecessor
    – if $\mathsf{Hash.Eval}(hfk,\langle e',d\rangle) \neq h$ or $\Sigma.\mathsf{Verify}(vk,h,s) \neq 1$: output 0 and quit
• output 1
```

Figure 10.5: Algorithms of the Ξ hash chain scheme constructed from hash function Hash and signature scheme Σ

structure. The security properties of hash chains are due to the collision resistance of the underlying hash function. Taking the latest block on a hash chain as the chain's digest and the number of blocks on a hash chain as the chain's version number, the set of blocks serves as the chain's membership and append-only proofs. Therefore, hash chain proofs are not concise.

10.2.2 Personal Reputation Systems from Hash Chains

We now construct our personal reputation system, PRS, from the Ξ hash chain scheme, hash function Hash, signature scheme Σ, and commitment scheme Γ. Note that the hash function and signature scheme are as in the construction of Ξ, so we can use the hash function key and signing keys from hash chains for other purposes as well.

Construction 46. *The* PRS *personal reputation system consists of algorithms and protocols* ($\mathsf{GlobalSetup}, \mathsf{RateeSetup}, \mathsf{PlatformSetup}, \mathsf{IssuePTR}, \mathsf{IssueRT}, \mathsf{Rate}, \mathsf{Verify}$) *shown in Figures 10.6 and 10.7.*

The PRS scheme's $\mathsf{GlobalSetup}$ algorithm sets up parameters for the commitment and hash chain schemes. Algorithm $\mathsf{RateeSetup}$ initializes a hash chain from global parameters. The hash chain's public set *ESet* of blocks is complemented by a public set *RSet* that stores the actual ratings received by the ratee. Algorithm $\mathsf{PlatformSetup}$ chooses a signing–signature verification key pair for the platform.

Operation $\mathsf{IssuePTR}$ has the rater choose a signing–signature verification key pair. The platform issues a certificate on the chosen verification key. The certificate also considers the ratee that will eventually output the rating token that corresponds to the PTR output by this operation. This is achieved by making the platform's signature in the certificate dependent on the self-signed certificate of the ratee. Since the certificate's signature can be verified under the platform's verification key, the certificate ties together all three parties that participate in the transaction identified by *tid*. The rater's PTR consists of a secret and a public part; the secret part is the rater's signing key, the public part is the rater's verification key and the corresponding certificate, as well as the platform's verification key and the transaction number.

Operation $\mathsf{IssueRT}$ has the rater request a certificate on her signature verification key under the ratee's signing key, the key that the ratee also uses to sign blocks for her hash chain. The rater's rating token then consists of her previously obtained PTR and the certificate. The rater and the ratee engage in an execution of protocol Rate in order to register the issuance of the rating token with the ratee's hash chain; an empty (unprocessed) rating is used for that purpose.

Operation Rate has the rater compute and sign the data for a new block in the ratee's hash chain. The new block's data includes a commitment on the rater's unprocessed rating. When computing the signature, the rater considers the commitment and a hash of the most recent block from the hash chain, assuming that the computed data will end up in the successor block of the block whose hash has been signed. The ratee appends a block with the given data to her hash chain. After the rater has verified that the block has indeed been appended to the ratee's hash chain, she sends her processed rating to the ratee. The processed rating consists of the public part of the rater's PTR and the decommit value for the block's commitment, including the unprocessed rating. The ratee then includes the received processed rating in her *RSet*.

In operation Verify, the verifier verifies the given ratee's hash chain and makes sure that the given rating r occurs in the ratee's *RSet*. The verifier checks that there is a hash

```
GlobalSetup($1^\lambda$)
• $cpp \leftarrow \Gamma.\mathsf{Setup}(1^\lambda)$
• $hfk \leftarrow \Xi.\mathsf{Setup}(1^\lambda)$
• $pp \leftarrow (cpp, hfk)$

RateeSetup($pp$)
• $(usk, ESet_{uvk}) \leftarrow \Xi.\mathsf{Init}(hfk)$
• $RSet_{uvk} \leftarrow \emptyset$
• publish $(ESet_{uvk}, RSet_{uvk})$
• output $usk$

PlatformSetup($pp$)
• $(psk, pvk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
• publish $pvk$
• output $psk$

IssuePTR($pp, pvk | pp, psk, tid, uvk$)
Rr: $(rsk, rvk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$
Rr: $\sigma \leftarrow \Sigma.\mathsf{Sign}(rsk, \mathsf{Hash.Eval}(hfk, rvk))$
Rr: send $(rvk, \sigma)$ to platform
Pr: if $\Sigma.\mathsf{Verify}(rvk, \mathsf{Hash.Eval}(hfk, rvk), \sigma) \neq 1$: abort protocol
P: $e_0 \leftarrow \min ESet_{uvk}$
P: $t \leftarrow \Sigma.\mathsf{Sign}(psk, \mathsf{Hash.Eval}(hfk, \langle tid, e_0, rvk \rangle))$
P: send $ptr' \leftarrow (pvk, uvk, rvk, tid, t)$ to rater
P: $ptr \leftarrow (rsk, ptr')$
P: output $ptr$

IssueRT($pp, uvk, ptr | pp, usk$)
Rr: $\sigma \leftarrow \Sigma.\mathsf{Sign}(rsk, \mathsf{Hash.Eval}(hfk, rvk)$
Rr: send $(rvk, \sigma)$ to ratee
Re: if $\Sigma.\mathsf{Verify}(rvk, \mathsf{Hash.Eval}(hfk, rvk), \sigma) \neq 1$: abort protocol
Re: $crt \leftarrow \Sigma.\mathsf{Sign}(usk, \mathsf{Hash.Eval}(hfk, rvk))$
Re: send $crt$ to rater
Rr: $rt \leftarrow (rsk, ptr', crt)$
Rr&Re: $(r | r) \leftarrow \mathsf{Rate}(pp, uvk, rt, \perp, pp, usk)$
Rr: output $rt$
```

Figure 10.6: Algorithms GlobalSetup, RateeSetup and PlatformSetup, and protocols IssuePTR and IssueRT of our personal reputation system

```
$\underline{\mathsf{Rate}(pp,uvk,rt,msg|pp,usk)}$
Rr: if $\Xi.\mathsf{Verify}(hfk,ESet_{uvk})\neq 1$: abort protocol
Rr: $e\leftarrow \max ESet_{uvk}$
Rr: $(c,d)\leftarrow \Gamma.\mathsf{Commit}(cpp,msg)$
Rr: $data\leftarrow (rvk,ptr',crt,c)$
Rr: $h\leftarrow \mathsf{Hash.Eval}(hfk,\langle e,data\rangle)$
Rr: $\sigma\leftarrow \Sigma.\mathsf{Sign}(rsk,h)$
Rr: send $(data,h,\sigma)$ to ratee
Re: if $\mathsf{Hash.Eval}(hfk,\langle \max ESet_{uvk},data\rangle)\neq h$ or $\Sigma.\mathsf{Verify}(rvk,h,\sigma)\neq 1$: abort protocol
Re: $data'\leftarrow (rvk,ptr',crt,c,h,\sigma)$
Re: $\Xi.\mathsf{Append}(hfk,usk,data',ESet_{uvk})$
Re: send “data appended” to rater
Rr: $(d',h',s')\leftarrow \max ESet_{uvk}$
Rr: if $d'\neq (rvk,ptr',crt,c,h,\sigma)$, $h'\neq \mathsf{Hash.Eval}(hfk,\langle e',d'\rangle)$, or $\Sigma.\mathsf{Verify}(uvk,h',s')\neq 1$:
    abort protocol
Rr: $r\leftarrow (ptr',d,msg)$
Rr: send $r$ to ratee
Rr: output $r$
Re: $RSet_{uvk}\leftarrow RSet_{uvk}\cup\{r\}$
Re: publish $RSet_{uvk}$
Re: output $r$

$\underline{\mathsf{Verify}(pp,uvk,r,pvk)}$
• if $\Xi.\mathsf{Verify}(hfk,ESet_{uvk})\neq 1$ or $r\notin RSet_{uvk}$ or $pvk\neq r.ptr'.pvk$ or $uvk\neq r,ptr',uvk$:
  output 0 and quit
• let $e_0\leftarrow \min ESet_{uvk}$
• if $\Sigma.\mathsf{Verify}(pvk,\mathsf{Hash.Eval}(hfk,\langle ptr'.tid,e_0,ptr'.rvk\rangle))\neq 1$: output 0 and quit
• if more than two or less than two entries of $ESet_{uvk}$ contain $r.ptr'$: output 0 and quit
• let $e$ be the maximal entry in $ESet_{uvk}$ that contains $r.ptr'$
• parse $e$ as $((rvk,ptr',crt,c,h,\sigma),h',\sigma')$
• let $e'$ be $e$'s predecessor
• if $\Sigma.\mathsf{Verify}(uvk,\mathsf{Hash.Eval}(hfk,rvk),crt)\neq 1$, $\Sigma.\mathsf{Verify}(pvk,\mathsf{Hash.Eval}(hfk,\langle ptr'.tid,e_0,$
  $rvk\rangle),ptr'.t)\neq 1$, $\mathsf{Hash.Eval}(hfk,\langle e',(rvk,ptr',crt,c)\rangle)\neq h$, $\Sigma.\mathsf{Verify}(rvk,h,\sigma)\neq 1$, or
  $\Gamma.\mathsf{Open}(c,d)\neq 1$: output 0 and quit
• output 1
```

Figure 10.7: Protocol Rate and algorithm Verify of our personal reputation system

chain entry for the rating, that the hash chain entry's commitment is a commitment on the rating, and that the various signatures encountered during the tests are valid.

Correctness. The conceptual simplicity of our PRS scheme makes it easy to see that the scheme is correct. Correctness boils down to correctness of signatures and commitments, as well as our hash chain scheme Ξ ensuring that, except with negligible probability, blocks are totally ordered by the hash function and the set of blocks on a chain containing exactly one self-signed certificate.

Efficiency. Most parts of our algorithms and protocols are fairly efficient, typically consisting of computing or verifying one or two keys, signatures, and hashes. The least efficient part of the PRS scheme is the $\Xi.\mathsf{Verify}$ operation that is called as a subroutine during rating computation and verification. The operation's inefficiency is caused by it requiring time at least linear in the number of blocks on the chain, which is about twice the number of ratings that the ratee has obtained.

Security. Our PRS scheme satisfies both our security notions, rating unforgeability and rater anonymity towards the general public. Rater anonymity towards the general public is achieved, because the rater identities (raters' signature verification keys) used within our reputation system are used for a single transaction. A new rater identity is generated for every transaction, as can be seen in operation IssuePTR. Unless the transaction identifiers or contents of ratings enable the public to lift raters' anonymity or allow linking transactions involving the same rater, the general public is unable to identify raters, and this holds even in an information theoretic sense. Thus, the following lemma is established.

Lemma 47. *The* PRS *scheme provides rater anonymity towards the general public.*

Hence, there is no way for any outside party to link two ratings, or to identify the rater for a transaction unless any of the parties involved in the transaction decides to lift the rater's anonymity.

Lemma 48. *If our* PRS *scheme is instantiated with a collision resistant hash function* Hash*, an* EUF-CMA*-secure signature scheme* Σ*, and a binding commitment scheme* Γ*, then* PRS *provides rating unforgeability.*

Proof. Assume to the contrary of the lemma that there is a PPT adversary that breaks our reputation system's rating unforgeability. Such an adversary outputs a tuple (uvk, r), such that $\mathsf{Verify}(pp, uvk, r, pvk) = 1$ for verification key pvk of one of the honest platforms controlled by the experiment, and none of the honest users has participated in an execution of the Rate protocol that resulted in r.

Looking at the verification algorithm and the origins of components considered by the verification algorithm, there are only two components that are not under full adversarial control, namely, $ptr' = (pvk, rvk, tid, t)$, which is part of r, and the data $(rvk, ptr', c, h, \sigma)$ from the most recent hash chain block that contains ptr'. Both components contain a signature under a signing key that is not given to the adversary.

We now prove that every adversary that forges a rating forges at least one of the signatures t from ptr' or σ from the hash chain block data. From the fact that rating verification succeeds under an honest platform's verification key, and the fact that, in the unforgeability experiment, honest platforms only hand out PTRs to honest raters, we know that the rater's verification key rvk from ptr' belongs to an honest rater, or the adversary has forged the signature t from ptr', which is a signature on rvk under the platform's verification key.

If the adversary did not forge a signature under t, then signature verification guarantees that ptr' originates from an execution of protocol $\mathsf{IssuePTR}$ between an honest platform and an honest rater. This means that the rvk component from ptr' is the signature verification key used to verify signature σ, and the signing key that corresponds to rvk is held by an honest rater. Since no honest rater has performed a rating that resulted in $r = (ptr', d, msg)$ and (d, msg) is the open value for the commitment c on which σ is a signature, σ is a forgery.

In either case, the adversary has created a forgery under an honest party's signature verification key. However, the security of Σ, Hash and Γ and the combination of these primitives in accordance with the provably secure hash–then–sign and commit–then–sign approaches imply that PPT adversaries succeed in forging signatures with probabilities at most negligible in the security parameter. As a consequence, PPT adversaries succeed in breaking PRS's rating unforgeability only with negligible probability. $\square$

More security. The literature on reputation systems typically impose additional security properties on reputation systems. Popular examples include that

1. ratings are bound to transactions,
2. repeated rating for the same transaction can be detected,
3. misbehaving raters can be identified, and
4. self-rating is prevented.

Our notion of PTRs covers Notions 1–3: a PTR binds a transaction to all ratings for that transaction, as well as to the one-time identity of the respective rater. By considering outside processes, namely, processes of the platform, given a PTR and its transaction identifier tid, the platform can identify a misbehaving rater.

Concerning Notion 4 (*prevention of self-rating*) it is impossible to guarantee that the sets of raters and ratees are disjoint. After all, it is possible for a crowd worker to communicate a task via a platform and solve the task as a crowd worker on that platform. However, our use of PTRs and the rating unforgeability property, together ensure that a ratee buying from themselves (via some platform) is essentially the only scenario in which self-rating can occur.

Thus, the ratio of costs for obtaining the rating and benefits from the rating have to be considered. The benefits of a rating are highly dependent on the type of rating, for example, whether a 5-star scale is used or ratings are textual. Both types of ratings are supported by our construction, and different raters and platforms may have different guidelines and preferences for the type of ratings they create or issue PTRs for. This heterogeneity has to be sorted out by reputation evaluation functions, which are researched in economics. Grishpoun et al.'s CCR [Gri+09], briefly discussed in Chapter 11, also provides solutions.

Admittedly, in our PRS scheme, a malicious ratee could even set up her own platform, buying from herself via that platform, so the ratee could create PTRs themselves and thus minimize the cost of self-rating. However, reputation evaluation functions should also consider the source of the PTR for a rating when evaluating that rating: the confidence in ratings for transactions brokered by some obscure platform no one has ever heard of should be fairly low, in turn minimizing the benefit of self-rating. This is one of the features of CCR [Gri+09].

A security property typically not considered in the literature that must be considered with regard to our construction is the *possibility of rating deletion.* Since ratees have full control over their *ESet*s and *RSet*s, they can try to delete unfavorable ratings. However, due to our use of hash chains, a ratee can only delete the newest block from a hash chain, although the then-newest entry may then be deleted as well. Malicious ratees are effectively restricted to such truncating of their hash chains, because deleting any other block from the *ESet* results in one of the blocks loosing its predecessor. Then, the predecessor relation does not totally order the set of blocks, and thus, the chain would not be verifiable any more.

A ratee also must consider whether deleting an unfavorable old rating offsets the cost of deleting all newer ratings as well. One may also consider the deletion of entries from a ratee's *RSet* as an additional or alternative method of deleting unfavorable ratings, but huge discrepancies in cardinalities of a ratee's *ESet* and *RSet* should raise alarms at the verifier.

It must also be noted that raters obtain a receipt for their rating before they send their actual rating to the ratees; the receipt is the hash chain block ratees add to their chains in order to register the rating event with the chains. Raters obtaining these receipts is the reason for our construction using commitments to hide the actual reviews.

Ratings as well as receipts can be used by raters to check at any time that their ratings are included in a ratee's *ESet* and *RSet*, providing a means for the detection of rating deletion. Once rating deletion is detected, raters can take appropriate action, although such action is outside the scope of our personal reputation system. For example, ratees may be reported to platforms or legal action may be taken. Of course, detection of rating deletion by ratees requires raters to occasionally check that their ratings are still present in the hash chains of the respective ratees.

We particularly note that the possibility of rating deletion and the detection mechanism for rating deletion are independent of the authenticated data structure used to implement ratee's *ESet*s. With every data structure, a ratee can revert the data structure to an earlier state; this is a consequence of ratees having full control over their respective *ESet*.

10.3 Discussion

We conclude our discussion of personal reputation by discussing some of the shortcomings and drawbacks of our PRS construction.

Improving efficiency. One major shortcoming of the PRS scheme is the need for verifying a hash chain during Rate and Verify operations. Hash chain verification takes time at least linear in the number of blocks on the chain, and needs to transmit the whole chain to the chain's verifier, resulting in bandwidth use proportional to the size of the hash chain. This is because the hash chain is an authenticated data structure that has no concise membership proofs. Yet, hash chains are fork consistent, append-only and membership secure.

In the construction of PRS, hash chains may be replaced by other fork consistent authenticated data structures that have concise membership proofs. However, we expect replacing hash chains to provide no new insights. Replacing hash chains would require no significant changes — one minor change would be the need for signing key generation to happen explicitly during RateeSetup, rather than implicitly as part of the hash chain's Init operation, as well as ratees explicitly signing data before appending it to the authenticated data structure. On the other hand, security would not be affected by replacing hash chains for another fork consistent, append-only and membership secure authenticated data structure.

Improving practicability. From a practical point of view, it is problematic that our construction requires ratees to be online constantly to serve requests for their *ESet*s and *RSet*s, so others may verify the raters' hash chains. This is a consequence of the decentralization that we aim for.

In practice, it is likely for a cloud service to address the technical aspects of the ratee's role, resulting in a middle ground between semi-centralized storage of reputation by platforms and the implied vendor lock-in, the decentralized approach that requires ratees to concern themselves with technical details of a personal reputation system, and the semi-centralized storage of reputation of many ratees by few specialized cloud services. Such services themselves may result in vendor lock-in, just at a different vendor, namely, the cloud service provider, and with higher stakes, namely, a ratee's reputation aggregated from multiple platforms. However, as long as ratees keep a copy of their signing keys and a copy of their hash chain, vendor lock-in with respect to cloud services can be prevented. Given their copies, raters can take on the technical role of ratees for themselves without having to fear loss of reputation.

11 State of the Art

In the cryptographic literature on reputation systems, the focus is often on reputation systems for single platforms, for example, online shops or peer-to-peer systems. Nevertheless, cross-platform reputation has also been considered. For instance, due to the cross-platform aspect, a user's reputation in one online forum affects her reputation in another online forum. Single platform reputation systems have been considered in [And+08; BEJ18; BJK15; CSK13; HBC15; IJ02; KSG03]. Cross-platform reputation has been considered in [DO15; Gri+09; PS08].

Cross-platform reputation. The work of Grinshpoun et al. [Gri+09] focuses on exchanging reputation scores among communities. This is achieved via the cross-community reputation system CCR, providing mechanisms for translating scores between scales, like 3-star scale to 5-star scale, as well as weighing reputation based on relevance and trust. For example, an online forum focussing on automobiles may see some limited relevance in reputation scores imported from a discussion forum focussing on vehicle engines.

Pingel and Steinbrecher [PS08] also address cross-platform reputation. They focus on aggregating reputation scores in a secure manner, maintaining anonymity of rated entities and even unlinkability of entities' pseudonyms used on different platforms. In Pingel's and Steinbrecher's approach, rated entities store their aggregated reputation scores. Having rated entities store their own reputation is seen as a means to enhance data privacy by granting entities full control over their reputation scores.

Dennis and Owen [DO15] address the problem of storage of reputation and argue for decentralized storage, rather than storage of reputation scores on a single centralized server (or one server per community). They suggest to realize decentralized storage via distributed ledgers, specifically block chain. In particular, Dennis and Owen consider integrating their reputation system with the Bitcoin crypto-currency.

We use aspects of the mentioned cross-platform reputation systems for our PRS scheme. At the same time, we go beyond the above proposals or avoid some of their shortcomings. Like Pingel and Steinbrecher, we have rated entities store their own reputation, but do not limit ourselves to reputation that can be aggregated. Instead, we gather reputation of any kind (5-star scale, textual, ...) rather than aggregating it to some value. Due to our construction supporting essentially all imaginable ways of expressing reputation, we need to rely on translation between different forms of feedback, as proposed for Grinshpoun et al.'s CCR. Finally, on the technical side, with our use of hash chains, we use techniques from block chains. However, in contrast to Dennis and Owen, due to the significant ecological impact of consensus mechanism like proofs-of-work typically seen as part of block chains, we put away with consensus mechanisms altogether.

Hash chains and the crowd. Our approach of applying hash chains to crowd sourcing and crowd work is not unique in the cryptographic and security literature. However, typically, block chains, the crypto-currency variety of hash chains, are applied, because the security of the proposals rely on the integrated consensus mechanisms in block chains. As an example, Li et al. [Li+19] propose to decentralize entire platforms by realizing them via block chain. In this setting, the platforms' processes are implemented as smart contracts that are executed on the chain.

Another example application for hash chains in crowd sourcing is crowd decision making. Ast and Sewrjugin [AS] suggest using a block chain for private court adjudication. The chain would be used to register and research evidence, and to reach verdicts and document them.

Finally, Buccafurri et al. [Buc+17] recognize that the crypto-currency variety of hash chains is not suited to be applied to crowd sourcing. This is because block chains rely on miners to verify the chain and compute new blocks that contain a list of recent transactions as data. In order to maintain high throughput, new blocks that elongate the chain need to be computed as fast as possible. For that, miners are financially incentivized to compute blocks, for example, by transaction fees. Buccafurri et al. argue that the fees from transactions related to crowd sourcing are too low for miners to consider registering such transactions with the chain via new blocks, because competing transactions would result in higher fees. Therefore, Buccafurri et al. propose to establish a hash chain for crowd sourcing applications on top of a social network, they suggest Twitter, rather than on top of a block chain.

Part IV

Cloud Architectures

12 Introduction to Cryptographic Cloud Architectures

Cloud computing is a paradigm of outsourcing computation and data storage to a remote location, onto devices owned by cloud service providers. A trademark characteristic of cloud computing is that resources can be shared between the various customers of the cloud service provider, and that resources can be allotted to customers based on their current demand. As a consequence of resources being shared and provided on demand, cloud computing is often advertised as being cheaper than the outsourcer owning and operating the resources on their own.

However, while outsourcing computation and data storage may have cost benefits, the security implications of cloud computing must also be considered. Typical questions concern the trustworthiness of the cloud service provider, proper separation of data between the provider's clients, and security against outside threats. Particularly concerning data storage, cloud computing guidelines call for the encryption of data before storage [Clo11]. However, once data has been encrypted, it generally cannot be operated upon in a meaningful way, unless it is either decrypted or the encryption scheme is compatible with the desired operation. An example of the latter case is searchable encryption [SWP00], c.f. Part II. Depending on who performs encryption, encryption can only prevent outside attackers form accessing data or may even stop inside attackers like malicious server administrators at the cloud service provider. The former is achieved if the cloud service provider performs encryption; the latter is achieved if the data is encrypted by its owner. Who performs encryption has additional security implications due to the necessary key management [Clo12]. For example, if encryption is performed by the cloud service provider, there must be provisions for the data owner to access encryption keys, and thus their data, without the cloud service provider's involvement. Such provisions are necessary to ensure data owners can still access their data in case the cloud service provider goes out of business.

As mentioned, who encrypts the data determines what threats are eliminated by encryption. Encryption by data owners eliminates threats from outsiders as well as from insiders at the cloud service provider, whereas encryption by the cloud service provider only prevents attacks from the outside. Thus, encryption by data owners achieves stronger security, yet does not take full advantage of the cloud, because data owners still need to provide their own resources to perform encryption. If we restrict ourselves to these naïve approaches to encryption in cloud computing, we have to make the trade-off between strong security and extensive cloud usage. However, concerning encryption, a middle ground can be covered by a "trusted cloud" that mediates between the data owner and the public cloud. Despite the trusted cloud being labeled "trusted," there are multiple ways to achieve trustworthiness of the cloud service provider ranging from relying on the providers good will via service level agreements and organizational measures to technical and cryptographic measures.

Our contribution. Our main contribution is an architecture for the cloud that employs a trusted cloud. While the idea of a trusted cloud is not new, we present a novel generic approach to securing computation in the trusted cloud. For that, we introduce the cryptographic computation service (CCS), which we bundle with hardware security to serve as the trust anchor in the trusted cloud. Due to the CCS, the cloud is capable of efficiently and securely storing and computing on data in the cloud.

We have presented the architecture and an early version of the CCS at FPS 2017 [Blö+17]. The early version was targeted specifically at enforcing access restrictions to data stored in the cloud. An improved version of the CCS has been presented at ARES 2018 [BL18]. The improvement allows for computations other than enforcement of access control to be also performed in the cloud. In this thesis, we present a version of the CCS that, in contrast to the ARES 2018 result, does not distinguish between computations related to the enforcement of access control and other computations.

Overview of Part IV. In the upcoming chapter, we first present our cloud architecture, and then discuss the CCS. Our presentation particularly focuses on architectural aspects of the trusted cloud and on illustrating the versatility of our cloud due to the CCS. To that end, we apply the CCS in several example scenarios. In Chapter 14, we discuss related work on cloud architectures and technologies for secure computation and storage in cloud environments.

13 A Cloud Architecture for Cryptography in the Cloud

Inspired by the work of Green et al. on outsourced decryption for ABE [GHW11], we have developed a cloud architecture that enables organizations to outsource their cryptographic computations to the cloud without needing to fully trust the cloud service provider. Due to our architecture, if a breach of security occurs, cloud service providers operating according to our architecture can plausibly deny wrongdoing on their part.

13.1 Our Cloud Architecture

We have proposed a preliminary version of our BGKL architecture in [Blö+17]. The architecture features three parts, located at the organization, in a *trusted cloud*, and in the public cloud. Trustworthiness of the trusted cloud is achieved by various technical and organizational means; particularly the trusted cloud features hardware security modules (HSMs) that serve as trust anchors and are used for operations that are highly security critical. While HSMs are highly secure, a drawback in them is their low throughput. A measure taken to balance security and efficiency in the trusted cloud is to have a trusted (compute) server in addition to HSMs. Trusted compute servers provides the computational power the HSMs lack at the cost of some degree of security. As a consequence, HSMs and trusted compute servers operate on distinct sets of cryptographic keys and data. The trusted server operates on keys and data having limited security implications. In contrast, keys and data with strong security implications are handled by HSMs.

Our setup involves a public cloud, because the measures taken to achieve security in the trusted cloud are somewhat financially costly and impact efficiency, so the public cloud operates on data not critical for security. Using the public cloud for such operations lowers costs and improves efficiency — when compared to computations solely in the trusted cloud — without affecting security.

Our original proposal of the BGKL architecture was aimed at organizations outsourcing file storage and access control to the cloud. Particularly, the organization was not to be involved in enforcing access control beyond assigning the access rights of the organization's members. Within the BGKL architecture, file storage and access control are achieved by applying ABE. The organization holds its members' user keys in an organizational key store (OKS). Whenever one of the organization's members wants to access a file, she does so via a service that runs at a trusted server in the trusted cloud, and the trusted cloud requests the member's key from the OKS. The trusted cloud then obtains the (encrypted) requested file from the public cloud, and decrypts the file using the member's key. Since we use hybrid ABE, the member's user key has strong security implications. Specifically, the user key can decrypt all files the member has access to, so if the key leaks, many files are leaked. In contrast, fresh symmetric keys are used for individual files, so if a symmetric key leaks, only one file is leaked. Therefore, we consider user keys as having strong security implications and symmetric keys as having limited security implications. As a consequence, during decryption in the original use of the BGKL architecture, the asymmetric part of hybrid decryption is performed by an HSM, while the symmetric part is performed by a trusted server. After decryption, the resultant plaintext is sent to the organization's member.

Since the original version of our architecture was developed with this usage scenario in mind, the BGKL architecture has some drawbacks that prevent it from being used in combination with some other cryptographic primitives. We have addressed those shortcomings in follow-up work [BL18]. The work aims at making the architecture compatible with cryptographic primitives beyond (attribute-based) encryption. Specifically, while the original work on the BGKL architecture aims at storage of and access control to files, the follow-up work aims at making stored documents searchable, c.f. searchable encryption in Part II.

Due to the relation between organization's members and the trusted cloud, searchable encryption, particularly verifiable searchable encryption, could not easily be combined with the BGKL architecture. This is because services running in the trusted cloud take on the roles of users with respect to the public cloud. However, one of the measures taken to make the trusted cloud trustworthy is that it lacks persistent memory, which in turn means that the trusted cloud can only play the users' parts of cryptographic primitives that do not require users to permanently hold states. This is why verifiable searchable encryption, which necessitates users to keep states, is incompatible with the original BGKL architecture. The lack of support for stateful primitives is one of the drawbacks identified in our follow-up work on the BGKL architecture.

Furthermore, as a relic of the BGKL architecture being tailored to a specific use case, adding a means for storing user states is a non-trivial task. Specifically, the HSMs should be involved in storing user states in some capacity due to security reasons. However, in the BGKL architecture, HSMs are tailored to ABE. Due to security reasons, HSMs do not admit general computation,[4] so expanding their operations is not an option. The means used to achieve HSM security also make HSMs very expensive, so putting HSMs tailored to storing user states in the trusted cloud also is not an option. Following the approach of specialized HSMs would also mean that specialized HSMs for doing some computations for searchable encryption would be needed if we were to adapt searchable encryption to the original BGKL architecture.

Since the approach of inflating the number of HSMs in the trusted cloud is not an option due to economics, in [BL18], we propose to put away with specialized HSMs altogether. Instead, we introduce a general service, called cryptographic computation service (CCS) that can perform broad classes of cryptographic computations shared as building blocks by many cryptographic schemes. Examples of building blocks can be found at the start of Chapter 3, particularly Section 3.1. The cryptographic schemes presented throughout this thesis demonstrate why a service providing access to the basic cryptographic building blocks in a secure way is useful. The CCS runs at all HSMs in our architecture.

This allows us to construct stateful services. Specifically, the CCS allows us to restore

[4]General computation, although feasible in HSMs, prevents HSMs to successfully pass certification processes. Having passed certification is often necessary for HSMs to be used, for example, in the financial service industry.

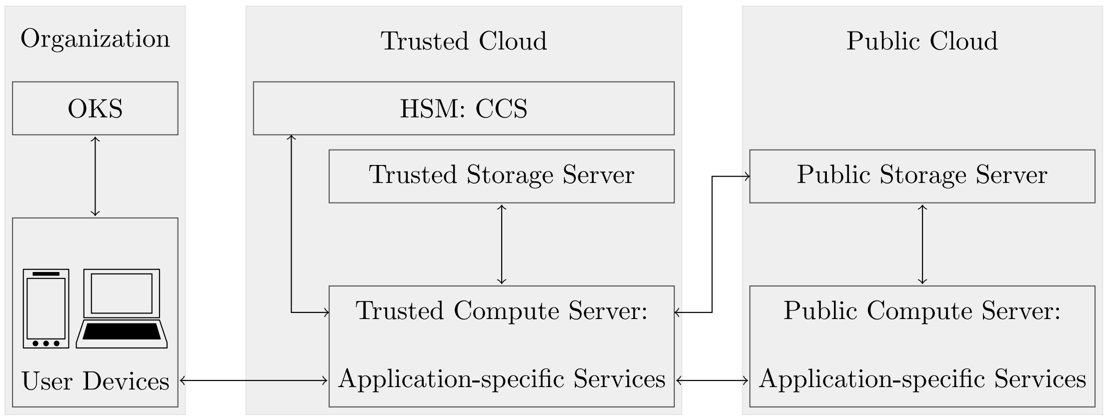


Figure 13.1: Overview of the improved BGKL architecture: entities at the organization and in the cloud, and communication between them

services' states; we illustrate this further below. With stateful services, we also need a way of storing services' states persistently. Due to costs, it is preferable for services running in the trusted cloud to store their states in the public cloud. However, the availability of the public cloud may not be given at all times, so at least some type of emergency storage needs to be present in the trusted cloud, for example, to cover non-availability of the public cloud due to power outages. Therefore, we allow the trusted cloud to provide some persistent memory in order to mitigate issues with the trusted cloud communicating with the public cloud.

As a consequence, we have revised the BGKL cloud architecture. The resulting improved architecture is shown in Figure 13.1. The figure shows the three parts of the architecture, the entities based in these parts, and communication between entities. Particularly, we can identify the central role that the trusted compute servers play in communication. The servers ensure that the organization and the public cloud do not need to be aware of each other or of the (other) entities located in the trusted cloud.

Unfortunately, Figure 13.1 only illustrates how the CCS, run at HSMs, is integrated in our architecture. However, the figure does not illustrate how the CCS achieves the properties and power ascribed to it. We address the CCS in detail in the next section.

13.2 The Cryptographic Heart of our Architecture

As stated before, the cryptographic computation service (CCS) runs at HSMs. The service encapsulates the cryptographic essence of many potential specialized services that may otherwise run on HSMs within our architecture. The cryptographic essence is given in the form of fundamental building blocks, some of which are presented in Section 3.1; this includes hash functions, pseudorandom functions, and trapdoor permutations.

The CCS provides an interface for application-specific services that run at trusted compute servers to evaluate a *concrete instantiation* of such a function on a given key and argument. Examples of such instantiations are the RSA [RSA78] and DLog-based trapdoor permutations [DH76], the AES pseudorandom function [DR98], and the Keccak/SHA-3 hash function [Ber+13] that can also be used as a pseudorandom function. However, the whole reason for the use of HSMs is to achieve security by not having the compute servers operate on cryptographic keys that come with strong security implications. So the question arises, how application-specific services can communicate what keys to use to the CCS without the application-specific services learning the actual keys.

For that, we adopt the concept of *tickets* form the Kerberos system [Neu+05]. In the Kerberos system, a ticket represents a user's right to use a service. Tickets are issued by some authority and eliminate the need for the authority to directly communicate with the service. Moreover, the issuing authority can use the ticket to communicate restrictions and conditions to the service. Example restrictions are validity periods of the ticket. Meanwhile, the user is unable to learn the conditions or to modify the ticket. Kerberos achieves this by having a symmetric key shared between an authority and a service. The key is used to encrypt the ticket with a CCA-secure symmetric encryption scheme, so the service can be sure that the ticket originates from the authority and has not been modified.[5] Due to CCA-secure encryption, no party — with the exception of the authority and the service — learns the contents of the ticket, and the service can be sure that the ticket originates from the authority and has not been modified.

In our architecture, we use tickets such that the organization, represented by its OKS, can grant its members access to services running in the trusted cloud by allowing those services to make use of the organization's cryptographic keys without learning the keys. In this setting, the OKS sends a ticket to a user, the user forwards the ticket to the service she wants to use, and the service redeems the ticket at the CCS in order to get cryptographic functions evaluated. For this to work, a ticket must state what service is allowed to present the ticket to the CCS. We achieve this by requiring application-specific services to hold a signing–signature verification key pair. The verification key is then included in the ticket. When using the ticket, the application-specific service signs its request and the CCS can establish the origin and authenticity of the request.

Of course, for functional purposes, the ticket also includes a list of cryptographic keys that the CCS can use to serve services' requests and what cryptographic functions the keys can be used in conjunction with. Then, a ticket for some application-specific service X is a ciphertext

$$t_X = \mathsf{Enc}(k_{\text{shared}}, \langle vk_X, \{(id_{(f,k)}, f, k)\}_{(f,k)\text{ accessible to }X}\rangle),$$

where k_{shared} is the symmetric key shared between the organization and the CCS and vk_X is X's signature verification key with corresponding signing key sk_X. A ticket can hold many function–key pairs (f, k) represented by the tuples $(id_{(f,k)}, f, k)$. Each such tuple consists of an identifier $id_{(f,k)}$ of the function–key pair that the service X can use to refer to the pair, a function identifier f, and a key k to be used with the function. Then, a request of service X to the CCS is a tuple

$$(t_X, id_{(f,k)}, a, \sigma),$$

where $\sigma \leftarrow \mathsf{Sign}(sk_X, \langle t_X, id_{(f,k)}, a\rangle)$ is a signature on the request under the service's signing key. If $\mathsf{Dec}(k_{\text{shared}}, t_X) = \perp$ or $\mathsf{Verify}(t_X.vk_X, \langle t_X, id_{(f,k)}, a\rangle, \sigma) \neq 1$, the CCS's reply to the request is $\perp$. Otherwise, the reply is $f(k, a)$, meaning that, if the tests succeed, then the CCS evaluates function f with key k on argument a and hands over the result to the application-specific service.

Our tickets are encrypted under a symmetric key shared between an organization and the CCS. This conflicts with the desired degree of flexibility in the cloud, particularly, the ability to share resources like HSMs among multiple customers of the cloud service provider. We address this issue by introducing *master tickets.* A master ticket is a way for an organization to communicate a shared key for the CCS to an HSM, and to claim the

[5]While this description is not entirely correct, it conveys the basic idea of tickets.

```
KeyGen($1^\lambda$)
• $(p,\mathbb{G},g) \leftarrow \mathsf{GroupGenerator}(1^\lambda)$
• $(sk \leftarrow_\$ \mathbb{Z}_p$
• $h \leftarrow g^{sk}$
• $pk \leftarrow (p,\mathbb{G},g,h)$
• output $(pk,sk)$

Enc($pk,msg$)
• $r \leftarrow_\$ \mathbb{Z}_p$
• output $(C,C') = (msg \cdot h^r, g^r)$

Dec($sk,ct$)
• output $C \cdot C'^{sk}$
```

Figure 13.2: The ElGamal encryption scheme in a group $\mathbb{G}$ of prime order p with generator g; the group and its parameters are chosen by some cryptographic PPT algorithm GroupGenerator that takes the security parameter λ as input

HSM. For this, the shared key is encrypted using CCA-secure public key encryption under a public key of the targeted HSM. If an HSM receives a master ticket that it can decrypt and the HSM is not already claimed, the HSM becomes claimed and initializes the CCS to use the shared key from the master ticket. While the HSM is claimed, it rejects all further attempts to claim it. Eventually, the organization releases its claim on the HSM. Upon doing so, the HSM erases the shared key. Afterwards, the HSM can be claimed by the same or another organization.

13.3 Usage of the Cryptographic Computation Service

We have claimed that the CCS allows for application-specific services to become stateful. For that, we now demonstrate how a service can store its state in the public cloud in a secure way and later on recover the state.

In order to store information securely in the public cloud, we need to encrypt the data. Specifically, we choose the ElGamal [ElG85] public key encryption scheme for that purpose. The scheme is shown in Figure 13.2 and is known to be CPA-secure if the decisional Diffie-Hellman assumption holds in the group that the GroupGenerator algorithm outputs. We use the ElGamal scheme as part of a hybrid encryption scheme, c.f. Section 3.2.4. We denote the corresponding symmetric encryption scheme as Π. The hybrid version of the ElGamal scheme is shown in Figure 13.3.

Deriving ticket contents. In order for the hybrid scheme to be used securely in conjunction with our CCS, we need to analyze the cryptographic keys used in the scheme and how they are used, and then identify the security implications of a revelation of the respective key to the public. The ElGamal public key being public means that revelation of the key happens anyways, so there are no security implications concerning the public key. On the other hand, if the ElGamal secret key was revealed to the public, all information ever encrypted under the corresponding public key would be leaked to the public. Therefore, keeping the key private is of paramount importance and the ElGamal scheme's secret key must be kept in the HSM. The symmetric scheme's secret key is only used for an individual

```
$\mathsf{KeyGen}(1^\lambda)$
• output $\mathsf{ElGamal.KeyGen}(1^\lambda)$

$\mathsf{Enc}(pk, msg)$
• $k \leftarrow \Pi.\mathsf{KeyGen}(1^\lambda)$
• $c_0 \leftarrow \mathsf{ElGamal.Enc}(pk, k)$
• $c_1 \leftarrow \Pi.\mathsf{Enc}(k, msg)$
• output $(c_0, c_1)$

$\mathsf{Dec}(sk, ct)$
• $k \leftarrow \mathsf{ElGamal.Dec}(sk, c_0)$
• output $\Pi.\mathsf{Dec}(k, c_1)$
```

Figure 13.3: The hybrid version of the ElGamal encryption scheme with complementary symmetric encryption scheme Π

message. If a symmetric key is leaked, this has limited impact on security, because then only a single message is leaked. Thus, it is acceptable for the symmetric scheme's secret key to be used in the trusted cloud, but outside the HSM.

As a consequence of our considerations, the ticket for a stateful application-specific service that runs in the trusted cloud contains a tuple (state_key, DLog, (pk, sk)). Note that the key consists of the public and the secret key, so the CCS knows what group to operate on.

Making services stateful. When a service stores its state, it encrypts the state under key pk using hybrid ElGamal and stores the resulting ciphertext in the public cloud. When restoring a state, the service obtains the encrypted state from the public cloud and runs the decryption procedure from hybrid ElGamal on the ciphertext. The only instruction the service is unable to perform on its own is the computation of $(g^r)^{sk}$. This is where the service's ticket and the CCS come in. The service requests the CCS to perform the operation identified by identifier state_key from the ticket on argument g^r. All other parts of ElGamal encryption are performed outside the CCS, but in the trusted cloud. Once the service has recovered the secret key for the symmetric encryption scheme, it can decrypt the symmetric ciphertext and then restore its state from the result.

Security. Due to ElGamal decryption being performed completely within the trusted cloud, all security properties of the ElGamal scheme remain intact. Particularly, to adversaries outside the trusted cloud, our adapted ElGamal scheme, with its decryption procedure jointly performed by a trusted server and an HSM, behaves exactly like the base scheme does. Concerning adversaries inside the trusted cloud, our use of the trusted compute server ensures that PPT adversaries learn nothing about the ElGamal secret key that cannot also be learned from the public key.

Other services. The thought process of our derivation of ticket contents is not specific to the ElGamal scheme. Indeed, the original work on the BGKL architecture uses the same thought process to devise what part of an ABE scheme to run on what devices, c.f. Section 3.2 of [Blö+17]. In particular, we have adapted the CPA-secure ABE scheme of Rosuelakis and Waters [RW13] and have applied the outsourced decryption technique of

Green et al. [GHW11] to the ABE scheme. Due to Green et al.'s technique, in our original work on the BGKL architecture, we reduce the HSM's workload to some consistency checks and an exponentiation, because the most complex aspects of decryption are performed outside the HSM through "outsourced decryption." Specifically, outsourced decryption takes a public key derived from a user's secret ABE key to strip policy information off of ABE ciphertexts. The reduced ciphertext then can only be decrypted to a plaintext using a secret key that complements the public key derived from the user's ABE key.

The reduced ciphertexts the HSM operates on are triples $(T_0, T_1, T_2) \in \mathbb{G}_T \times \mathbb{G} \times \mathbb{G}_T$ for groups $\mathbb{G}$ and $\mathbb{G}_T$ of prime order p with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, and two hash functions $H_1 : \mathbb{G}_T \times \{0,1\}^\lambda \to \mathbb{Z}_p$ and $H_2 : \mathbb{G}_T \to \{0,1\}^\lambda$, where λ is the security parameter of the scheme. The private key that complements the public key derived from a user's ABE key is an element $z \in \mathbb{Z}_p$. The reduced ciphertext triples must satisfy

1. $R = T_0/T_2^z$
2. $msg = T_1 \oplus H_2(R)$
3. $s = H_1(R, msg)$
4. $T_0 = R \cdot e(g,g)^{\alpha s}$, where $e(g,g)^\alpha$ is a public parameter of the ABE scheme.

These consistency checks originate from the Fujisaki–Okamoto transform [FO13], which transforms cpa-secure asymmetric encryption schemes into cca-secure ones. Here, the Fujisaki-Okamoto transform is applied to the ABE scheme as part of outsourced decryption. If a triple satisfies these conditions, the ciphertext is decrypted by the HSM and the decryption result is the message *msg*. The message is then further processed by an application-specific service in the trusted cloud.

When adapting this scheme to work with the CCS rather than an application-specific HSM, we let the consistency checks and most computations be performed by the respective application-specific service in the trusted cloud. Then, our CCS only needs to perform the single exponentiation T_2^z. This is the same operation as in the case of ElGamal decryption, albeit in a different group, namely, the target group $\mathbb{G}_T$ of a bilinear map.

We can also apply the thought process outlined above to searchable symmetric encryption (SSE). Using the process, we have adapted a single-user SSE scheme to our (improved) cloud architecture, resulting in a multi-user searchable encryption scheme [BL18]. Specifically, we have adapted Bost's Σοφος scheme [Bos16], c.f. Section 5.5. For that scheme, the application-specific service that provides searchable encryption to organizations' members needs to rely on the CCS when formulating search queries and, in contrast to the encryption example, the CCS is used multiple times. Namely, the CCS is required to compute keyword dependent function keys and for computing the inverse of a trapdoor permutation. Further application of the CCS is required, because the Σοφος scheme requires the service that runs in the trusted cloud to be stateful: The service needs to remember for each keywords how many distinct documents contain the keyword.

The Σοφος scheme is not the only SSE scheme we have adapted to our cloud architecture. As an alternative to Σοφος, we have also adapted the scheme of Etemad et al. [Ete+18]. The adaption of Etemad et al.'s scheme can be found in an extended version of our work on the BGKL cloud architecture and our improvements [Blö+20], not yet submitted for publication.

As with the ElGamal encryption scheme, when adapting the ABE scheme and the searchable encryption scheme to be used with our CCS, concerning adversaries outside our trusted cloud, the security properties of the base schemes carry over to the adapted schemes. This is, again, because from outside attackers' views, the adapted schemes are indistinguishable from the base schemes. Also as with the ElGamal scheme, adversaries inside the trusted cloud do not learn more about the secret keys used by the CCS than what is revealed by function evaluations and tickets. However, we use a CCA-secure encryption scheme to encrypt tickets. This implies that no PPT attacker can learn any information on the secret key from the ticket. The security properties of pseudorandom functions and trapdoor permutations imply that evaluating the functions or their inverses does not allow PPT adversaries to learn any information about the secret keys involved.

13.4 Trust in the Cloud

As mentioned before, our cloud architecture and the CCS enable organizations to outsource computations and data storage to the cloud while not fully trusting the cloud service provider that performs computations or stores the data. This particularly holds with respect to the cloud service provider running our trusted cloud.

With our cloud architecture, the cloud service provider never gets access to plaintext data. This is achieved by technical and organizational means that prevent the cloud service provider from accessing data at the trusted compute server. Beyond the trusted compute server and the HSMs, data does not exist in plaintext form within out trusted cloud.

The measures taken to secure the trusted compute server take various forms. For example, there should be multiple levels of physical access restrictions to trusted compute servers, as to prevent any unauthorized access. The trusted compute servers also lack persistent memory, as to prevent anyone with authorized physical access to the servers from getting access to plaintext data. Furthermore, we can rely on remote attestation in conjunction with signatures on services' requests at the CCS to prevent impersonation attacks. Then it is hardly possible for a service that does not run at a trusted compute server to successfully redeem a service ticket at the CCS. Additionally, trusted compute servers and the CCS communicate via encrypted channels, so no plaintext data is exchanged between them.

Similarly, access to plaintext data via trusted storage servers is prevented via encryption. Services running at trusted compute servers can still access data stored at trusted storage servers by relying on the CCS for decryption, as shown above for the services' persistent states. The same mechanism can also be applied for the organization to securely transmit data to services running at trusted servers and vice versa, as well as for services running at trusted compute servers to securely store data at and retrieve data from storage servers in the public cloud.

The measures taken to make trusted compute servers trusted together with the means for secure communication and storage of data provided by the CCS make sure that plaintext data exists in the trusted cloud at only two places, namely, in trusted compute servers and in HSMs. Both of these places effectively prevent access to the plaintext data. The trusted compute servers prevent access via technical and organizational means, some of which we have outlined above. The HSMs prevent access by relying on hardware security.

We have previously mentioned that if a breach of security occurs, a cloud service provider running our trusted cloud can plausibly deny wrongdoing on their part. This holds true, because the cloud service provider does not have access to plaintext data. After all, the provider cannot leak information it does not have access to. Thus, our cloud architecture and the CCS at its heart protect organizations' data when outsourcing data storage and computation to the cloud, while also protecting the cloud service provider from being wrongly blamed for security breaches.

14 State of the Art

Non-trivial cloud architectures for secure data storage and computation have been rarely covered by the literature. The literature often assumes the naïve cloud model featuring only users and cloud services and communication features exactly one user and one cloud service.

Within that setting, secure computation and data storage in the cloud can be achieved by applying generic techniques. Popular examples of techniques that achieve arbitrary computation in a generic, yet secure way, are garbled circuits and fully homomorphic encryption. Concerning storage and communication, there are the techniques of oblivious RAM and oblivious transfer.

With *garbled circuits* [Yao86], programs are transformed into circuits that can be applied to encrypted data and the result of the application is an encryption of the result of the application of the original program to the plaintext data. A drawback of this approach is that garbled circuits can only be evaluated once for security reasons, and then have to be renewed. This makes garbled circuits expensive in terms of computation and bandwidth, because the circuits need to be computed by the user in order to be secure. As a consequence, in the simple cloud model, users still have to perform a lot of computations, that they preferably would not perform themselves.

With *fully homomorphic encryption* [Gen09], computations can be performed on homomorphically encrypted data and the result of computations again is homomorphically encrypted and suitable for further computations up to a certain threshold. After reaching the threshold, ciphertexts need to be refreshed in order to guarantee that they can eventually be decrypted, because too many computations may result in decryption failures occurring with non-negligible probability, c.f. Definition 9. It must be noted, that fully homomorphic encryption is currently too inefficient to be practical [Bug+11].

With *oblivious RAM* [GO96], it is possible to hide what memory address has been effectively accessed and what operation was performed on that address. This is achieved by reading and writing several cells for each operation, so an observer is unable to distinguish read access from write access, and is unable to determine the memory cell that holds the accessed information.

With *oblivious transfer* [Rab81], two parties can exchange information without learning what information they have shared. Specifically, one of the parties holds two data items and the other party chooses which of the data items to access. The party holding the data does not learn which of the items was accessed and the other party only learns the accessed data item, but not the other one.

Concrete techniques for secure computation relying on the simple cloud model are presented in large volumes in the literature. One example is searchable encryption [SWP00] (also see Section 5.7). Another example is the suggestion of Troncoso-Pastoriza and Pérez-González to outsource digital signal processing to the cloud by virtualizing (hardware) signal processors [TP10]. The virtualized processors, called CryptoDSPs, would then be deployed in the cloud. The processors are supposed to keep the data and the computation (program) secret. Unfortunately, the authors do not give a concrete instantiation, but only list (some of the above) generic techniques that can be used to maintain the required degree of secrecy.

In contrast to generic techniques and the simple cloud model, the works of Sadeghi et al. [SSW10] and Bugiel et al. [Bug+11] specifically use a non-trivial cloud model. Sadeghi et al. employ tamper-proof hardware to securely perform computations in the cloud. Specifically, the hardware is used to compute garbled circuits for the required computations and the circuits are then evaluated in the cloud. The hardware is given to a cloud service provider by the user that wants to outsource computations to the cloud.

Bugiel et al. pick up some open questions left in the work of Sadeghi et al. The authors address secure cloud computing, and suggest a cloud setup similar to ours, consisting of users, a trusted cloud and the public cloud. The trusted cloud is to achieve security, while the public cloud is to achieve efficiency. To that end, the trusted cloud computes garbled circuits that are evaluated in the public cloud. Bugiel et al.'s trusted cloud is to be instantiated at the user as a private cloud, negating some of the benefits of cloud computing, or on leased HSMs located in the cloud, which is problematic because HSMs are not allowed to perform arbitrary computations, c.f. Section 13.1, which would be necessary for computing circuits. However, Bugiel et al.'s trusted cloud can be implemented by our trusted cloud.

A non-trivial cloud model can also be found in multi-cloud architectures which feature cloud resources offered by multiple independent cloud service providers. In this setting, in order to keep programs or data private, one employs the generic technique of secure multi-party computation [GMW87]. Proposals that use multi-party computation in this setting can be found in the work of Loftus and Smart [LS11] and Bohli et al. [Boh+13], amongst others.

The idea of multi-party computation is for a number of parties to jointly compute a function of the parties' private inputs without revealing those inputs to the other parties. This requires a certain threshold of parties to behave honestly; typically an honest majority is required. The honesty requirement is the reason why independent cloud service providers are required in multi-cloud architectures. However, great care must be taken when choosing providers, because occasionally seemingly independent providers cooperate. For example, the German telecommunications provider Telekom offers its own "Open Telekom Cloud" [TSyb], while also hosting parts of Microsoft's "Azure" cloud [TSya].

Part V

The Future . . .

15 Bringing It All Together

In this thesis, we have presented three main research areas, namely, searchable encryption, reputation systems, and cryptographic cloud architectures. The topics are contextualized in ways that best illustrate the core purposes of the presented constructions. That is searchable, yet secure, outsourced document storage for searchable encryption, crowd work as motivation for our personal reputation system, and secure outsourcing of data storage and computation as motivation for our cloud architecture. We now apply all our constructions to crowd work, binding the constructions together in order to address issues with crowd work that cannot be addressed solely by our personal reputation system.

As stated in Chapters 1 and 9, in crowd work, the crowd invests its labor in exchange for money, in order to solve tasks proposed by a requester via an open call on the internet. In practice, the open calls are not directed to everyone. Instead, calls are made via online platforms to those platforms' work forces.

In Chapter 9, we have identified the significant impact of a crowd worker's reputation on the worker's earnings. Since platforms store their workers' reputation as earned on the respective platform, it is unprofitable for a crowd worker to work on multiple platforms, assuming a sufficient amount of work is available on that platform. Crowd workers operating on multiple platforms split their reputation, so no platform is aware of the workers' total reputation. As a consequence, workers operating on multiple platforms are not treated as the reputable workers they are, particularly when compared to workers that have earned the same total reputation on a single platform.

Platforms focussing on creative tasks often offer requesters to approach highly reputable workers directly. For example, a designer's style may appeal to a requester seeking a new company logo. So the requester may place a direct order with the designer via the platform, instead of communicating the task via open calls. Requesters directly approaching individual crowd workers does not fit the strict definition of crowd working. However, the reputable workers still are crowd workers, who stand out from the crowd due to their quality work. So, while not strictly being crowd work, we still consider requests directed at individuals as part of crowd work if the requests are made through a platform that promotes crowd work and if the targeted individuals are crowd workers on that platform.

This group of crowd workers can greatly benefit from our personal reputation system from Chapter 10. First, our system allows the workers to earn reputation from multiple platforms. Second, our system prevents loss of reputation due to unfortunate events, for example, platform bankruptcy. Crowd workers may also benefit from our personal reputation system in a third way, namely, if we go beyond reputation. Crowd workers may have skills and experiences that qualify them for performing certain tasks. For example, language skills or experiences working with a certain software may be required for solving a certain task.

In current practice, platforms require workers to pass tests to verify workers have the necessary skills and experiences. Workers operating on multiple platforms then have to pass tests on each platform individually. This redundancy is a waste of the workers' time and the platforms' resources, particularly if each platform has their own version of a test. Thus, it would be beneficial to workers and platforms to have workers pass a test once and have the test results documented in a worker's decentralized platform-independent profile.

By admitting our personal reputation system to include test results as ratings, we make a first step at evolving our reputation system towards a decentralized platform-independent profile for crowd workers. Such a profile comes with privacy implications. For instance, we need to ask what information crowd workers are willing to share with the public and what information should be restricted to potential requesters. In the setting of crowd work, everyone is a potential requester, so simply distinguishing potential requesters from non-requesters is not an option. Instead, we need an access control mechanism that can work with fine-grained access policies, but is also compatible with the notion of a personal reputation system. ABE lends itself as an access control mechanism.

Then, we need means for entities authorized to access some information to be able to search that information. To that end, we can employ searchable encryption with access control. Admittedly, our constructions of searchable encryption with access control are possibly not the best options to be combined with our hash chain-based personal reputation system. This is due to the multi-level index structures in SEAC, dSEAC and DIA, c.f. Sections 6.2, 6.3 and 7.2, and the limited expressiveness of access policies in RLS, c.f. Section 7.1. We leave finding a searchable encryption scheme that integrates well with a personal reputation system as an open problem for future research.

As part of our discussion of practical aspects of our personal reputation system PRS, c.f. Section 10.3, we consider the possibility of specialized cloud services that operate the reputation system on crowd workers' behalves. One advantage of such services is that crowd workers do not need to concern themselves with the technicalities of operating a personal reputation system. With our proposed extension of personal reputation systems towards platform-independent crowd worker profiles, access control mechanisms increase the number of technicalities. So there is an even greater need for online services operating crowd workers' profiles on crowd workers' behalves.

Running these services securely is a challenge, particularly if we want to mitigate vendor lock-in. This means, crowd workers must, at all times, maintain full control over their profiles and be able to migrate the services for their profiles from one cloud service provider to another. Maintaining full control over their data requires crowd workers to keep cryptographic keys private and to not reveal the keys to cloud service providers. Our cloud architecture and the CCS, c.f. Chapter 13, provide means for online services to operate on crowd workers' cryptographic keys without the cloud service providers learning the keys. Then, crowd workers only need to run the organizational key store from our architecture. Due to the setting, we assume the organizational key store to be rather simple and bundled with a client-side application that, apart from storing master keys for the personal reputation system and a searchable encryption scheme with access control, also has the capacity to generate such keys. As a result, crowd workers do not need to concern themselves with technicalities. They simply install an application on their devices, choose the cloud service provider they want to use, and everything beyond is handled automatically by the application. We leave realizing this idea as future work for the members of research program "Digital Future."

Bibliography

[Abd+08] Michel Abdalla et al. “Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions.” In: *J. Cryptology* 21.3 (2008), pp. 350–391. DOI: 10.1007/s00145-007-9006-6. URL: https://doi.org/10.1007/s00145-007-9006-6.

[AMR17] James Alderman, Keith M. Martin, and Sarah Louise Renwick. “Multi-level Access in Searchable Symmetric Encryption.” In: *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*. Ed. by Michael Brenner et al. Vol. 10323. LNCS. Springer, 2017, pp. 35–52. DOI: 10.1007/978-3-319-70278-0_3. URL: https://doi.org/10.1007/978-3-319-70278-0_3.

[And+08] Elli Androulaki, Seung Geol Choi, Steven M. Bellovin, and Tal Malkin. “Reputation Systems for Anonymous Networks.” In: *Privacy Enhancing Technologies, 8th International Symposium, PETS 2008, Leuven, Belgium, July 23-25, 2008, Proceedings*. Ed. by Nikita Borisov and Ian Goldberg. Vol. 5134. LNCS. Springer, 2008, pp. 202–218. DOI: 10.1007/978-3-540-70630-4_13. URL: https://doi.org/10.1007/978-3-540-70630-4_13.

[AS] Federico Ast and Alejandro Sewrjugin. *The CrowdJury, a Crowdsourced Justice System for the Collaboration Era*. Checked 2019-08-30. URL: https://www.weusecoins.com/assets/pdf/library/The%20CrowdJury%20a%20Crowdsourced%20Justice%20System%20for%20the%20Collaboration%20Era.pdf.

[Ash+16] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. “Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations.” In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. Ed. by Daniel Wichs and Yishay Mansour. ACM, 2016, pp. 1101–1114. DOI: 10.1145/2897518.2897562. URL: https://doi.org/10.1145/2897518.2897562.

[BEJ18] Johannes Blömer, Fabian Eidens, and Jakob Juhnke. “Practical, Anonymous, and Publicly Linkable Universally-Composable Reputation Systems.” In: *Topics in Cryptology - CT-RSA 2018 - The Cryptographers’ Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*. Ed. by Nigel P. Smart. Vol. 10808. LNCS. Springer, 2018, pp. 470–490. DOI: 10.1007/978-3-319-76953-0_25. URL: https://doi.org/10.1007/978-3-319-76953-0_25.

[Ber+13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Keccak." In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings.* Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, 2013, pp. 313–314. DOI: `10.1007/978-3-642-38348-9\_19`. URL: `https://doi.org/10.1007/978-3-642-38348-9%5C_19`.

[BJK15] Johannes Blömer, Jakob Juhnke, and Christina Kolb. "Anonymous and Publicly Linkable Reputation Systems." In: *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers.* Ed. by Rainer Böhme and Tatsuaki Okamoto. Vol. 8975. LNCS. Springer, 2015, pp. 478–488. DOI: `10.1007/978-3-662-47854-7_29`. URL: `https://doi.org/10.1007/978-3-662-47854-7_29`.

[BL17] Johannes Blömer and Gennadij Liske. "Subtleties in Security Definitions for Predicate Encryption with Public Index." In: *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings.* Ed. by Johannes Blömer, Ilias S. Kotsireas, Temur Kutsia, and Dimitris E. Simos. Vol. 10693. LNCS. Springer, 2017, pp. 438–453. DOI: `10.1007/978-3-319-72453-9_35`. URL: `https://doi.org/10.1007/978-3-319-72453-9_35`.

[BL18] Johannes Blömer and Nils Löken. "Cloud Architectures for Searchable Encryption." In: *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018.* Ed. by Sebastian Doerr, Mathias Fischer, Sebastian Schrittwieser, and Dominik Herrmann. ACM, 2018, 25:1–25:10. DOI: `10.1145/3230833.3230853`. URL: `http://doi.acm.org/10.1145/3230833.3230853`.

[BL19a] Johannes Blömer and Nils Löken. "Dynamic Searchable Encryption with Access Control." In: *Foundations and Practice of Security - 12th International Symposium, FPS 2019.* Vol. 12056. LNCS. In print. Springer, 2019.

[BL19b] Johannes Blömer and Nils Löken. "Personal Cross-Platform Reputation." In: *The 15th International Workshop on Security and Trust Management (STM 2019).* Vol. 11738. LNCS. In print. Springer, 2019.

[Blö+17] Johannes Blömer, Peter Günther, Volker Krummel, and Nils Löken. "Attribute-Based Encryption as a Service for Access Control in Large-Scale Organizations." In: *Foundations and Practice of Security - 10th International Symposium, FPS 2017.* Vol. 10723. LNCS. Springer, 2017, pp. 3–17. DOI: `10.1007/978-3-319-75650-9_1`. URL: `https://doi.org/10.1007/978-3-319-75650-9_1`.

[Blö+20] Johannes Blömer, Peter Günther, Volker Krummel, and Nils Löken. "A Cloud Architecture for Cryptography in the Cloud." In preparation for submission. 2020.

[Boh+13] Jens-Matthias Bohli, Nils Gruschka, Meiko Jensen, Luigi Lo Iacono, and Ninja Marnau. "Security and Privacy-Enhancing Multicloud Architectures." In: *IEEE Trans. Dependable Sec. Comput.* 10.4 (2013), pp. 212–224. DOI: `10.1109/TDSC.2013.6`. URL: `https://doi.org/10.1109/TDSC.2013.6`.

[Bon+04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. "Public Key Encryption with Keyword Search." In: *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings.* Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, 2004, pp. 506–522. DOI: 10.1007/978-3-540-24676-3_30. URL: https://doi.org/10.1007/978-3-540-24676-3_30.

[Bös+14] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. "A Survey of Provably Secure Searchable Encryption." In: *ACM Comput. Surv.* 47.2 (2014), 18:1–18:51. DOI: 10.1145/2636328. URL: http://doi.acm.org/10.1145/2636328.

[Bos16] Raphael Bost. "$\Sigma o\phi o\varsigma$: Forward Secure Searchable Encryption." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 1143–1154. DOI: 10.1145/2976749.2978303. URL: http://doi.acm.org/10.1145/2976749.2978303.

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." In: *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.* Ed. by Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby. ACM, 1993, pp. 62–73. DOI: 10.1145/168588.168596. URL: https://doi.org/10.1145/168588.168596.

[BSW06] Dan Boneh, Emily Shen, and Brent Waters. "Strongly Unforgeable Signatures Based on Computational Diffie-Hellman." In: *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings.* Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. LNCS. Springer, 2006, pp. 229–240. DOI: 10.1007/11745853_15. URL: https://doi.org/10.1007/11745853_15.

[BSW07] John Bethencourt, Amit Sahai, and Brent Waters. "Ciphertext-Policy Attribute-Based Encryption." In: *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA.* IEEE Computer Society, 2007, pp. 321–334. DOI: 10.1109/SP.2007.11. URL: https://doi.org/10.1109/SP.2007.11.

[Buc+17] Francesco Buccafurri, Gianluca Lax, Serena Nicolazzo, and Antonino Nocera. "Tweetchain: An Alternative to Blockchain for Crowd-Based Applications." In: *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings.* Ed. by Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone. Vol. 10360. LNCS. Springer, 2017, pp. 386–393. DOI: 10.1007/978-3-319-60131-1_24. URL: https://doi.org/10.1007/978-3-319-60131-1%5C_24.

[Bug+11] Sven Bugiel, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. "Twin Clouds: Secure Cloud Computing with Low Latency - (Full Version)." In: *Communications and Multimedia Security, 12th IFIP TC 6 / TC 11 International Conference, CMS 2011.* Vol. 7025. LNCS. Springer, 2011, pp. 32–44. DOI: 10.1007/978-3-642-24712-5_3. URL: https://doi.org/10.1007/978-3-642-24712-5_3.

[Can+96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. "Adaptively Secure Multi-Party Computation." In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. Ed. by Gary L. Miller. ACM, 1996, pp. 639–648. DOI: 10.1145/237814.238015. URL: https://doi.org/10.1145/237814.238015.

[Cas+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries." In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*. Vol. 8042. LNCS. Springer, 2013, pp. 353–373. DOI: 10.1007/978-3-642-40041-4_20. URL: https://doi.org/10.1007/978-3-642-40041-4_20.

[Cas+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation." In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014*. The Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/dynamic-searchable-encryption-very-large-databases-data-structures-and-implementation.

[Cas+15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. "Leakage-Abuse Attacks Against Searchable Encryption." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 2015, pp. 668–679. DOI: 10.1145/2810103.2813700. URL: https://doi.org/10.1145/2810103.2813700.

[CD15] Payal Chaudhari and Manik Lal Das. "Privacy-preserving Attribute Based Searchable Encryption." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 899. URL: http://eprint.iacr.org/2015/899.

[CG09] Christian Cachin and Martin Geisler. "Integrity Protection for Revision Control." In: *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009*. Ed. by Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud. Vol. 5536. LNCS. 2009, pp. 382–399. DOI: 10.1007/978-3-642-01957-9_24. URL: https://doi.org/10.1007/978-3-642-01957-9_24.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. "The random oracle methodology, revisited." In: *J. ACM* 51.4 (2004), pp. 557–594. DOI: 10.1145/1008731.1008734. URL: https://doi.org/10.1145/1008731.1008734.

[Cha07] Melissa Chase. "Multi-authority Attribute Based Encryption." In: *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*. Ed. by Salil P. Vadhan. Vol. 4392. LNCS. Springer, 2007, pp. 515–534. DOI: 10.1007/978-3-540-70936-7_28. URL: https://doi.org/10.1007/978-3-540-70936-7_28.

[CK10] Melissa Chase and Seny Kamara. "Structured Encryption and Controlled Disclosure." In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*. Ed. by Masayuki Abe. Vol. 6477. LNCS. Springer, 2010, pp. 577–594. DOI: 10.1007/978-3-642-17373-8_33. URL: https://doi.org/10.1007/978-3-642-17373-8_33.

[Clo11] Cloud Security Alliance. *Security Guidance for Critical Areas of Focus in Cloud Computing V3.0.* Checked 2019-08-19. 2011. URL: `https://downloads.cloudsecurityalliance.org/initiatives/guidance/csaguide.v3.0.pdf`.

[Clo12] Cloud Security Alliance. *SecaaS Category 8 Encryption Implementation Guidance.* Checked 2019-08-19. 2012. URL: `https://downloads.cloudsecurityalliance.org/initiatives/secaas/SecaaS_Cat_8_Encryption_Implementation_Guidance.pdf`.

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. "Privacy Preserving Keyword Searches on Remote Encrypted Data." In: *Applied Cryptography and Network Security, Third International Conference, ACNS 2005.* Vol. 3531. LNCS. 2005, pp. 442–455. DOI: `10.1007/11496137_30`. URL: `https://doi.org/10.1007/11496137_30`.

[CSK13] Sebastian Clauß, Stefan Schiffner, and Florian Kerschbaum. "k-anonymous reputation." In: *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013.* Ed. by Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng. ACM, 2013, pp. 359–368. DOI: `10.1145/2484313.2484361`. URL: `https://doi.org/10.1145/2484313.2484361`.

[Cur+06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. "Searchable symmetric encryption: improved definitions and efficient constructions." In: *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006.* ACM, 2006, pp. 79–88. DOI: `10.1145/1180405.1180417`. URL: `http://doi.acm.org/10.1145/1180405.1180417`.

[Cur+11] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. "Searchable symmetric encryption: Improved definitions and efficient constructions." In: *Journal of Computer Security* 19.5 (2011), pp. 895–934. DOI: `10.3233/JCS-2011-0426`. URL: `https://doi.org/10.3233/JCS-2011-0426`.

[DGC15] Qiuxiang Dong, Zhi Guan, and Zhong Chen. "Attribute-Based Keyword Search Efficiency Enhancement via an Online/Offline Approach." In: *21st IEEE International Conference on Parallel and Distributed Systems, ICPADS 2015, Melbourne, Australia, December 14-17, 2015.* IEEE Computer Society, 2015, pp. 298–305. DOI: `10.1109/ICPADS.2015.45`. URL: `https://doi.org/10.1109/ICPADS.2015.45`.

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography." In: *IEEE Trans. Information Theory* 22.6 (1976), pp. 644–654. DOI: `10.1109/TIT.1976.1055638`. URL: `https://doi.org/10.1109/TIT.1976.1055638`.

[DO15] Richard Dennis and Gareth Owen. "Rep on the block: A next generation reputation system based on the blockchain." In: *10th International Conference for Internet Technology and Secured Transactions, ICITST 2015, London, United Kingdom, December 14-16, 2015.* IEEE, 2015, pp. 131–138. DOI: `10.1109/ICITST.2015.7412073`. URL: `https://doi.org/10.1109/ICITST.2015.7412073`.

[DR98] Joan Daemen and Vincent Rijmen. “The Block Cipher Rijndael.” In: *Smart Card Research and Applications, This International Conference, CARDIS ’98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Vol. 1820. LNCS. Springer, 1998, pp. 277–284. DOI: `10.1007/10721064\_26`. URL: `https://doi.org/10.1007/10721064%5C_26`.

[DRD08] Changyu Dong, Giovanni Russello, and Naranker Dulay. “Shared and Searchable Encrypted Data for Untrusted Servers.” In: *Data and Applications Security XXII, 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security, DBSec 2008*. Vol. 5094. LNCS. Springer, 2008, pp. 127–143. DOI: `10.1007/978-3-540-70567-3_10`. URL: `https://doi.org/10.1007/978-3-540-70567-3_10`.

[EG12] Enrique Estellés-Arolas and Fernando González-Ladrón-De-Guevara. “Towards an integrated crowdsourcing definition.” In: *Journal of Information Science* 38.2 (2012), pp. 189–200.

[ElG85] Taher ElGamal. “A public key cryptosystem and a signature based on the discrete logarithm.” In: *IEEE Trans. Inf. Theory* 31.4 (1985), pp. 469–472.

[Ete+18] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. “Efficient Dynamic Searchable Encryption with Forward Privacy.” In: *PoPETs* 2018.1 (2018), pp. 5–20. DOI: `10.1515/popets-2018-0002`. URL: `https://doi.org/10.1515/popets-2018-0002`.

[Fer+18] Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão. “BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage.” In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 588. URL: `https://eprint.iacr.org/2018/588`.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. “Secure Integration of Asymmetric and Symmetric Encryption Schemes.” In: *J. Cryptology* 26.1 (2013), pp. 80–101. DOI: `10.1007/s00145-011-9114-1`. URL: `https://doi.org/10.1007/s00145-011-9114-1`.

[Gen09] Craig Gentry. “Fully homomorphic encryption using ideal lattices.” In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*. ACM, 2009, pp. 169–178. DOI: `10.1145/1536414.1536440`. URL: `http://doi.acm.org/10.1145/1536414.1536440`.

[GHW11] Matthew Green, Susan Hohenberger, and Brent Waters. “Outsourcing the Decryption of ABE Ciphertexts.” In: *20th USENIX Security Symposium*. USENIX Association, 2011. URL: `http://static.usenix.org/events/sec11/tech/full_papers/Green.pdf`.

[Gir+17] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. “Practical Passive Leakage-abuse Attacks Against Symmetric Searchable Encryption.” In: *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017*. Ed. by Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello. SciTePress, 2017, pp. 200–211. DOI: `10.5220/0006461202000211`. URL: `https://doi.org/10.5220/0006461202000211`.

[GMP16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. "TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption." In: *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference*. Vol. 9816. LNCS. Springer, 2016, pp. 563–592. DOI: 10.1007/978-3-662-53015-3_20. URL: https://doi.org/10.1007/978-3-662-53015-3_20.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. Ed. by Alfred V. Aho. ACM, 1987, pp. 218–229. DOI: 10.1145/28395.28420. URL: https://doi.org/10.1145/28395.28420.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *J. ACM* 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553. URL: https://doi.org/10.1145/233551.233553.

[Goh03] Eu-Jin Goh. "Secure Indexes." In: *IACR Cryptology ePrint Archive* 2003 (2003), p. 216. URL: http://eprint.iacr.org/2003/216.

[Goy+06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. "Attribute-based encryption for fine-grained access control of encrypted data." In: *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*. ACM, 2006, pp. 89–98. DOI: 10.1145/1180405.1180418. URL: http://doi.acm.org/10.1145/1180405.1180418.

[Gri+09] Tal Grinshpoun, Nurit Gal-Oz, Amnon Meisels, and Ehud Gudes. "CCR: A Model for Sharing Reputation Knowledge Across Virtual Communities." In: *2009 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2009, Milan, Italy, 15-18 September 2009, Main Conference Proceedings*. IEEE Computer Society, 2009, pp. 34–41. DOI: 10.1109/WI-IAT.2009.13. URL: https://doi.org/10.1109/WI-IAT.2009.13.

[Gru+18] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 315–331. DOI: 10.1145/3243734.3243864. URL: https://doi.org/10.1145/3243734.3243864.

[GS12] Jens Groth and Amit Sahai. "Efficient Noninteractive Proof Systems for Bilinear Groups." In: *SIAM J. Comput.* 41.5 (2012), pp. 1193–1232. DOI: 10.1137/080725386. URL: https://doi.org/10.1137/080725386.

[HBC15] Ferry Hendrikx, Kris Bubendorfer, and Ryan Chard. "Reputation systems: A survey and taxonomy." In: *J. Parallel Distrib. Comput.* 75 (2015), pp. 184–197. DOI: 10.1016/j.jpdc.2014.08.004. URL: https://doi.org/10.1016/j.jpdc.2014.08.004.

[HOR15] Brett Hemenway, Rafail Ostrovsky, and Alon Rosen. "Non-committing Encryption from Φ-hiding." In: *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9014. LNCS. Springer, 2015, pp. 591–608. DOI: 10.1007/978-3-662-46494-6_24. URL: https://doi.org/10.1007/978-3-662-46494-6%5C_24.

[Hor17] Helena Horton. "Garden Shed Becomes Top-Rated London Restaurant on TripAdvisor after Site Tricked by Fake Reviews." In: *The Telegraph* 2017-12-06 (2017). Checked 2019-07-17. URL: https://www.telegraph.co.uk/news/2017/12/06/garden-becomes-top-rated-london-restaurant-tripadvisor-site/.

[IJ02] Roslan Ismail and Audun Jøsang. "The Beta Reputation System." In: *15th Bled eConference: eReality: Constructing the eEconomy, Bled, Slovenia, June 17-19, 2002*. 2002, p. 41. URL: http://aisel.aisnet.org/bled2002/41.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation." In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation.

[JG09] Audun Jøsang and Jennifer Golbeck. "Challenges for robust trust and reputation systems." In: *5th International Workshop on Security and Trust Management (STM 2009), Saint Malo, France, September 2009*. Preprint. 2009.

[KB14] Abdellah Kaci and Thouraya Bouabana-Tebibel. "Access control reinforcement over searchable encryption." In: *Proceedings of the 15th IEEE International Conference on Information Reuse and Integration, IRI 2014, Redwood City, CA, USA, August 13- 15, 2014*. Ed. by James Joshi, Elisa Bertino, Bhavani M. Thuraisingham, and Ling Liu. IEEE Computer Society, 2014, pp. 130–137. DOI: 10.1109/IRI.2014.7051882. URL: https://doi.org/10.1109/IRI.2014.7051882.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269.

[KM19] Seny Kamara and Tarik Moataz. "Computationally Volume-Hiding Structured Encryption." In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. LNCS. Springer, 2019, pp. 183–213. DOI: 10.1007/978-3-030-17656-3_7. URL: https://doi.org/10.1007/978-3-030-17656-3_7.

[KMO18] Seny Kamara, Tarik Moataz, and Olga Ohrimenko. "Structured Encryption and Leakage Suppression." In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, 2018, pp. 339–370. DOI: 10.1007/978-3-319-96884-1_12. URL: https://doi.org/10.1007/978-3-319-96884-1_12.

[KO12] Kaoru Kurosawa and Yasuhiro Ohtaki. "UC-Secure Searchable Symmetric Encryption." In: *Financial Cryptography and Data Security - 16th International Conference, FC 2012*. Vol. 7397. LNCS. Springer, 2012, pp. 285–298. DOI: 10.1007/978-3-642-32946-3_21. URL: https://doi.org/10.1007/978-3-642-32946-3_21.

[KO13] Kaoru Kurosawa and Yasuhiro Ohtaki. "How to Update Documents Verifiably in Searchable Symmetric Encryption." In: *Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, November 20-22. 2013. Proceedings*. Ed. by Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab. Vol. 8257. LNCS. Springer, 2013, pp. 309–328. DOI: 10.1007/978-3-319-02937-5_17. URL: https://doi.org/10.1007/978-3-319-02937-5_17.

[KP13] Seny Kamara and Charalampos Papamanthou. "Parallel and Dynamic Searchable Symmetric Encryption." In: *Financial Cryptography and Data Security - 17th International Conference, FC 2013*. Ed. by Ahmad-Reza Sadeghi. Vol. 7859. LNCS. Springer, 2013, pp. 258–274. DOI: 10.1007/978-3-642-39884-1_22. URL: https://doi.org/10.1007/978-3-642-39884-1_22.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. "Dynamic searchable symmetric encryption." In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 965–976. DOI: 10.1145/2382196.2382298. URL: http://doi.acm.org/10.1145/2382196.2382298.

[KSG03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. "The Eigentrust algorithm for reputation management in P2P networks." In: *Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003*. Ed. by Gusztáv Hencsey, Bebo White, Yih-Farn Robin Chen, László Kovács, and Steve Lawrence. ACM, 2003, pp. 640–651. DOI: 10.1145/775152.775242. URL: https://doi.org/10.1145/775152.775242.

[KV08] Eike Kiltz and Yevgeniy Vahlis. "CCA2 Secure IBE: Standard Model Efficiency through Authenticated Symmetric Encryption." In: *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*. Ed. by Tal Malkin. Vol. 4964. LNCS. Springer, 2008, pp. 221–238. DOI: 10.1007/978-3-540-79263-5_14. URL: https://doi.org/10.1007/978-3-540-79263-5%5C_14.

[Li+19] Ming Li, Jian Weng, Anjia Yang, Wei Lu, Yue Zhang, Lin Hou, Jia-Nan Liu, Yang Xiang, and Robert H. Deng. "CrowdBC: A Blockchain-Based Decentralized Framework for Crowdsourcing." In: *IEEE Trans. Parallel Distrib. Syst.* 30.6 (2019), pp. 1251–1266. DOI: 10.1109/TPDS.2018.2881735. URL: https://doi.org/10.1109/TPDS.2018.2881735.

[Lie+10] Peter van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. "Computationally Efficient Searchable Symmetric Encryption." In: *Secure Data Management, 7th VLDB Workshop, SDM 2010*. Vol. 6358. LNCS. Springer, 2010, pp. 87–100. DOI: 10.1007/978-3-642-15546-8_7. URL: https://doi.org/10.1007/978-3-642-15546-8_7.

[Lök17] Nils Löken. "Searchable Encryption with Access Control." In: *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*. ACM, 2017, 24:1–24:6. DOI: 10.1145/3098954.3098987. URL: http://doi.acm.org/10.1145/3098954.3098987.

[LS11] Jake Loftus and Nigel P. Smart. “Secure Outsourced Computation.” In: *Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings.* Ed. by Abderrahmane Nitaj and David Pointcheval. Vol. 6737. LNCS. Springer, 2011, pp. 1–20. DOI: `10.1007/978-3-642-21969-6\_1`. URL: `https://doi.org/10.1007/978-3-642-21969-6%5C_1`.

[LW11] Allison B. Lewko and Brent Waters. “Decentralizing Attribute-Based Encryption.” In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings.* Ed. by Kenneth G. Paterson. Vol. 6632. LNCS. Springer, 2011, pp. 568–588. DOI: `10.1007/978-3-642-20465-4_31`. URL: `https://doi.org/10.1007/978-3-642-20465-4_31`.

[LZ14] Jia-Zhi Li and Lei Zhang. “Attribute-Based Keyword Search and Data Access Control in Cloud.” In: *Tenth International Conference on Computational Intelligence and Security, CIS 2014.* IEEE Computer Society, 2014, pp. 382–386. DOI: `10.1109/CIS.2014.113`. URL: `https://doi.org/10.1109/CIS.2014.113`.

[LZC15] Chang Liu, Liehuang Zhu, and Jinjun Chen. “Efficient Searchable Symmetric Encryption for Storing Multiple Source Data on Cloud.” In: *2015 IEEE TrustCom/BigDataSE/ISPA, Volume 1.* IEEE Computer Society, 2015, pp. 451–458. DOI: `10.1109/Trustcom.2015.406`. URL: `https://doi.org/10.1109/Trustcom.2015.406`.

[Mic19] Antonis Michalas. “The lord of the shares: combining attribute-based encryption and searchable encryption for flexible data sharing.” In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019.* Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, 2019, pp. 146–155. DOI: `10.1145/3297280.3297297`. URL: `https://doi.org/10.1145/3297280.3297297`.

[MT19] Evangelia Anna Markatou and Roberto Tamassia. “Full Database Reconstruction with Access and Search Pattern Leakage.” In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 395. URL: `https://eprint.iacr.org/2019/395`.

[Nav15] Muhammad Naveed. “The Fallacy of Composition of Oblivious RAM and Searchable Encryption.” In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 668. URL: `http://eprint.iacr.org/2015/668`.

[Neu+05] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn. *The Kerberos Network Authentication Service (V5).* RFC 4120. 2005.

[ÖAS15] Cengiz Örencik, Mahmoud Alewiwi, and Erkay Savas. “Secure Sketch Search for Document Similarity.” In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1.* IEEE, 2015, pp. 1102–1107. DOI: `10.1109/Trustcom.2015.489`. URL: `https://doi.org/10.1109/Trustcom.2015.489`.

[ÖKS13] Cengiz Örencik, Murat Kantarcioglu, and Erkay Savas. “A Practical and Secure Multi-keyword Search Method over Encrypted Cloud Data.” In: *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013.* IEEE Computer Society, 2013, pp. 390–397. DOI: `10.1109/CLOUD.2013.18`. URL: `https://doi.org/10.1109/CLOUD.2013.18`.

[Öre+16] Cengiz Örencik, Ayse Selcuk, Erkay Savas, and Murat Kantarcioglu. "Multi-Keyword search over encrypted data with scoring and search pattern obfuscation." In: *Int. J. Inf. Sec.* 15.3 (2016), pp. 251–269. DOI: `10.1007/s10207-015-0294-9`. URL: `https://doi.org/10.1007/s10207-015-0294-9`.

[Poh+17] Geong Sen Poh, Ji-Jian Chin, Wei-Chuen Yau, Kim-Kwang Raymond Choo, and Moesfa Soeheila Mohamad. "Searchable Symmetric Encryption: Designs and Challenges." In: *ACM Comput. Surv.* 50.3 (2017), 40:1–40:37. DOI: `10.1145/3064005`. URL: `http://doi.acm.org/10.1145/3064005`.

[PS08] Franziska Pingel and Sandra Steinbrecher. "Multilateral Secure Cross-Community Reputation Systems for Internet Communities." In: *Trust, Privacy and Security in Digital Business, 5th International Conference, TrustBus 2008, Turin, Italy, September 4-5, 2008, Proceedings.* Ed. by Steven Furnell, Sokratis K. Katsikas, and Antonio Lioy. Vol. 5185. LNCS. Springer, 2008, pp. 69–78. DOI: `10.1007/978-3-540-85735-8_8`. URL: `https://doi.org/10.1007/978-3-540-85735-8_8`.

[PS16] David Pointcheval and Olivier Sanders. "Short Randomizable Signatures." In: *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings.* Ed. by Kazue Sako. Vol. 9610. LNCS. Springer, 2016, pp. 111–126. DOI: `10.1007/978-3-319-29485-8\_7`. URL: `https://doi.org/10.1007/978-3-319-29485-8%5C_7`.

[PZ13] Raluca A. Popa and Nickolai Zeldovich. "Multi-Key Searchable Encryption." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 508. URL: `http://eprint.iacr.org/2013/508`.

[Rab81] Michael O. Rabin. *How to exchange secrets with oblivious transfer.* Tech. rep. Technical ReportTR-81. Aiken Computation Lab, Harvard University, 1981. URL: `https://eprint.iacr.org/2005/187`.

[RMÖ15] Cédric Van Rompay, Refik Molva, and Melek Önen. "Multi-user Searchable Encryption in the Cloud." In: *Information Security - 18th International Conference, ISC 2015, Trondheim, Norway, September 9-11, 2015, Proceedings.* Ed. by Javier López and Chris J. Mitchell. Vol. 9290. LNCS. Springer, 2015, pp. 299–316. DOI: `10.1007/978-3-319-23318-5_17`. URL: `https://doi.org/10.1007/978-3-319-23318-5_17`.

[RMÖ17] Cédric Van Rompay, Refik Molva, and Melek Önen. "A Leakage-Abuse Attack Against Multi-User Searchable Encryption." In: *PoPETs* 2017.3 (2017), p. 168. DOI: `10.1515/popets-2017-0034`. URL: `https://doi.org/10.1515/popets-2017-0034`.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (1978), pp. 120–126. DOI: `10.1145/359340.359342`. URL: `http://doi.acm.org/10.1145/359340.359342`.

[RW13] Yannis Rouselakis and Brent Waters. "Practical constructions and new proof methods for large universe attribute-based encryption." In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13.* ACM, 2013, pp. 463–474. DOI: `10.1145/2508859.2516672`. URL: `http://doi.acm.org/10.1145/2508859.2516672`.

[RW15] Yannis Rouselakis and Brent Waters. “Efficient Statically-Secure Large-Universe Multi-Authority Attribute-Based Encryption.” In: *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*. Ed. by Rainer Böhme and Tatsuaki Okamoto. Vol. 8975. LNCS. Springer, 2015, pp. 315–332. DOI: 10.1007/978-3-662-47854-7_19. URL: https://doi.org/10.1007/978-3-662-47854-7_19.

[SL08] Aameek Singh and Ling Liu. “Sharoes: A Data Sharing Platform for Outsourced Enterprise Storage Environments.” In: *ICDE 2008*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE Computer Society, 2008, pp. 993–1002. DOI: 10.1109/ICDE.2008.4497508. URL: https://doi.org/10.1109/ICDE.2008.4497508.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. “Practical Dynamic Searchable Encryption with Small Leakage.” In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL: http://www.internetsociety.org/doc/practical-dynamic-searchable-encryption-small-leakage.

[SSW10] Ahmad-Reza Sadeghi, Thomas Schneider, and Marcel Winandy. “Token-Based Cloud Computing.” In: *Trust and Trustworthy Computing, Third International Conference, TRUST 2010*. Vol. 6101. LNCS. Springer, 2010, pp. 417–429. DOI: 10.1007/978-3-642-13869-0_30. URL: https://doi.org/10.1007/978-3-642-13869-0_30.

[Sun+14] Wenhai Sun, Shucheng Yu, Wenjing Lou, Y. Thomas Hou, and Hui Li. “Protecting your right: Attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud.” In: *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. IEEE Computer Society, 2014, pp. 226–234. DOI: 10.1109/INFOCOM.2014.6847943. URL: https://doi.org/10.1109/INFOCOM.2014.6847943.

[Sun+16] Wenhai Sun, Shucheng Yu, Wenjing Lou, Y. Thomas Hou, and Hui Li. “Protecting Your Right: Verifiable Attribute-Based Keyword Search with Fine-Grained Owner-Enforced Search Authorization in the Cloud.” In: *IEEE Trans. Parallel Distrib. Syst.* 27.4 (2016), pp. 1187–1198. DOI: 10.1109/TPDS.2014.2355202. URL: https://doi.org/10.1109/TPDS.2014.2355202.

[SW05] Amit Sahai and Brent Waters. “Fuzzy Identity-Based Encryption.” In: *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*. Ed. by Ronald Cramer. Vol. 3494. LNCS. Springer, 2005, pp. 457–473. DOI: 10.1007/11426639_27. URL: https://doi.org/10.1007/11426639_27.

[SWP00] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. “Practical Techniques for Searches on Encrypted Data.” In: *2000 IEEE Symposium on Security and Privacy, S&P 2000*. IEEE Computer Society, 2000, pp. 44–55. DOI: 10.1109/SECPRI.2000.848445. URL: https://doi.org/10.1109/SECPRI.2000.848445.

[Tom+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. "Transparency Logs via Append-Only Authenticated Dictionaries." In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* Pre-proceeding. ACM, 2019. URL: `https://eprint.iacr.org/2018/721`.

[TP10] Juan Ramón Troncoso-Pastoriza and Fernando Pérez-González. "CryptoDSPs for Cloud Privacy." In: *Web Information Systems Engineering - WISE 2010 Workshops - WISE 2010 International Symposium WISS, and International Workshops CISE, MBC.* Vol. 6724. LNCS. Springer, 2010, pp. 428–439. DOI: `10.1007/978-3-642-24396-7_34`. URL: `https://doi.org/10.1007/978-3-642-24396-7_34`.

[TSya] T-Systems International GmbH. *Microsoft Azure – Skalierbare Cloud Infrastruktur & Plattformen – TelekomCLOUD.* Checked 2019-08-20. German only. URL: `https://cloud.telekom.de/de/infrastruktur/microsoft-azure-international`.

[TSyb] T-Systems International GmbH. *Open Telekom Cloud – Scalable Infrastructure.* Checked 2019-08-20. URL: `https://open-telekom-cloud.com/en`.

[Wat11] Brent Waters. "Ciphertext-Policy Attribute-Based Encryption: An Expressive, Efficient, and Provably Secure Realization." In: *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings.* Ed. by Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi. Vol. 6571. LNCS. Springer, 2011, pp. 53–70. DOI: `10.1007/978-3-642-19379-8_4`. URL: `https://doi.org/10.1007/978-3-642-19379-8_4`.

[Xia+16] Zhihua Xia, Xinhui Wang, Xingming Sun, and Qian Wang. "A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data." In: *IEEE Trans. Parallel Distrib. Syst.* 27.2 (2016), pp. 340–352. DOI: `10.1109/TPDS.2015.2401003`. URL: `https://doi.org/10.1109/TPDS.2015.2401003`.

[Yan+13] Kan Yang, Xiaohua Jia, Kui Ren, Bo Zhang, and Ruitao Xie. "DAC-MACS: Effective Data Access Control for Multiauthority Cloud Storage Systems." In: *IEEE Trans. Information Forensics and Security* 8.11 (2013), pp. 1790–1801. DOI: `10.1109/TIFS.2013.2279531`. URL: `https://doi.org/10.1109/TIFS.2013.2279531`.

[Yan+15] Yanjiang Yang, Joseph K. Liu, Kaitai Liang, Kim-Kwang Raymond Choo, and Jianying Zhou. "Extended Proxy-Assisted Approach: Achieving Revocable Fine-Grained Encryption of Cloud Data." In: *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II.* Ed. by Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl. Vol. 9327. LNCS. Springer, 2015, pp. 146–166. DOI: `10.1007/978-3-319-24177-7_8`. URL: `https://doi.org/10.1007/978-3-319-24177-7_8`.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)." In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986.* IEEE Computer Society, 1986, pp. 162–167. DOI: `10.1109/SFCS.1986.25`. URL: `https://doi.org/10.1109/SFCS.1986.25`.

[YJ12] Kan Yang and Xiaohua Jia. “Attributed-Based Access Control for Multi-authority Systems in Cloud Storage.” In: *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012.* IEEE Computer Society, 2012, pp. 536–545. DOI: 10.1109/ICDCS.2012.42. URL: https://doi.org/10.1109/ICDCS.2012.42.

[YJR13] Kan Yang, Xiaohua Jia, and Kui Ren. “Attribute-based fine-grained access control with efficient revocation in cloud storage systems.” In: *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13.* ACM, 2013, pp. 523–528. DOI: 10.1145/2484313.2484383. URL: http://doi.acm.org/10.1145/2484313.2484383.

[Yu+10] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. “Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing.” In: *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA.* IEEE Computer Society, 2010, pp. 534–542. DOI: 10.1109/INFCOM.2010.5462174. URL: https://doi.org/10.1109/INFCOM.2010.5462174.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. “All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption.” In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 707–720. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang.

[ZXA14] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. “VABKS: Verifiable attribute-based keyword search over outsourced encrypted data.” In: *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014.* IEEE Computer Society, 2014, pp. 522–530. DOI: 10.1109/INFOCOM.2014.6847976. URL: https://doi.org/10.1109/INFOCOM.2014.6847976.