Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Data Structures for Puncturable Encryption

Bachelor Thesis

Mirco Stäuble

August 25, 2021

Advisors: Prof. Dr. Kenny Paterson, Matilda Backendal

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Puncturable encryption (PE) provides security guarantees for selected encrypted messages after secret key corruption, i.e. it provides forward security. In this thesis, we analyse current data-structure-based PE schemes. We extract relevant features and create a blueprint for new data structures usable for PE. To possibly allow storage reduction we introduce a new class of PE schemes which we call *Dynamic Puncturable Encryption* (DPE) schemes. We present new construction ideas and provide a more in-depth analysis for new PE schemes based on DPE and ratcheting. During the analysis of current approaches, we found requirements such as a lower bound on the number of needed keys to be able to achieve perfect correctness. Additionally, we proved that a naïve approach to PE achieves optimal storage bounds within $\mathcal{O}(n)$, whereby n denotes the number of supported messages (tags), for its class of PE schemes, and found that any scheme trying to achieve better storage requirements compared to it needs to sacrifice some algorithm efficiency or correctness.

Contents

Contents iii

1 Introduction 1

2 Preliminaries 5
2.1 Abbreviations . . . 5
2.2 Notation and Conventions . . . 6
2.2.1 Assumptions . . . 6
2.3 Symmetric Encryption . . . 7
2.3.1 Syntax . . . 7
2.3.2 Correctness . . . 8
2.4 Public-key Encryption . . . 8
2.4.1 Syntax . . . 8
2.4.2 Correctness . . . 8
2.5 Security . . . 8
2.5.1 Symmetric IND-CPA . . . 9
2.5.2 Symmetric IND-CCA . . . 10
2.5.3 Asymmetric IND-CPA . . . 11
2.5.4 Asymmetric IND-CCA . . . 12

3 Puncturable Encryption 13
3.1 Symmetric Puncturable Encryption . . . 13
3.2 Public-key Puncturable Encryption . . . 14
3.3 Security . . . 16
3.3.1 Symmetric IND-CPA including FS . . . 16
3.3.2 Symmetric IND-CCA including FS . . . 18
3.3.3 Asymmetric IND-CPA including FS . . . 19
3.3.4 Asymmetric IND-CCA including FS . . . 19
3.4 Relevant features and special orderings . . . 20
3.4.1 Features . . . 21

3.4.2 In order puncturing 22
3.4.3 Puncturing before next encryption 22
3.4.4 Quasi-Perfect Ordering 22

4 The Naïve Solution 25
4.1 Main construction idea 25
4.2 Theoretical construction 26
4.2.1 Symmetric encryption 27
4.2.2 Public-key encryption 28
4.3 Theoretical analysis 30
4.3.1 Feature summary 31

5 Background and Related Work 33
5.1 PE based on Bloom Filter Encryption 33
5.1.1 Construction 34
5.1.2 Theoretical analysis 37
5.2 Perfect Binary Tree PE 42
5.2.1 Construction 42
5.2.2 Theoretical analysis 45

6 Blueprint 49
6.1 Perfect correctness requirement 50
6.2 Storage reduction requirement 51
6.3 Consequence for new constructions 54

7 Dynamic Puncturable Encryption 55
7.1 Syntax, correctness and security 55
7.2 Problems of DPE 59
7.2.1 Unsuitability for PKPE 59

8 New Construction Ideas 61
8.1 Puncturable Encryption based on Ratcheting 61
8.2 Chained Perfect Binary Tree Puncturable Encryption 62
8.3 Dynamic Perfect Binary Tree Puncturable Encryption 63
8.4 BFE-based Puncturbale Encryption with Perfect Correctness . 65
8.5 Chained BFE-based Puncturable Encryption 66
8.6 Bloom Filter Cascade Puncturable Encryption 66
8.7 Puncturable Encryption through Re-Encryption 66
8.8 Lambda-Structures 68
8.9 BFE-based Puncturable Encryption with unbounded tag support 69
8.10 Dynamic Naïve Solution 69
8.11 Exploration Summary 69
8.12 Discarded data structures 70

8.12.1 Cuckoo Filters 70
8.12.2 Morton Filters 70
8.12.3 Counting Filters 70
8.12.4 Self-balancing trees 71
8.12.5 Heaps 71

9 Theoretical Constructions **73**
9.1 Dynamic Naïve SPE 73
9.1.1 Theoretical analysis 76
9.2 Ratcheting 79
9.2.1 Theoretical analysis 81

10 Discussion and Conclusion **85**
10.1 Discussion 86
10.2 Important Features, DSPE Conclusion and Future Work . . . 89

A Appendix **91**
A.1 BFE vs. naïve storage comparison 91
A.2 In order puncturing in PBT based PE 91

Bibliography **95**

Chapter 1

Introduction

The concept of PUNCTURABLE ENCRYPTION (PE) was introduced by Green and Miers in 2015 [10]. The basic idea of PE is to achieve encryption like in any other common encryption scheme, but additionally it includes a *puncturing* algorithm which makes it possible to revoke decryption capabilities of the key for selected messages the secret-key was punctured on. In the last few years, a lot of research was done in the field related to puncturable encryption, regarding possible applications of the concept as well as different construction ideas. One of the main applications is to provide forward security. The basic idea of forward secure encryption is to provide the guarantee that a secret-key, which gets compromised by an adversary in the future, is not able to decrypt messages encrypted in the past. The most trivial idea to achieve forward security is to generate a new secret-key after a certain predefined time elapsed. We call the time-space in which the secret-key stays the same a *window*. Using this idea, a key that gets leaked now is only able to decrypt all messages that were sent in the current window but can not decrypt any messages sent in any timeslot before, therefore achieving forward security. Using PE to achieve forward secrecy enables fine-grained control over the decryption capabilities of the secret-key, up to the level of revoking decryption capabilities on a per-message level.

One way of achieving fine-grained forward security using PE is to puncture on ciphertexts. By puncturing on a ciphertext C, corresponding to the encryption of a message M, one revokes the capability of the key to decrypt C. This can be seen as puncturing the secret-key on the message M to prevent its decryption. We will use the terms ´puncturing on a ciphertext´ and ´puncturing on a message´ interchangeably, whereby we mean the puncturing on a ciphertext notion for both. Another way to achieve fine-grained forward security using PE is to puncture on tags associated to a message instead of puncturing on individual ciphertexts. The concept then consists of the idea that after puncturing a secret-key on a specific tag, no message as-

sociated with this tag can be decrypted with the punctured key. This allows to remove the decryption capability for possibly multiple messages with one puncture call. It is important to note that by using a per-message tag, this concept is equivalent to puncturing on the messages themselves.

Recent research has shown a lot of different applications for Puncturable Encryption both in a synchronous and asynchronous setting. From the idea of providing general forward secrecy, PE can be used to achieve forward security in asynchronous messaging systems. This was the main goal of the research by Green and Miers [10]. They also suggested that the concept can be used to securely delete files in cloud based storage, or by using the idea of puncturing on tags instead of individual messages, one could revoke decryption capabilities for all messages corresponding to a tag which could correspond to a specific user or a certain topic.

Other ideas for applications of PE include the construction of forward-secure 0-RTT key-exchange protocols [11, 12, 6, 1], protect against replay attacks (as mentioned in [1]), construct backward-secure searchable encryption [14], achieve public-key watermarking schemes [4], ´forgetting´ data in distributed systems [8], forward-secret proxy re-encryption [7], and making IoT more secure by providing efficient forward secrecy for IoT devices [17].

A shortcoming of the PE schemes proposed thus far is that the secret-key either grows with each puncture, or starts out very large. For constructions based on perfect binary trees and Bloom filters it has been pointed out that encryption/decryption can be rather inefficient. This thesis aims to improve on the following aspects, required secret-key storage and algorithm efficiency for encryption /decryption / puncturing, by exploring if data structures could be used to store secret-key elements in ways that are both storage efficient and allow for fast retrieval and deletion. One goal of this thesis is to extract relevant features from the hitherto used data structures to construct a blueprint which guides the search for new schemes. We try to come up with new construction ideas which could beat a naïve solution in at least one aspect, i.e. storage requirements, speed, etc.

During the analysis of the data structures, we found requirements such as needing at least one unique key for every supported tag to be able to achieve perfect correctness or that it is only possible to achieve storage reduction through a hierarchical derivation of keys if we do not want to sacrifice perfect correctness. Additionally, we introduce a new family of puncturable encryption schemes, which we call Dynamic Puncturable Encryption schemes. Such schemes are able to extend the supported tag-space on demand and could therefore support a potentially unbounded set of tags. Finally, we present two new schemes, one based on ratcheting and a naïve dynamic PE construction. Both beat a naïve solution regarding the needed storage whilst still providing fast algorithms in most scenarios.

Naturally, there are also ways of constructing PE which do not rely on data structures. Examples include the use of puncturable pseudo-random functions (PPRF) [1, 4], using distributed key-distribution and key encapsulation techniques [13], using an adaptation of Fully-Key Homomorphic Encryption (FKHE), called Delegetable Fully-Key Homomorphic Encryption (DFKHE) [15], and general constructions out of any public-key encryption scheme [14]. In the following we focus on the constructions that make use of explicit data structures, since our goal is to explore data structures to store secret-key elements to achieve compact storage and efficient algorithms.

Chapter 2

Preliminaries

Before we can start analysing current PE constructions, we need to define some common notation and assumptions which will be used in the rest of the thesis. The provided definitions for symmetric encryption and public-key encryption, and their correctness and security, act as foundations for the upcoming definitions of PE.

2.1 Abbreviations

PE	Puncturable Encryption
SPE	Symmetric Puncturable Encryption
PkPE	Public-key Puncturable Encryption
FuPE	Fully Puncturable Encryption
DPE	Dynamic Puncturable Encryption
DSPE	Dynamic Symmetric Puncturable Encryption
FS	Forward Security/Forward-Secure
MSK	Master Secret Key
HBE	Hierarchy Based Encryption
BF	Bloom Filter
BFE	Bloom Filter Encryption
PBT	Perfect Binary Tree
0-RTT	Zero Round Trip Time
IoT	Internet of Things
ind-cpa	Indistinguishability under Chosen-Plaintext Attack

ind-cca	Indistinguishability under Chosen-Ciphertext Attack
CPBTPE	Chained Perfect Binary Tree Puncturable Encryption
DPBTPE	Dynamic Perfect Binary Tree Puncturable Encryption
BFEPEPC	Bloom Filter Encryption based Puncturable Encryption with Perfect Correctness
CBFEPE	Chained Bloom Filter Encryption based Puncturable Encryption
PPRF	Puncturable Pseudo-Random Function
FKHE	Fully-Key Homomorphic Encryption
DFKHE	Delegetable Fully-Key Homomorphic Encryption
SE	Symmetric Encryption
PkE	Public-key Encryption

2.2 Notation and Conventions

We shortly provide the relevant syntax we use in our definitions. Given a deterministic algorithm A, we denote by $b \leftarrow A(x)$ assigning the output of A, on input x, to the variable b. Additionally, we use $x \leftarrow y$ to denote copying the value of a variable or set y to x. In contrast, if A is a probabilistic algorithm, $b \leftarrow_{\$} A(x)$ denotes the assignment of the output of A, on input x, to the variable b. Similarly, if B is a set, we denote the picking of a random sample out of the set B by $b \leftarrow_{\$} B$. Using $|x|$ we denote the length of x. Is x an array, then $|x|$ corresponds to the number of entries. If x is a message, then $|x|$ is measured in bits and if x is a number, then $|x|$ denotes the absolute value of x.

By $negl(\lambda)$ we describe a negligible function in the security parameter λ. We use the following definition for negligible functions:

Definition 2.1 *We call a function $\mu : \mathbb{N} \to \mathbb{R}$ negligible, if for every $c \in \mathbb{N}$ there exists a natural number n_0, such that for all $x > n_0$ we have that $|\mu(x)| < \frac{1}{x^c}$.*

In this thesis, we will often use the term *data structure for PE*. By this, we mean specifically ´a data structure holding the various secret-key (public-key) components which make up the secret-key (public-key) of a PE scheme´.

2.2.1 Assumptions

For the theoretical construction of our schemes, we assume all the algorithms to receive only tags in the supported tag-space. If this is not the case, the encryption and decryption algorithms will fail and return $\perp$, while the puncturing algorithm just returns the unchanged secret key. We assume the same to happen for any other kind of unexpected input.

For all constructions we assume to use sequence numbers as tags, i.e. a tag-space supporting n tags consist of the sequence numbers $\{0, 1, 2, \ldots, n-1\}$.

In this thesis we consider different classes of PE schemes:

- Non-hierarchical, perfectly correct PE schemes:

  PE schemes based on non-hierarchical data structures achieving perfect correctness.

- Non-hierarchical, non-perfectly correct PE schemes:

  PE schemes based on non-hierarchical data structures not achieving perfect correctness.

- Hierarchical, perfectly correct PE schemes:

  PE schemes based on hierarchical data structures achieving perfect correctness.

- Hierarchical, non-perfectly correct PE schemes:

  PE schemes based on hierarchical data structures not achieving perfect correctness.

- Dynamic PE schemes:

  PE schemes based on dynamic puncturable encryption as introduced in chapter 7.

For the definition of perfect correctness we refer to chapter 3 and for the definition of a hierarchical data structure in the context of PE we refer to section 6.2.

2.3 Symmetric Encryption

The formal definition of a standard symmetric encryption scheme provides the basis for the definitions of SPE schemes.

2.3.1 Syntax

Definition 2.2 *A* Symmetric Encryption *(SE) scheme with message-space $\mathcal{M}$, secret-key-space $\mathcal{K}$ and ciphertext-space $\mathcal{C}$, consists of a triple of algorithms* (SE.KeyGen, SE.Enc, SE.Dec) *with the following syntax:*

- $SK \leftarrow_\$ \text{SE.KeyGen}(\lambda)$*: Given a security parameter λ the* SE.KeyGen *algorithm outputs a secret-key $SK \in \mathcal{K}$.*

- $C \leftarrow \text{SE.Enc}(SK, M)$*: Given a secret-key $SK \in \mathcal{K}$ and a message $M \in \mathcal{M}$,* SE.Enc *outputs the encrypted message $C \in \mathcal{C}$.*

- $M \leftarrow$ SE.Dec(SK, C): *Given a secret-key* $SK \in \mathcal{K}$ *and a ciphertext* $C \in \mathcal{C}$, SE.Dec *outputs the decrypted message* M.

2.3.2 Correctness

Definition 2.3 *We say a symmetric encryption scheme given by* (SE.KeyGen, SE.Enc, SE.Dec) *is correct, if for all* $M \in \mathcal{M}$ *and for all* $SK \leftarrow_\$$ SE.KeyGen(λ) *we have that*

$$Pr[M == \text{SE.Dec}(SK, \text{SE.Enc}(SK, M))] = 1$$

2.4 Public-key Encryption

Similar to symmetric encryption, we provide the definition of a standard public-key encryption scheme as a basis for the definitions of a PkPE scheme.

2.4.1 Syntax

Definition 2.4 *A* Public-key Encryption *(*PkE*) scheme with message-space* $\mathcal{M}$, *secret-key-space* $\mathcal{K}_1$, *public-key-space* $\mathcal{K}_2$ *and ciphertext-space* $\mathcal{C}$, *consists of a triple of algorithms* (PkE.KeyGen, PkE.Enc, PkE.Dec) *with the following syntax:*

- $(PK, SK) \leftarrow_\$$ PkE.KeyGen(λ): *Given a security parameter* λ *the* PkE.KeyGen *algorithm outputs a public-key* $PK \in \mathcal{K}_2$ *and secret-key* $SK \in \mathcal{K}_1$.
- $C \leftarrow$ PkE.Enc(PK, M): *Given a public-key* $PK \in \mathcal{K}_2$ *and a message* $M \in \mathcal{M}$, PkE.Enc *outputs the encrypted message* $C \in \mathcal{C}$.
- $M \leftarrow$ PkE.Dec(SK, C): *Given a secret-key* $SK \in \mathcal{K}_1$ *and a ciphertext* $C \in \mathcal{C}$, PkE.Dec *outputs the decrypted message* M.

2.4.2 Correctness

Definition 2.5 *We say a public-key encryption scheme given by* (PkE.KeyGen, PkE.Enc, PkE.Dec) *is correct, if* $\forall\ M \in \mathcal{M}$ *and* $\forall\ (PK, SK) \leftarrow_\$$ PkE.KeyGen(λ) *we have that*

$$Pr[M == \text{PkE.Dec}(SK, \text{PkE.Enc}(PK, M))] = 1$$

2.5 Security

To get a basis for the security definitions for PE schemes, we provide game-based security notions for both indistinguishability under chosen-plaintext attacks (ind-cpa) as well as indistinguishability under chosen-ciphertext attacks (ind-cca) for SE schemes as well as PkE schemes.

We use the following definitions for games and the advantage of an adversary playing a game.

Definition 2.6 *We denote a game X using scheme Y by* $\mathbf{G}_Y^X(\lambda)$*, where* λ *denotes the security parameter. Further, we denote the fact that a game* $\mathbf{G}_Y^X(\lambda)$ *produces an output z by* $z \Leftarrow \mathbf{G}_Y^X(\lambda)$*.*

Definition 2.7 $\mathbf{Adv}_Y^X(\mathcal{A}, \lambda)$ *denotes the advantage an adversary* $\mathcal{A}$ *has when playing a game X using scheme Y. We define it by*

$$\mathbf{Adv}_Y^X(\mathcal{A}, \lambda) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}_Y^X(\lambda)\right] - \frac{1}{2}\right|.$$

Note that we scale the advantage by subtracting $\frac{1}{2}$. We do this because all games we will define have binary outputs (either *True* or *False*). Therefore, an adversary who randomly guesses the output value will achieve an advantage of 0. In addition, we multiply the scaled absolute value by 2 such that the maximum advantage is 1.

2.5.1 Symmetric IND-CPA

The game begins by choosing a random bit b which later on decides which message we are going to encrypt. In a next step we initialize the key by running the SE.KeyGen algorithm on a security parameter λ. Now the adversary takes over, which is modelled as a program $\mathcal{A}$, taking no inputs, but having access to the SE.Enc algorithm through a subroutine Challenge which we call the Challenge Oracle. This subroutine takes as input two messages M_0 and M_1 of same length, i.e. $|M_0| = |M_1|$, and returns a ciphertext C. We require that both messages are elements of the messages-space $\mathcal{M}$. The game is formally described in figure 2.1.

The adversary $\mathcal{A}$ wins the game $\mathbf{G}_{\text{SE}}^{\text{ind-cpa}}(\lambda)$ if the output b^* matches the randomly selected bit b. Using Definition 2.7 we arrive at

$$\mathbf{Adv}_{\text{SE}}^{\text{ind-cpa}}(\mathcal{A}, \lambda) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}_{\text{SE}}^{\text{ind-cpa}}(\lambda, n)\right] - \frac{1}{2}\right|.$$

We are now able to formally define ind-cpa security for a SE scheme as follows:

Definition 2.8 *We consider a symmetric encryption scheme* ind-cpa *secure, if there exists a negligible function* $negl : \mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}_{\text{SE}}^{\text{ind-cpa}}(\mathcal{A}, \lambda) \leq negl(\lambda).$$

Game $\mathbf{G}_{\text{SE}}^{\text{ind-cpa}}(\lambda)$	Challenge(M_0, M_1):
1 $b \leftarrow\$ \{0,1\}$	5 $C \leftarrow\$ \text{SE.Enc}(SK, M_b)$
2 $SK \leftarrow\$ \text{SE.KeyGen}(\lambda)$	6 Return C
3 $b^* \leftarrow\$ \mathcal{A}^{\text{Challenge}(\cdot,\cdot)}()$	
4 Return $b == b^*$	

Figure 2.1: Game formalizing ind-cpa security of a SE scheme.

Game $\mathbf{G}_{\text{SE}}^{\text{ind-cca}}(\lambda)$	Challenge(M_0, M_1):
1 $\mathcal{CT} \leftarrow \emptyset$	6 $C \leftarrow\$ \text{SE.Enc}(SK, M_b)$
2 $b \leftarrow\$ \{0,1\}$	7 $\mathcal{CT} \leftarrow \mathcal{CT} \cup \{C\}$
3 $SK \leftarrow\$ \text{SE.KeyGen}(\lambda)$	8 Return C
4 $b^* \leftarrow\$ \mathcal{A}^{\text{Challenge}(\cdot,\cdot)}_{\text{Dec}(\cdot)}()$	Dec(C):
5 Return $b == b^*$	9 if $C \in \mathcal{CT}$: Return $\perp$
	10 $M \leftarrow \text{SE.Dec}(SK, C)$
	11 Return M

Figure 2.2: Game formalizing ind-cca security of a SE scheme.

2.5.2 Symmetric IND-CCA

To arrive at chosen-ciphertext security (ind-cca) for SE we need to adapt the game given in figure 2.1 to allow an adversary to decrypt chosen ciphertexts. To include this functionality, we provide a new subroutine Dec, called the Decryption Oracle. Using this new oracle, the adversary can decrypt any ciphertext C, unless C corresponds to a message encrypted by the adversary using the Challenge subroutine[1]. Otherwise, the game follows the same principle as $\mathbf{G}_{\text{SE}}^{\text{ind-cpa}}(\lambda, n)$. We call this game $\mathbf{G}_{\text{SE}}^{\text{ind-cca}}(\lambda, n)$ and formally define it in figure 2.2. Following definition 2.7 we define the advantage of an adversary $\mathcal{A}$ playing $\mathbf{G}_{\text{SE}}^{\text{ind-cca}}(\lambda)$ by

$$\mathbf{Adv}_{\text{SE}}^{\text{ind-cca}}(\mathcal{A}, \lambda) = 2 \left| Pr\left[True \Longleftarrow \mathbf{G}_{\text{SE}}^{\text{ind-cca}}(\lambda) \right] - \frac{1}{2} \right| .$$

This now allows us to formally define the ind-cca security as follows:

[1] This is to prevent the trivial attack of decrypting the ciphertext received by the Challenge subroutine to guess b with probability 1.

Game $\mathbf{G}^{\text{ind-cpa}}_{\text{PKE}}(\lambda)$

1 $b \leftarrow\$ \{0,1\}$
2 $(PK, SK) \leftarrow\$ \text{PKE.KEYGEN}(\lambda)$
3 $b^* \leftarrow\$ \mathcal{A}^{\text{CHALLENGE}(\cdot,\cdot)}(PK)$
4 Return $b == b^*$

CHALLENGE(M_0, M_1):

5 $C \leftarrow\$ \text{PKE.ENC}(PK, M_b)$
6 Return C

Figure 2.3: Game formalizing ind-cpa security of a PKE scheme.

Definition 2.9 *We say a* SYMMETRIC PUNCTURABLE ENCRYPTION *scheme is* ind-cca *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{ind-cca}}_{\text{SE}}(\mathcal{A}, \lambda) \leq negl(\lambda).$$

2.5.3 Asymmetric IND-CPA

The game used for asymmetric ind-cpa security works as follows. First we sample a random bit b which later on decides which message gets encrypted in all subsequent calls of the CHALLENGE subroutine. In a next step, we use the game parameters λ, whereby λ is a security parameter, to generate the public-key PK and the secret key SK. Now the adversary is run, having access to the public-key as well as the challenge subroutine. The access to PK allows an adversary to encrypt any message, therefore an encryption oracle is not necessary in this game. The challenge subroutine works similarly as the one in the symmetric game. The adversary submits two messages M_0 and M_1 from the message-space $\mathcal{M}$ of same length, i.e. $|M_0| = |M_1|$. The subroutine returns the ciphertext C corresponding to the encryption of M_b. In a last step, the adversary makes a guess b^* of the randomly sampled bit b. He wins if $b == b^*$ and looses otherwise. The game is formally described in figure 2.3. We define the advantage of an adversary playing this game by

$$\mathbf{Adv}^{\text{ind-cpa}}_{\text{PKE}}(\mathcal{A}, \lambda) = 2\left|Pr\left[True \Longleftarrow \mathbf{G}^{\text{ind-cpa}}_{\text{PKE}}(\lambda)\right] - \frac{1}{2}\right|.$$

and arrive at the following definition regarding ind-cpa security of a PKE scheme:

Definition 2.10 *We say a public-key encryption scheme is* ind-cpa *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\text{PKE}}(\mathcal{A}, \lambda) \leq negl(\lambda).$$

Game $\mathbf{G}^{\text{ind-cca}}_{\text{PKE}}(\lambda)$	$\text{DEC}(C)$:
1 $b \leftarrow\$ \{0,1\}$	6 if $C \in \mathcal{CT}$: Return $\perp$
2 $\mathcal{CT} \leftarrow \emptyset$	7 $M \leftarrow \text{PKE.DEC}(SK, C)$
3 $(PK, SK) \leftarrow\$ \text{PKE.KEYGEN}(\lambda)$	8 Return M
4 $b^* \leftarrow\$ \mathcal{A}^{\text{CHALLENGE}(\cdot,\cdot)}_{\text{DEC}(\cdot)}(PK)$	$\text{CHALLENGE}(M_0, M_1)$:
5 Return $b == b^*$	9 $C \leftarrow\$ \text{PKE.ENC}(SK, M_b)$
	10 $\mathcal{CT} \leftarrow \mathcal{CT} \cup \{C\}$
	11 Return C

Figure 2.4: Game formalizing ind-cca security of a PKE scheme.

2.5.4 Asymmetric IND-CCA

In the same way we extended $\mathbf{G}^{\text{ind-cpa}}_{\text{SE}}(\lambda)$ to $\mathbf{G}^{\text{ind-cca}}_{\text{SE}}(\lambda)$ we will now extend $\mathbf{G}^{\text{ind-cpa}}_{\text{PKE}}(\lambda)$ to a new game called $\mathbf{G}^{\text{ind-cca}}_{\text{PKE}}(\lambda)$. To arrive at this extension, we need to add a new oracle DEC, called the DECRYPTION ORACLE. Using this oracle, the adversary has access to the PKE.DEC algorithm. The game follows the same structure as the ind-cpa game, but additionally we initialize a control set $\mathcal{CT}$ at the start of the game. The new oracle, the adversary can use, works as follows. On input a ciphertext C, it returns the message M corresponding to the decryption of C given the current secret key, if $C \notin \mathcal{CT}$. The CHALLENGE ORACLE works in the same way it does in the ind-cpa game, but additionally adds the outputted ciphertext C to the control set $\mathcal{CT}$ [2]. At the end, the adversary makes a guess b^* to indicate which bit b we might have used to encrypt the message(s) in the CHALLENGE subroutine and wins if $b == b^*$ (see figure 2.4). We define the advantage of an adversary playing this game by

$$\mathbf{Adv}^{\text{ind-cca}}_{\text{PKE}}(\mathcal{A}, \lambda) = 2 \left| Pr\left[True \Longleftarrow \mathbf{G}^{\text{ind-cca}}_{\text{PKE}}(\lambda) \right] - \frac{1}{2} \right|,$$

and arrive at the following definition regarding ind-cca security of a PKE scheme:

Definition 2.11 *We say a public-key encryption scheme is* ind-cca *secure, if there exists a negligible function negl* : $\mathbb{N} \to \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{ind-cca}}_{\text{PKE}}(\mathcal{A}, \lambda) \leq negl(\lambda).$$

[2] This prevents a trivial attack.

Chapter 3

Puncturable Encryption

Based on the definitions of SE and PkE we are now able to formally define symmetric PE (SPE) and public-key PE (PkPE). Our definitions are inspired by the original definitions introduced by Green and Miers [10] and adapted to the ideas given in the definitions from Sun et al. [13], to also allow puncturing on an unbounded number of tags. Further, we will only provide the definitions for tag-based puncturable encryption. By using one tag for one message only, the definitions will be equivalent to the puncturing on a message approach, since by puncturing on the message specific tag we revoke the decryption capability for only the corresponding message, therefore resulting in the same decryption capabilities as if we punctured on the message itself.

3.1 Symmetric Puncturable Encryption

Definition 3.1 *A tag-based* Symmetric Puncturable Encryption (SPE) *scheme with tag-space $\mathcal{T}$, message-space $\mathcal{M}$, key-space $\mathcal{K}$, and ciphertext-space $\mathcal{C}$, consists of a tuple of four algorithms (*SPE.KeyGen, SPE.Enc, SPE.Dec, SPE.Punct*) with the following syntax:*

- $SK_0 \leftarrow\!\!\$\ \text{SPE.KeyGen}(\lambda, n, params)$: *Given a security parameter λ and a maximum number of tags n, where $n \in \mathbb{N} \cup \{\infty\}$ and ´∞´ corresponds to an unbounded number of tags, as well as an array params holding additional parameters if needed, the* SPE.KeyGen *algorithm outputs an initial secret key $SK_0 \in \mathcal{K}$.*
- $C \leftarrow\!\!\$\ \text{SPE.Enc}(SK_i, M, \tau)$: *On input a secret key $SK_i \in \mathcal{K}$, a message $M \in \mathcal{M}$ and a tag $\tau \in \mathcal{T}$, the* SPE.Enc *algorithm outputs a ciphertext $C \in \mathcal{C} \cup \{\perp\}$, where $\perp$ indicates that the encryption failed.*
- $M \leftarrow \text{SPE.Dec}(SK_i, C, \tau)$: *Given a secret-key $SK_i \in \mathcal{K}$, a ciphertext $C \in \mathcal{C}$, and a tag $\tau \in \mathcal{T}$, corresponding to C, the* SPE.Dec *algorithm outputs $M \in$*

$\{\mathcal{M}\} \cup \{\perp\}$, *where* $'\perp'$ *indicates that the decryption failed.*

- $SK_i \leftarrow$SPE.Punct(SK_{i-1}, τ)*: Takes as input a secret-key* $SK_{i-1} \in \mathcal{K}$ *and a tag* $\tau \in \mathcal{T}$ *and outputs a (new*[1]*) secret-key* $SK_i \in \mathcal{K}$.

Using our definition given above, we define perfect correctness of a SPE scheme as follows:

Definition 3.2 *We say that a tag-based* Symmetric Puncturable Encryption *scheme given by (*SPE.KeyGen, SPE.Enc, SPE.Dec, SPE.Punct*) with tag-space* $\mathcal{T}$*, message-space* $\mathcal{M}$*, key-space* $\mathcal{K}$*, and ciphertext-space* $\mathcal{C}$ *achieves perfect correctness, if* $\forall\lambda$, $\forall n \in \mathbb{N} \cup \{\infty\}$, *and* $\forall\tau \in \mathcal{T}$, $SK_0 \leftarrow_{\$}$ SPE.KeyGen$(\lambda, n, params)$, *and* $C \leftarrow_{\$}$ SPE.Enc(SK_i, M, τ), *where* $M \in \mathcal{M}$, $C \in \mathcal{C}$, *and* $i \in \{0, \ldots, n\}$ *we have that*

- $Pr[M ==$ SPE.Dec$(SK_0,$SPE.Enc$(SK_0, M, \tau), \tau)] = 1$
- *For all sequences of* SPE.Punct *calls* $SK_i \leftarrow$SPE.Punct(SK_{i-1}, τ_i)*, for* $i = 1$ *to* n*, we have that* $Pr[M ==$ SPE.Dec$(SK_i, C, \tau)] = 1$*, if* $\tau \notin \bigcup_i \tau_i$.

Additionally, we define a less strict correctness notion which we call *relaxed correctness*, allowing for a negligible correctness error.

Definition 3.3 *We say that a tag-based* Symmetric Puncturable Encryption *scheme given by (*SPE.KeyGen, SPE.Enc, SPE.Dec, SPE.Punct*) with tag-space* $\mathcal{T}$*, message-space* $\mathcal{M}$*, key-space* $\mathcal{K}$*, and ciphertext-space* $\mathcal{C}$ *achieves relaxed correctness, if* $\forall\lambda$, $\forall n \in \mathbb{N} \cup \{\infty\}$, *and* $\forall\tau \in \mathcal{T}$, $SK_0 \leftarrow_{\$}$ SPE.KeyGen$(\lambda, n, params)$, *and* $C \leftarrow_{\$}$ SPE.Enc(SK_i, M, τ), *where* $M \in \mathcal{M}$, $C \in \mathcal{C}$, *and* $i \in \{0, \ldots, n\}$, *there exists a negligible function* $negl : \mathbb{N} \rightarrow \mathbb{R}$ *such that*

- $Pr[M ==$ SPE.Dec$(SK_0,$SPE.Enc$(SK_0, M, \tau), \tau)] = 1$
- *For all sequences of* SPE.Punct *calls* $SK_i \leftarrow$SPE.Punct(SK_{i-1}, τ_i)*, for* $i = 1$ *to* n*, we have that* $Pr[M ==$ SPE.Dec$(SK_i, C, \tau)] = 1 - negl(\lambda)$*, if* $\tau \notin \bigcup_i \tau_i$*, i.e. we allow for a negligible correctness error introduced by puncturing calls*[2].

3.2 Public-key Puncturable Encryption

Definition 3.4 *A* public-key Puncturable Encryption (PkPE) *scheme based on tags with tag-space* $\mathcal{T}$*, message-space* $\mathcal{M}$*, secret-key-space* $\mathcal{K}_1$*, public-key-space* $\mathcal{K}_2$*, and ciphertext-space* $\mathcal{C}$*, consists of a tuple of four algorithms (*PkPE.KeyGen, PkPE.Enc, PkPE.Dec, PkPE.Punct*) with the following syntax:*

[1] On unexpected input, the unchanged secret-key gets returned.

[2] This means it is possible to not be able to decrypt messages associated to an unpunctured tag τ.

- $(PK, SK_0) \leftarrow_{\$} \text{PkPE.KeyGen}(\lambda, n, params)$: *Given a security parameter* λ *and a maximum number of tags* n, *where* $n \in \mathbb{N} \cup \{\infty\}$ *and* $'\infty'$ *corresponds to an unbounded number of tags as well as an array params holding additional parameters if needed, the* PkPE.KeyGen *algorithm outputs an initial secret-key* $SK_0 \in \mathcal{K}_1$ *and a public-key* $PK \in \mathcal{K}_2$.

- $C \leftarrow_{\$} \text{PkPE.Enc}(PK, M, \tau)$: *On input a public-key PK, a message* $M \in \mathcal{M}$ *and a tag* $\tau \in \mathcal{T}$, *the* PkPE.Enc *algorithm outputs the ciphertext* $C \in \mathcal{C} \cup \{\perp\}$, *where* $\perp$ *indicates that the encryption failed.*

- $M \leftarrow \text{PkPE.Dec}(SK_i, C, \tau)$: *Given a secret-key* $SK_i \in \mathcal{K}_1$, *a ciphertext* $C \in \mathcal{C}$, *and a tag* $\tau \in \mathcal{T}$, *corresponding to* C, *the* PkPE.Dec *algorithm outputs* $M \in \{\mathcal{M}\} \cup \{\perp\}$, *where* $'\perp'$ *indicates that the decryption failed.*

- $SK_i \leftarrow \text{PkPE.Punct}(SK_{i-1}, \tau)$: *Takes as input a secret-key* $SK_{i-1} \in \mathcal{K}_1$ *and a tag* $\tau \in \mathcal{T}$ *and outputs a (new) secret-key* $SK_i \in \mathcal{K}_1$.

Next, we formally define perfect correctness of a PkPE scheme.

Definition 3.5 *We say that a tag-based* public-key Puncturable Encryption *scheme given by (*PkPE.KeyGen, PkPE.Enc, PkPE.Dec, PkPE.Punct*) with tag-space* $\mathcal{T}$, *message-space* $\mathcal{M}$, *secret-key-space* $\mathcal{K}_1$, *public-key-space* $\mathcal{K}_2$, *and ciphertext-space* $\mathcal{C}$ *achieves perfect correctness, if for all* λ, $\forall n \in \mathbb{N} \cup \{\infty\}$, *and for all* $\tau \in \mathcal{T}$, $(PK, SK_0) \leftarrow_{\$} \text{PkPE.KeyGen}(\lambda, n, params)$, *and* $C \leftarrow_{\$} \text{PkPE.Enc}(PK, M, \tau)$, *where* $M \in \mathcal{M}$ *and* $C \in \mathcal{C}$, *we have that*

- $Pr[M == \text{PkPE.Dec}(SK_0, \text{PkPE.Enc}(PK, M, \tau), \tau)] = 1$

- *For all sequences of* PkPE.Punct *calls* $SK_i \leftarrow \text{PkPE.Punct}(SK_{i-1}, \tau_i)$, *for* $i = 1$ *to* n, *we have that* $Pr[M == \text{PkPE.Dec}(SK_i, C, \tau)] = 1$, *if* $\tau \notin \bigcup_i \tau_i$.

Additionally, we define relaxed correctness for a public-key PE scheme as follows:

Definition 3.6 *We say that a tag-based* public-key Puncturable Encryption *scheme given by (*PkPE.KeyGen, PkPE.Enc, PkPE.Dec, PkPE.Punct*) with tag-space* $\mathcal{T}$, *message-space* $\mathcal{M}$, *secret-key-space* $\mathcal{K}_1$, *public-key-space* $\mathcal{K}_2$, *and ciphertext-space* $\mathcal{C}$ *achieves relaxed correctness, if for all* λ, $\forall n \in \mathbb{N} \cup \{\infty\}$, *and for all* $\tau \in \mathcal{T}$, $(PK, SK_0) \leftarrow_{\$} \text{PkPE.KeyGen}(\lambda, n, params)$, *and* $C \leftarrow_{\$} \text{PkPE.Enc}(PK, M, \tau)$, *where* $M \in \mathcal{M}$ *and* $C \in \mathcal{C}$, *there exists a negligible function* $negl : \mathbb{N} \rightarrow \mathbb{R}$ *such that*

- $Pr[M == \text{PkPE.Dec}(SK_0, \text{PkPE.Enc}(PK, M, \tau), \tau)] = 1$

- *For all sequences of* PkPE.Punct *calls* $SK_i \leftarrow \text{PkPE.Punct}(SK_{i-1}, \tau_i)$, *for* $i = 1$ *to* n, *we have that* $Pr[M == \text{PkPE.Dec}(SK_i, C, \tau)] = 1 - negl(\lambda)$, *if* $\tau \notin \bigcup_i \tau_i$.

3.3 Security

Based on the security definitions presented in 2.5 we provide game-based security notions for ind-cpa and ind-cca security for both symmetric and public-key puncturable encryption schemes. Since PE schemes have an additional attribute, the number of supported tags n, we need to adapt our game and advantage definitions as follows:

Definition 3.7 *We denote a game X using scheme Y by* $\mathbf{G}_Y^X(\lambda, n)$*, where* λ *denotes the security parameter and n denotes the maximum number of tags. Further, we denote the fact that a game* $\mathbf{G}_Y^X(\lambda, n)$ *produces an output z by* $z \Leftarrow \mathbf{G}_Y^X(\lambda, n)$*.*

Definition 3.8 $\mathbf{Adv}_Y^X(\mathcal{A}, \lambda, n)$ *denotes the advantage an adversary* $\mathcal{A}$ *has when playing a game X using scheme Y. We define it by*

$$\mathbf{Adv}_Y^X(\mathcal{A}, \lambda, n) = 2\left|Pr\left[True \Longleftarrow \mathbf{G}_Y^X(\lambda, n)\right] - \frac{1}{2}\right|.$$

3.3.1 Symmetric IND-CPA including FS

We need to adapt the game described in Figure 2.1 to also allow puncturing and to give the adversary access to the punctured key at a certain time in the game. We include these functionalities by defining three subroutines, namely PUNCT, which we will call the PUNCTURING ORACLE, CORR, which we call the CORRUPTION ORACLE, and a CHALLENGE subroutine, CHALLENGE. All three subroutines will be described in more detail in the following description of the game, which we will call $\mathbf{G}_{\text{SPE}}^{\text{fs-ind-cpa}}(\lambda, n)$. Our game follows a similar construction idea to the one describe by Susilo et al. [15] and the one by Sun et al. [13].

Analogously to the ind-cpa game, the game starts by choosing a random bit b followed by running the SPE.KEYGEN algorithm to initialize the key $\mathcal{SK}_0$. Then the adversary $\mathcal{A}$ is run, having access to the PUNCTURING ORACLE, PUNCT, the CORRUPTION ORACLE, CORR, as well as the CHALLENGE subroutine. Using PUNCT the adversary can puncture the key on an arbitrary tag $\tau \in \mathcal{T}$ whereby the PUNCTURING ORACLE keeps track of the number of puncturings performed as well as maintaining a list of tags, $\mathcal{PT}$, on which the key is already punctured on. We initialize $\mathcal{PT} \leftarrow \emptyset$ at the start of the game. At any point during the game, the adversary may decide to call CORR. The game rejects corruption if the current secret-key is not punctured on all tags used in calls of the CHALLENGE subroutine before. Otherwise, it returns the most recent punctured key SK_i to the adversary and sets the control set $\mathcal{CS}$ to the current $\mathcal{PT}$ set. Using the CHALLENGE subroutine, the adversary submits two messages $M_0, M_1 \in \mathcal{M}$ of same length and a target tag $\tau \in \mathcal{T}$.

Game $\mathbf{G}_{\text{SPE}}^{\text{fs-ind-cpa}}(\lambda, n)$

1 $b \leftarrow_\$ \{0,1\}$
2 $\mathcal{PT} \leftarrow \emptyset$; $\mathcal{CS} \leftarrow \emptyset$; $ChalT \leftarrow \emptyset$
3 $corrupted \leftarrow False$; $i \leftarrow 0$
4 $SK_0 \leftarrow_\$ \text{SPE.KeyGen}(\lambda, n, params)$
5 $b^* \leftarrow_\$ \mathcal{A}^{\text{Punct}(\cdot),\text{Corr}()}_{\text{Challenge}(\cdot,\cdot,\cdot)}()$
6 Return $b == b^*$

Challenge(M_0, M_1, τ):

7 if *corrupted* and $(\tau \notin \mathcal{CS})$: Return $\perp$
8 $ChalT \leftarrow ChalT \cup \{\tau\}$
9 $C \leftarrow_\$ \text{SPE.Enc}(SK_i, M_b, \tau)$
10 Return C

Punct(τ):

11 $i \leftarrow i + 1$
12 $SK_i \leftarrow \text{SPE.Punct}(SK_{i-1}, \tau)$
13 $\mathcal{PT} \leftarrow \mathcal{PT} \cup \{\tau\}$
14 Return

Corr():

15 if $ChalT \not\subseteq \mathcal{PT}$: Return $\perp$
16 $corrupted \leftarrow True$
17 $\mathcal{CS} \leftarrow \mathcal{PT}$
18 Return SK_i //current key

Figure 3.1: Game formalizing fs-ind-cpa security of a SPE scheme.

If after calling the Corruption Oracle $\tau \notin \mathcal{CS}$ the challenge gets rejected directly [3]. Otherwise, we encrypt message M_b, for the randomly sampled bit b at the start of the game, under the tag τ and return the ciphertext to the adversary. As a last step the adversary makes a guess b^* to indicate which message we might have chosen to encrypt during the challenge(s) and wins if $b == b^*$. This game is formally described in figure 3.1. Using definition 3.8, we define the advantage an adversary has while playing this game as follows:

$$\mathbf{Adv}_{\text{SPE}}^{\text{fs-ind-cpa}}(\mathcal{A}, \lambda, n) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}_{\text{SPE}}^{\text{fs-ind-cpa}}(\lambda, n)\right] - \frac{1}{2}\right|.$$

From it follows the definition of a fs-ind-cpa secure SPE scheme:

Definition 3.9 *We consider a* Symmetric Puncturable Encryption *scheme* fs-ind-cpa *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}_{\text{SPE}}^{\text{fs-ind-cpa}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

At this point we want to mention that to be sure a message encrypted under a tag τ cannot be decrypted by anyone in the future, both sides of the communication need to puncture the secret-key on tag τ[4].

[3] This prevents trivial attacks after an adversary has called the Corruption Oracle.

[4] This is the case since we use symmetric encryption, and therefore both sides of a communication hold the same key. Note that this is true for all SPE schemes.

Game $\mathbf{G}^{\text{fs-ind-cca}}_{\text{SPE}}(\lambda, n)$

```
1  b ←$ {0,1}; ChalT ← ∅
2  PT ← ∅; CS ← ∅; CT ← ∅
3  corrupted ← False; i ← 0
4  SK_0 ←$ SPE.KeyGen(λ, n, params)
5  b* ←$ A^{Punct(·),Corr()}_{Dec(·,·),Challenge(·,·,·)}()
6  Return b == b*
```

$\textsc{Dec}(C, \tau)$:

```
7  if C ∈ CT: Return ⊥
8  M ← SPE.Dec(SK_i, C, τ)
9  Return M
```

$\textsc{Challenge}(M_0, M_1, \tau)$:

```
10 if corrupted and (τ ∉ CS): Return ⊥
11 ChalT ← ChalT ∪ {τ}
12 C ←$ SPE.Enc(SK_i, M_b, τ)
13 CT ← CT ∪ {C}
14 Return C
```

$\textsc{Punct}(\tau)$:

```
15 i ← i + 1
16 SK_i ← SPE.Punct(SK_{i-1}, τ)
17 PT ← PT ∪ {τ}
18 Return
```

$\textsc{Corr}()$:

```
19 if ChalT ⊄ PT: Return: ⊥
20 corrupted ← True
21 CS ← PT
22 Return SK_i //current key
```

Figure 3.2: Game formalizing fs-ind-cca security of a SPE scheme.

3.3.2 Symmetric IND-CCA including FS

Using the same ideas we used to extended $\mathbf{G}^{\text{ind-cpa}}_{\text{SE}}(\lambda)$ to $\mathbf{G}^{\text{ind-cca}}_{\text{SE}}(\lambda)$, we extend $\mathbf{G}^{\text{fs-ind-cpa}}_{\text{SPE}}(\lambda, n)$ to $\mathbf{G}^{\text{fs-ind-cca}}_{\text{SPE}}(\lambda, n)$ by adding the Decryption Oracle. The game is given in figure 3.2.

We arrive at the following notion of the advantage an adversary has while playing the game.

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{SPE}}(\mathcal{A}, \lambda, n) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}^{\text{fs-ind-cca}}_{\text{SPE}}(\lambda, n) \right] - \frac{1}{2} \right|.$$

From it follows the fs-ind-cca security of a SPE scheme.

Definition 3.10 *We say a* Symmetric Puncturable Encryption *scheme is* fs-ind-cca *secure, if there exists a negligible function negl* : $\mathbb{N} \to \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{SPE}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

3.3.3 Asymmetric IND-CPA including FS

To extend $\mathbf{G}^{\text{ind-cpa}}_{\text{PkE}}(\lambda)$ to include forward security, we need to provide the adversary access to the PkPE.Punct algorithm as well as grant it access to the current secret-key at some point in the game. We achieve this in the same way we achieved it in the symmetric case by providing two new oracles, namely the Puncture Oracle, Punct, as well as the Corruption Oracle, Corr. We call this new game $\mathbf{G}^{\text{fs-ind-cpa}}_{\text{PkPE}}(\lambda, n)$ and parametrize it by a security parameter λ and a maximum number of tags n.

The game again starts by sampling a random bit b to decide which message(s) gets encrypted by the Challenge subroutine. We then initialize some control variables such as the sets $\mathcal{PT}$ and $\mathcal{CS}$, a boolean *corrupted*, and a counter i. We then use the PkPE.KeyGen algorithm on the parameters λ and n to generate the public-key as well as the initial secret-key. Then the adversary is run, having access to the Challenge subroutine, the Puncture Oracle, and the Corruption oracle. The Challenge subroutine works in the same way as it does in the asymmetric ind-cpa game, but additionally the challenge gets rejected after corruption if $\tau \notin \mathcal{CS}$. The Puncture Oracle, on input a tag $\tau \in \mathcal{T}$ increases the counter i and calculates the new secret-key SK_i using the PkPE.Punct algorithm on SK_{i-1} and τ. We then add τ to the set of punctured tags $\mathcal{PT}$ and return. The Corruption Oracle takes no inputs. The first time it gets called it sets the control variable *corrupted* to true, copies the set of punctured tags $\mathcal{PT}$ to the control set $\mathcal{CS}$ and returns the current secret-key SK_i to the adversary. All subsequent calls of this subroutine will return $\perp$. At the end of the game, the adversary makes a guess b^* for the random bit b and wins if $b == b^*$. The game is formally described in figure 3.3. We define the advantage of an adversary playing this game by

$$\mathbf{Adv}^{\text{fs-ind-cpa}}_{\text{PkPE}}(\mathcal{A}, \lambda, n) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}^{\text{fs-ind-cpa}}_{\text{PkPE}}(\lambda, n) \right] - \frac{1}{2} \right|.$$

and arrive at the following definition regarding fs-ind-cpa security of a PkPE scheme:

Definition 3.11 *We say a* public-key Puncturable Encryption *scheme is* fs-ind-cpa *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{fs-ind-cpa}}_{\text{PkPE}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

3.3.4 Asymmetric IND-CCA including FS

By providing a Corruption Oracle, Corr, and a Puncturing Oracle, Punct, we extend the $\mathbf{G}^{\text{ind-cca}}_{\text{PkE}}(\lambda)$ game to also include forward security to arrive at a new game which we call $\mathbf{G}^{\text{fs-ind-cca}}_{\text{PkPE}}(\lambda, n)$. The game is formally

Game $\mathbf{G}^{\text{fs-ind-cpa}}_{\text{PkPE}}(\lambda, n)$	$\text{Punct}(\tau)$:
1 $b \leftarrow\$ \{0,1\}$	11 $i \leftarrow i+1$
2 $\mathcal{PT} \leftarrow \emptyset$; $\mathcal{CS} \leftarrow \emptyset$; $ChalT \leftarrow \emptyset$	12 $SK_i \leftarrow \text{PkPE.Punct}(SK_{i-1}, \tau)$
3 $corrupted \leftarrow False$; $i \leftarrow 0$	13 $\mathcal{PT} \leftarrow \mathcal{PT} \cup \{\tau\}$
4 $(PK, SK_0) \leftarrow\$ \text{PkPE.KeyGen}(\lambda, n, params)$	14 Return
5 $b^* \leftarrow\$ \mathcal{A}^{\text{Punct}(\cdot),\text{Corr}()}_{\text{Challenge}(\cdot,\cdot,\cdot)}(PK)$	$\text{Corr}()$:
6 Return $b == b^*$	15 if $ChalT \not\subseteq \mathcal{PT}$: Return $\perp$
$\text{Challenge}(M_0, M_1, \tau)$:	16 $corrupted \leftarrow True$
7 if $corrupted$ and $(\tau \notin \mathcal{CS})$: Return $\perp$	17 $\mathcal{CS} \leftarrow \mathcal{PT}$
8 $ChalT \leftarrow ChalT \cup \{\tau\}$	18 Return SK_i //current key
9 $C \leftarrow\$ \text{PkPE.Enc}(PK, M_b, \tau)$	
10 Return C	

Figure 3.3: Game formalizing fs-ind-cpa security of a PkPE scheme.

described in figure 3.4. We define the advantage of an adversary playing this game by

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{PkPE}}(\mathcal{A}, \lambda, n) = 2 \left| Pr\left[True \Longleftarrow \mathbf{G}^{\text{fs-ind-cca}}_{\text{PkPE}}(\lambda, n)\right] - \frac{1}{2} \right|$$

and arrive at the following definition regarding fs-ind-cca security of a PkPE scheme:

Definition 3.12 *We say a* public-key Puncturable Encryption *scheme is* fs-ind-cca *secure, if there exists a negligible function* $negl : \mathbb{N} \to \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{PkPE}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

3.4 Relevant features and special orderings

To be able to compare different PE schemes, we briefly state which features we consider most relevant and introduce some special puncturing orders which may occur in real world applications and for which certain schemes may achieve benefits.

The goal of new PE schemes would then be to achieve a better performance compared to a baseline achieved by a naïvely constructed PE scheme regarding these features. A scheme performing especially well for a special ordering hints at possibly good performance for special applications. Performing

Game $\mathbf{G}^{\text{fs-ind-cca}}_{\text{PKPE}}(\lambda, n)$

1 $b \leftarrow_\$ \{0,1\}$; $ChalT \leftarrow \emptyset$
2 $\mathcal{PT} \leftarrow \emptyset$; $\mathcal{CS} \leftarrow \emptyset$; $\mathcal{CT} \leftarrow \emptyset$
3 $corrupted \leftarrow False$; $i \leftarrow 0$
4 $(PK, SK_0) \leftarrow_\$ \text{PKPE.KEYGEN}(\lambda, n, params)$
5 $b^* \leftarrow_\$ \mathcal{A}^{\text{PUNCT}(\cdot),\text{CORR}()}_{\text{DEC}(\cdot,\cdot),\text{CHALLENGE}(\cdot,\cdot,\cdot)}(PK)$
6 Return $b == b^*$

$\text{DEC}(C, \tau)$:

7 if $C \in \mathcal{CT}$: Return $\perp$
8 $M \leftarrow \text{PKPE.DEC}(SK_i, C, \tau)$
9 Return M

$\text{CHALLENGE}(M_0, M_1, \tau)$:

10 if $corrupted$ and $(\tau \notin \mathcal{CS})$: Return $\perp$
11 $ChalT \leftarrow ChalT \cup \{\tau\}$
12 $C \leftarrow_\$ \text{PKPE.ENC}(PK, M_b, \tau)$
13 $\mathcal{CT} \leftarrow \mathcal{CT} \cup \{C\}$
14 Return C

$\text{PUNCT}(\tau)$:

15 $i \leftarrow i + 1$
16 $SK_i \leftarrow \text{PKPE.PUNCT}(SK_{i-1}, \tau)$
17 $\mathcal{PT} \leftarrow \mathcal{PT} \cup \{\tau\}$
18 Return

$\text{CORR}()$:

19 if $ChalT \not\subseteq \mathcal{PT}$: Return $\perp$
20 $corrupted \leftarrow True$
21 $\mathcal{CS} \leftarrow \mathcal{PT}$
22 Return SK_i //current key

Figure 3.4: Game formalizing fs-ind-cca security of a PKPE scheme.

well in regard to almost all features would result in a good candidate for general applications using PE.

3.4.1 Features

Relevant features which make up the efficiency of a PE scheme are the following:

- Storage space:

  How much space is needed to store the secret-key and how does it change over time, i.e. growing, shrinking, remaining constant.

- Computation time:

  We compare the speed of the involved algorithms, i.e. encryption, decryption, puncturing and key extension for dynamic schemes (introduced in chapter 7) where necessary.

- Tag support:

  Does the construction support a limited amount of tags, or can the tag-space be unbounded.

- Correctness:

  We differentiate between perfect and relaxed correctness. A scheme with perfect correctness guarantees that decryption under a supported but unpunctured tag is always possible, while in a scheme with relaxed correctness it is possible for decryption to fail despite never puncturing on the associated tag.

- Special case benefits:

  We check if a construction performs especially well under a special ordering of encryption and puncture calls.

3.4.2 In order puncturing

Definition 3.13 *A sequence of puncture calls is called* In Order *if we puncture on the tags in the exact same order in which we used them the first time for encryption. This means if we puncture on a tag τ_i before we puncture on a tag τ_j we also used τ_i to encrypt a message before we used τ_j. This must hold for all combinations of tags τ_i and τ_j.*

3.4.3 Puncturing before next encryption

Definition 3.14 *If a sequence of puncture calls is in order and at any point in time there is at most one tag τ which is used for encryption but not punctured, we call the scenario ´puncturing before next encryption´.*

3.4.4 Quasi-Perfect Ordering

We call a sequence of encryption, decryption, and puncture calls Quasi-Perfect if the tags get punctured in order with respect to a window of size w, whereby we allow for local reorderings inside the window. We mean by this that the tags can be used and punctured in arbitrary order inside the current window.

In figure 3.5 we illustrate what such a quasi-perfect order could look like. Assume τ_2 is the first non-punctured tag, i.e. we already punctured on τ_0 and τ_1. Using a window of size $w = 5$, encryption, decryption and puncturing is allowed for all tags inside the window (depicted by gray nodes).

Figure 3.5: Illustration of quasi-perfect order with window size $w = 5$.

Once we puncture on τ_2 the window advances to the first non-punctured tag in the sequence (as illustrated in figure 3.6). Assume we puncture on τ_5, τ_3, and τ_2 in this order, the window will move forward once we punctured on τ_2 until it reaches τ_4 since it is the first non-punctured tag.

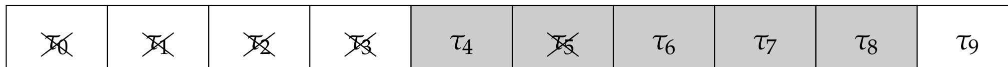


Figure 3.6: Illustration of window movement in quasi-perfect order with window size $w = 5$.

Chapter 4

The Naïve Solution

Before we start analysing hitherto used PE schemes and think about new constructions, we present the naïve way of constructing PE. This naïve construction gives us a baseline we try to beat with our new construction ideas.

To construct the naïve scheme we use black-box encryption schemes, i.e. we assume to have access to an encryption scheme and provide additional functionality to turn it into a PE scheme. The use of such black-box encryption schemes allows us to provide frameworks based on different data structures, which then can be combined with an arbitrary encryption scheme to arrive at a PE scheme. These frameworks then allow to adapt the scheme to individual needs by choosing an appropriate black-box scheme. We can use this idea not only for the naïve solution but also for other constructions of PE schemes.

4.1 Main construction idea

The main idea of the naïve solution is to use one key per tag. Using this simple idea one can implement PE rather easily but sacrifices efficiency in a general setting. By using one key pair for each tag, we arrive at a public-key PE scheme. We will now describe these schemes in more detail.

Initialization

The initialization for a scheme supporting n tags consists of generating one key component for each tag and storing it at the corresponding position inside a secret-key array.

Encryption and Decryption

Encryption (decryption) takes the secret-key array as input. To encrypt (decrypt) under a tag τ, the key component associated to τ is fetched from the

array and used to encrypt (decrypt) the message.

In figure 4.1 we illustrate key component retrieval for encryption (decryption). Assuming we want to encrypt message M under the tag τ_2, we fetch the secret-key SK_2 by accessing the secret-key array at the position indicated by τ_2.

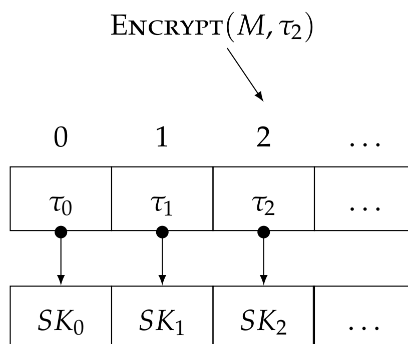


Figure 4.1: Illustration of key component retrieval in the naïve approach for tag τ_2.

Puncturing

To puncture on a tag, we delete its corresponding secret-key component and are therefore unable to decrypt any message encrypted under the punctured tag. We illustrated the effect of puncturing on the underlying data structure in figure 4.2.

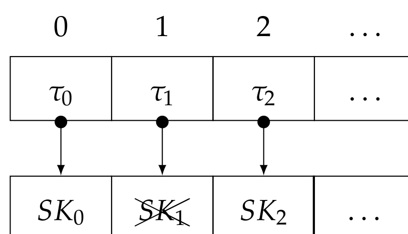


Figure 4.2: Example for puncturing on tag τ_1 in the naïve approach.

4.2 Theoretical construction

We start by providing the theoretical construction of the naïve PE scheme for symmetric encryption. In the next section, we describe how the algorithms

need to be extended to work in a public-key scenario.

4.2.1 Symmetric encryption

Key generation

Our key generation algorithm (in this case the initialization of the scheme) is parametrized by the number of supported tags n and a security parameter λ. It then performs the following steps. It initializes an array which will hold the n precomputed key components. Then it uses the SE.KeyGen function of our black-box encryption scheme to precompute the n key components and stores them in the array. Note that now the mapping from sequence number to key component is fixed and cannot change any more. Finally, we return the array holding the n precomputed key components.

Algorithm 1 Key Generation for naïve SPE

```
1: function KeyGen(λ, n, [])
2:     SK ← Array of length n                    ▷ Initialize array
3:     for i from 0 to n − 1 do                  ▷ Precompute key components
4:         SK[i] ←$ SE.KeyGen(λ)
5:     end for
6:     return SK
7: end function
```

Encryption

For encryption, we compute the number t, τ represents, and use the key component at position t to encrypt the message. If we previously punctured on τ the corresponding entry of the array will hold $\perp$, and the encryption will fail. The algorithm takes no additional arguments, i.e. $params = \emptyset$.

Algorithm 2 Encryption for for naïve SPE

```
1: function Encrypt(SK_i, M, τ)
2:     t ← getIndex(τ)                 ▷ Compute index τ corresponds to
3:     if SK_i[t] == ⊥ then            ▷ Punctured on τ before
4:         return ⊥
5:     end if
6:     C ←$ SE.Enc(SK_i[t], M)         ▷ Encrypt message
7:     return C
8: end function
```

Puncturing

Puncturing on a tag τ consists of replacing the entry corresponding to the number τ represents by $\perp$ and therefore effectively delete the secret-key component associated to τ. At this point we want to mention again that to be sure that a message encrypted using τ cannot be decrypted any more, both sides of a communication need to puncture on τ.

Algorithm 3 Puncturing for naïve SPE

```
1: function Puncture(SK_i, τ)
2:     t ← getIndex(τ)                ▷ Compute index τ corresponds to
3:     SK_i[t] ← ⊥                    ▷ Delete key component corresponding to τ
4:     return SK_i
5: end function
```

Decryption

The decryption algorithm works similarly to the encryption algorithm. We access the key component at the position t, τ indicates, and use it to decrypt our message. If we previously punctured on τ the entry storing the component will hold $\perp$. In this case, decryption will fail and the algorithm returns $\perp$.

Algorithm 4 Decryption for naïve SPE

```
1: function Decrypt(SK_i, C, τ)
2:     t ← getIndex(τ)                ▷ Compute index τ corresponds to
3:     if SK_i[t] == ⊥ then           ▷ Punctured on τ before
4:         return ⊥
5:     end if
6:     M ← SE.Dec(SK_i[t], C)         ▷ Decrypt message
7:     return M
8: end function
```

4.2.2 Public-key encryption

Key generation

Instead of using a black-box symmetric encryption scheme, we use a black-box public-key encryption scheme. On key generation, two key arrays get initialized, one storing the secret-key components and the other storing the public-key components respectively.

Algorithm 5 Key Generation for naïve PkPE

```
1:  function KeyGen(λ, n, [])
2:      SK ← Array of length n                       ▷ Initialize SK array
3:      PK ← Array of length n                       ▷ Initialize PK array
4:      for i from 0 to n − 1 do                     ▷ Precompute key component pairs
5:          (pk, sk) ←$ PkPE.KeyGen(λ)               ▷ Generate key component pairs
6:          PK[i] ← pk
7:          SK[i] ← sk
8:      end for
9:      return (PK, SK)
10: end function
```

Encryption

Instead of using the secret-key component to encrypt the message, the public-key component associate to the given tag is used in a public-key implementation[1].

Algorithm 6 Encryption for naïve PkPE

```
1: function Encrypt(PK, M, τ)
2:     t ← getIndex(τ)                 ▷ Compute index τ corresponds to
3:     C ←$ PkPE.Enc(PK[t], M)         ▷ Encrypt message
4:     return C
5: end function
```

Puncturing

As in the symmetric implementation, we delete the secret-key component associated to the tag on a puncture call.

Algorithm 7 Puncturing for naïve PkPE

```
1: function Puncture(SK_i, τ)
2:     t ← getIndex(τ)                 ▷ Compute index τ corresponds to
3:     SK_i[t] ← ⊥                     ▷ Delete key component corresponding to τ
4:     return SK_i
5: end function
```

Decryption

Using the secret-key component indicated by the tag τ we decrypt an encrypted message if the key is not punctured on τ.

[1]Note that in the public-key scenario we do allow encryption on punctured tags. This has the benefit of a consistent public-key, meaning we do not need to change it after initialization.

Algorithm 8 Decryption for naïve PKPE

```
1: function DECRYPT(SK_i, C, τ)
2:     t ← getIndex(τ)                    ▷ Compute index τ corresponds to
3:     if SK_i[t] == ⊥ then               ▷ Punctured on τ before
4:         return ⊥
5:     end if
6:     M ← PKPE.DEC(SK_i[t], C)           ▷ Decrypt message
7:     return M
8: end function
```

4.3 Theoretical analysis

We first provide a short list of pros and cons of the construction before providing a summary of the features achieved by the naïve PE scheme in figure 4.3, therefore creating a baseline for other constructions to beat.

The naïve PE scheme achieves the following benefits:

- Fast encryption
- Fast decryption
- Fast puncturing
- Shrinking key size on puncturing:

  Key size will be in $\Theta(u)$ where u denotes the number of unpunctured tags.
- Achieves perfect correctness:

  Because it uses one unique key component per tag.

Additionally, we get the following list of drawbacks:

- Key size is linear in the number of supported tags:

  This is due to the one to one mapping of a key and a tag.
- No special order benefits
- No support for unbounded tag-space:

  Since we need to fix the number of supported tags n on initialization, we can only use n different tags. Therefore, an implementation supporting $n \to \infty$ is not possible since the KEYGEN algorithm, called in the initialization process, will not terminate for $n = \infty$.

4.3.1 Feature summary

Variable	Description
n	Number of supported tags
u	Number of unpunctured tags
t	Runtime of black-box SE./PkPE.KeyGen() algorithm
e	Runtime of black-box SE./PkPE.Enc() algorithm
d	Runtime of black-box SE./PkPE.Dec() algorithm

Feature	**Naïve SPE**	**Naïve PkPE**
Secret-Key Storage		
Initial	$\Theta(n)$	$\Theta(n)$
During use	$\Theta(u)$	$\Theta(u)$
Worst case	$\Theta(n)$	$\Theta(n)$
Best case	$\Theta(u)$	$\Theta(u)$
Computation time		
KeyGen()	$\mathcal{O}(nt)$	$\mathcal{O}(nt)$
Encrypt()	$\mathcal{O}(e)$	$\mathcal{O}(e)$
Puncture()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Decrypt()	$\mathcal{O}(d)$	$\mathcal{O}(d)$
Tag support		
Size of tag-space	Bounded	Bounded
Correctness		
Achieves	Perfect	Perfect
Special case benefits		
In order puncturing	No benefits	No benefits
Puncturing before next encryption	No benefits	No benefits
Quasi-perfect ordering	No benefits	No benefits

Figure 4.3: Baseline features achieved by the naïve construction.

Chapter 5

Background and Related Work

In this thesis we try to improve the required secret-key storage and algorithm efficiency of PE schemes. To achieve this goal, we explore different hitherto used constructions and try to extract relevant features a data structure needs to provide to act as a basis of a PE scheme and try to answer the question what a data structure needs to provide to allow compact storage and what does it need to allow for fast retrieval and deletion of key components.

Of specific interest to this is the construction by Derler et al. [6] who propose the use of Bloom filters to construct PE. Compared to prior constructions they achieve shrinking secret-key size and constant size public keys. By studying their scheme, we hope to be able to extract the properties of a probabilistic data structure which can be used to decrease the storage needed for the secret-key by compromising on perfect correctness.

A construction based on perfect binary trees [10, 1, 7] achieves small secret-key storage by relying on a hierarchical derivation of individual key components. By analysing this scheme, we hope to be able to extract the properties of a 'hierarchical' data structure which can be used to decrease the secret-key storage but requires more time to derive the necessary key components.

5.1 PE based on Bloom Filter Encryption

Using Bloom filters to implement PE was originally proposed by Derler et al. [6] and was also subject in more recent work [16, 3]. We use the original construction as the main reference for the analysis.

A BFE-based PE scheme works in the following way. One decides on the size of a Bloom filter, s, and a number of hash functions, k. Depending on the number of supported tags, n, these choices affect the (possibly non-

negligible) correctness error of the resulting scheme[1]. On initialization, one key gets assigned to each Bloom filter entry. All these steps are performed by the KeyGen algorithm of a BFE-based scheme.

To encrypt a message under a tag, one computes the hash values of the tag using the k hash functions and then encrypts the message using the key components associated to these entries in such a way that either one of the key components could be used to decrypt the message. This functionality makes up the Encrypt algorithm of the scheme.

Decryption of a ciphertext works by again calculating the hash values for the involved tag and using the first key component found to decrypt the message. This will be done using the Decrypt algorithm.

Puncturing on a tag τ works by deleting all key components associated to the Bloom filter entries indicated by the k hash values of τ, making it impossible to decrypt any message encrypted using this tag. The functionality is provided by the Puncture algorithm.

A BFE-based PE scheme provides efficient decryption and puncturing, has a secret key that shrinks on puncture calls and provides a possible storage reduction compared to the naïve solution (later on we found out that this assumption is not true).

We will now present the approach in more detail.

5.1.1 Construction

Initialization

To set up a BFE-based PE scheme, we first initialize an empty Bloom filter of size s, where s denotes the number of different keys used, with corresponding k hash functions $h_1, \ldots, h_k$, where each hash function maps a tag τ from the tag-space $\mathcal{T}$ to a number in $\{0, 1, \ldots, s\}$, i.e. $h_i : \mathcal{T} \to \{0, 1, \ldots, s\}$, $i = 1, \ldots, k$. The k hash functions get sampled uniformly at random from a family of hash functions defined from the tag-space to the set $\{0, 1, \ldots, s\}$, whereby we do not allow two hash functions to be the same. Note that depending on how many hash functions are used and how many different key components we provide, we get a trade-off between the storage size needed and the probability that a collision occurs. Depending on the application, one might find different values useful to guarantee an upper bound on the collision probability for a certain number of tags we want to support. We then associate one key component pair to each entry of the Bloom filter. These associations will later on define which key components are used for encryption and decryption of a message corresponding to a given tag. Instead of assigning a key component pair to each entry one could also just

[1]Since these parameters affect the false positive probability of the Bloom filter.

assign one key component to one entry to arrive at a symmetric PE scheme. In this description we consider a public-key implementation, but the high level idea stays the same for a symmetric approach.

Encryption

To encrypt a message M under a tag τ we first compute the Bloom filter entries corresponding to the tag using the k hash functions. We then encrypt the message using the k public-key components associated to the k computed entries in such a way that we can decrypt the encrypted message using an arbitrary one of the k secret-key components associated to τ. If one of the entries is set to 1 we do not use this key component for encryption (since it was deleted by a previous puncturing call, see *Puncturing*). Using ideas of Hierarchy Based Encryption (HBE) one can even achieve a constant size public-key. For more details on this approach, we refer to the original paper by Derler et al. [6].

In the simple example given in figure 5.1 τ corresponds to the BF entries $0, 2$, and n. Therefore, the message M gets encrypted using the public-key components PK_0, PK_2, and PK_n and can later on be decrypted using any of the secret-key components SK_0, SK_2, or SK_n.

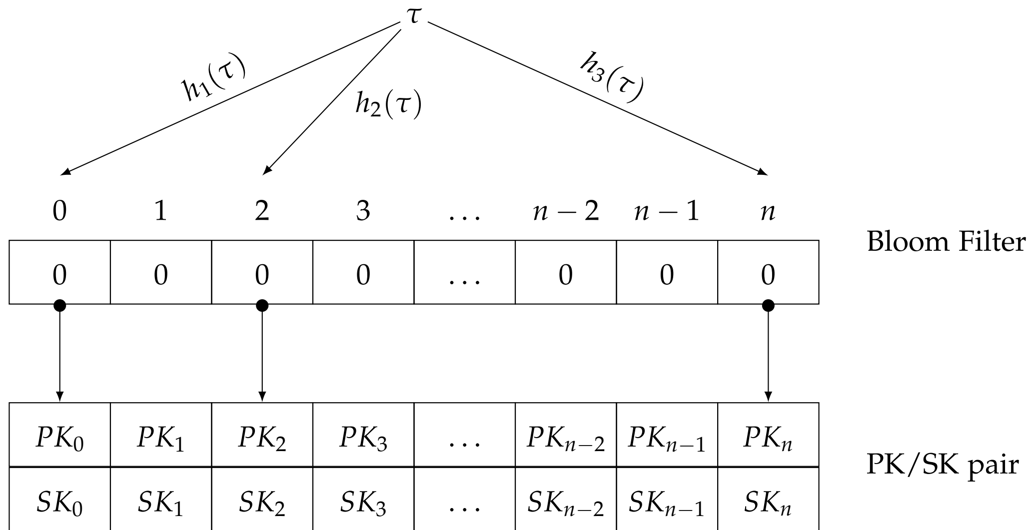


Figure 5.1: Example of key component retrieval for BFE-based PE scheme using $k = 3$ hash functions.

Puncturing

The puncturing algorithm can be implemented rather easily since it just removes all key component pairs associated to the Bloom filter entries of the tag punctured on. In addition to the deletion, we set the corresponding bits inside the Bloom filter to 1 to indicate that the corresponding secret-key component was removed.

The simple example in figure 5.2 shows how puncturing on a tag τ works. We set the Bloom filter entries $0, 2$ and n, corresponding to τ, to 1 and delete the secret-key component SK_0, SK_2 and SK_n.

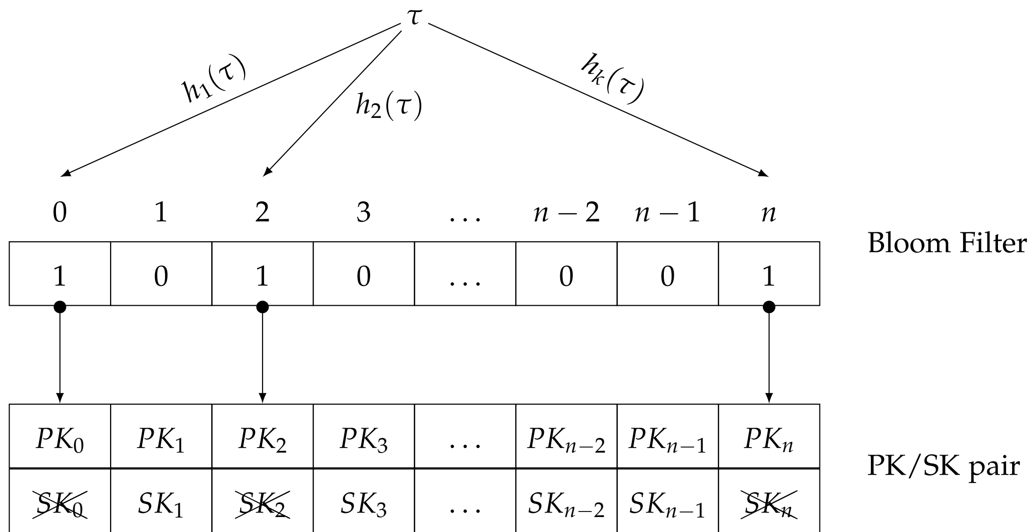


Figure 5.2: Example of puncturing on a tag τ using a BFE-based PE with $k = 3$ hash functions.

Decryption

Decryption follows the same principle as the encryption described above. Given a ciphertext C and a tag τ we compute the Bloom filter entries corresponding to τ and use the secret-key component corresponding to the first 0 entry we find to decrypt C.

This procedure is illustrated in figure 5.3 below. Since the BF entry 0, corresponding to $h_1(\tau)$, is 1, we cannot use SK_0 to decrypt C. Therefore we use SK_2 to decrypt the message since it is the first 0 entry of our computed hashes[2].

[2]Note that decryption using SK_n would also be possible. Which one of the possible secret-keys gets used for decryption is highly dependent on the implementation of the scheme.

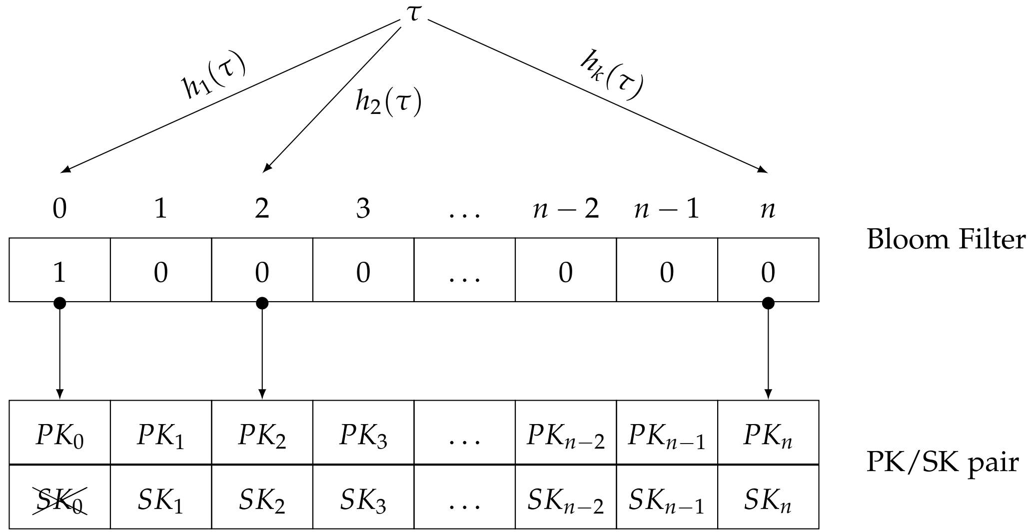


Figure 5.3: Example for decryption on a tag τ using a BFE-based PE with $k = 3$ hash functions.

5.1.2 Theoretical analysis

Providing highly efficient decryption and puncturing (p.1, [6]) and a shrinking key size are the main potential benefits behind the idea of a BFE-based PE scheme. During our analysis, we found that, although providing a shrinking secret-key, a BFE-based PE scheme cannot outperform the naïve solution regarding secret-key storage. We will now describe the reasons why in a little more detail.

Storage problem

In figure 5.4 we illustrated the initial secret-key storage needs of a BFE-based PE scheme usinge $k = 5$ hash functions, achieving an upper bound on the false positive probability of the Bloom filter of $p = 0.5$, and the naïve solution. As we can see the naïve solution requires less storage than a BFE-based scheme for every tag-space size. The gap between the two approaches increases, the smaller we choose p (as can be seen in A.1). To help understand these results, we present the calculation on an example.

- The naïve approach:

  In the naïve approach one needs to store one key component pair for each supported tag. This means that the required storage is linear in the number of supported tags n. Using 256-bit keys and a tag-space $\mathcal{T}$ with $|\mathcal{T}| = 10000$ we end up with a secret-key storage of $256b \cdot 10000 = 2.56Mb$.

- The BFE-based approach:

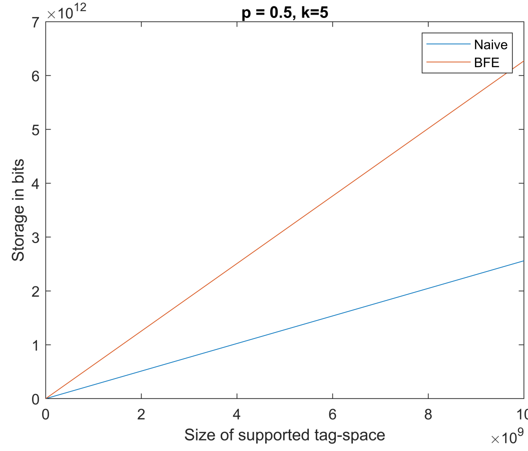


Figure 5.4: Secret-key storage comparison of a BFE-based PE scheme achieving an upper bound on the false positive probability of the Bloom filter of $p = 0.5$ using $k = 5$ hash functions and the naïve solution. Both schemes are using 256-bit secret-keys.

The secret-key storage in a BFE-based approach depends on the number of keys needed to guarantee the upper bound p of the false positive probability of the Bloom filter. After inserting e elements into a Bloom filter of size s using k hash functions, the false positive probability is given by

$$p = \left(1 - \left(1 - \frac{1}{s}\right)^{ek}\right)^{k}.$$

Using this equation we can derive the number of bits, s, a Bloom filter must have, given a maximum false positive rate of p, the number of inserted elements e and a number of hash functions k.

$$s = -\frac{1}{\sqrt[ke]{1 - \sqrt[k]{p}} - 1}$$

Since the false positive probability of the Bloom filter is maximized after inserting all n elements, we calculate the following size s of the Bloom filter to guarantee an upper bound of $p = 0.5$ on the false positive probability using $n = 10000$ and $k = 5$:

$$s = -\frac{1}{\sqrt[5\cdot 10000]{1 - \sqrt[5]{0.5}} - 1} \approx 24457$$

We therefore end up with a secret-key storage, assuming 256-bit keys, of $256b \cdot 24457 \approx 6.26Mb$.

It is important to note that since we delete up to k key components on a puncturing call in a BFE-based scheme, the needed storage decreases faster

compared to the naïve solution. But since for many applications the worst case storage needs are an essential feature, we conclude that a BFE-based scheme cannot outperform a naïve solution in that regard.

Based on our description of a BFE-based PE scheme and taking the storage analysis into account, we can extract the following features:

- Shrinking secret-key:

  After puncturing on a tag, the secret-key can never increase but mostly even shrinks.

- Constant size public-keys:

  Using IBE, the construction described by Derler et al. [6] achieves constant size public-keys[3].

Additionally, we can extract the following list of drawbacks:

- Non-zero false positive probability:

  Due to the nature of a Bloom filter it is possible to puncture on a tag τ by calling the puncture algorithm on different tags $\tau' \neq \tau$. This leads to a possibly non-negligible correctness error.

- No perfect correctness

  Because of the non-zero false positive probability caused by the possible collisions of tags, a BFE-based PE scheme is not able to achieve perfect correctness.

- No gains from special orders:

  Neither in order puncturing nor puncturing before next encryption nor a quasi-perfect puncturing order changes the efficiency or needed storage.

- Number of needed keys:

  As shown, a BFE-based scheme requires a bigger initial key-storage compared to the naïve solution. Additionally, the secret-key size tends to get large for small upper bounds on the false positive probability (as shown in A.1).

- No support for unbounded tag-space:

  Due to its design, it is not possible to define a BFE-based PE scheme which supports an unbounded number of tags without getting a high correctness error (tending to 1).

[3]Using IBE we can also achieve constant size public-keys in a naïve construction.

A complete summary of the features achieved by a BFE-based PE scheme as well as comparison to the naïve solution is presented in figure 5.5. We can see that the naïve solution outperforms a BFE-based scheme regarding all features analysed. It is important to note that for certain choices of p and k, it is theoretically possible to achieve a size $s < n$ for the Bloom filter, but this requires p to be unacceptably large (around 75%). Also we want to mention that the authors presented an implementation achieving constant size public-keys by making use of IBE in the original paper [6]. Using ´identities´ as tags, one can construct a naïve PE scheme using IBE which would achieve the same secret-key storage as the original naïve solution and would also have the benefit of constant-size public-keys. The main functional difference to such a construction would be that a BFE-based scheme is able to support arbitrary tags due to the use of hash functions.

Feature summary and comparison to naïve approach

Variable	Description
n	Number of supported tags
u	Number of unpunctured tags
t	Runtime of black-box PkPE.KeyGen() algorithm
e	Runtime of black-box PkPE.Enc() algorithm
d	Runtime of black-box PkPE.Dec() algorithm
p	Upper bound on the false positive probability of the Bloom filter
k	Amount of hash functions used
h	Time needed to compute one hash value
s	Size of the Bloom filter: $s = -\frac{1}{\sqrt[ke]{1-\sqrt[k]{p}}-1}$

Feature	BFE-based PE	Naïve PkPE
Secret-Key Storage		
Initial	$\Theta(s)$	$\Theta(n)$
During use	$\mathcal{O}(s)$	$\Theta(u)$
Worst case	$\Theta(s)$	$\Theta(n)$
Best case	$\mathcal{O}(s)$	$\Theta(u)$
Computation time		
KeyGen()	$\mathcal{O}(st)$	$\mathcal{O}(nt)$
Encrypt()	$\mathcal{O}(hke)$	$\mathcal{O}(e)$
Puncture()	$\mathcal{O}(hk)$	$\mathcal{O}(1)$
Decrypt()	$\mathcal{O}(hkd)$	$\mathcal{O}(d)$
Tag support		
Size of tag-space	Bounded	Bounded
Correctness		
Achieves	Non-negligible	Perfect
Special case benefits		
In order puncturing	No benefits	No benefits
Puncturing before next encryption	No benefits	No benefits
Quasi-perfect ordering	No benefits	No benefits

Figure 5.5: Features achieved by a BFE-based PE scheme compared to the naïve PkPE scheme. For each feature the better performing scheme is marked green and the worse performing one is marked red.

5.2 Perfect Binary Tree PE

PBT-based PE was used in many different papers [10, 1, 7] which we use as reference for the following analysis.

A PBT-based PE schemes stores the key components associated to the supported tags as leaves of a perfect binary tree. On initialization, one computes a ´root key´ which functions as the root of the tree. Additionally, we need a function f, which maps a tag to its corresponding leaf. To be able to derive the leaves from the root key we also need two one-way functions f_l and f_r, which given a key K from the key-space $\mathcal{K}$ calculate a new key $K' \in \mathcal{K}$, whereby f_l is used to compute the left child and f_r is used to compute the right child respectively.

Encryption (decryption) of a message under a tag τ works by first using f to find the leaf associated to τ and then use the functions f_l and f_r to derive its key component from the root key. We then can use it to encrypt (decrypt) the message.

To puncture on a tag τ the leaf storing the key component associated to τ and all nodes which could be used to derive the leaf, including the root key, get deleted. To still be able to derive the key components associated to unpunctured tags, all children of deleted non-leaf nodes get stored.

Such a PE scheme is able to reduce the needed storage space compare to the naïve approach by not needing to store all key components due to the hierarchical derivation of the leaf nodes. Additionally, it can achieve good performance for some special orderings of puncture calls.

We now present a more detailed description of the approach.

5.2.1 Construction

Initialization

To initialize a PBT-based PE scheme, one first needs to decide on how many tags the scheme should support. We denote the number of supported tags by $n = 2^a$. In a next step, one calculates a *Master secret-key* MSK, which will be the root of the tree. MSK is the initial secret-key generated by $\textsc{KeyGen}(\lambda, n)$, whereby λ defines the length of the key, i.e. 128 bit, 256 bit, etc. In addition, we need to choose two one-way functions f_l and f_r, which given a key K from the key space $\mathcal{K}$ calculate a new key $K' \in \mathcal{K}$, whereby f_l is used to compute the left child and f_r is used to compute the right child of a node respectively. The n different key components used for encryption and decryption correspond to the keys inside the leaves of the tree. Additionally, we define a function f, which given a tag assigns it to its corresponding leaf node. Depending on the output of f, f_l and f_r are used to derive the secret-key component.

Encryption

To encrypt a message $M \in \mathcal{M}$ under a tag τ one first calculates the leaf corresponding to τ using f. For a tree supporting $n = 2^a$ tags, the key can be calculated from the MSK using $\mathcal{O}(a)$, $a = log_2(n)$, calls of the functions f_l and f_r. Note that a is the worst case time needed to access a key component, since after puncturing we will use intermediate nodes as starting points. Using the derived key component, one can encrypt the message.

In figure 5.6 one can see a simple example on how we get an encryption key. Let $n = 4$, assume the tag space $\mathcal{T} = \{0,1\}^2$ be the set of all bit strings of length two, and let $f : \{0,1\}^2 \rightarrow \{0,1\}^2$ be the identity function. Given a tag $\tau = 01$ we use f to get the path through the tree, i.e. $f(\tau) = \tau = 01$. We then use f_l and f_r to derive the intermediate nodes, whereby we use f_l if the current bit is 0 and f_r if the current bit is 1. Once arrived at the leaf node, we use its key component, in this case SK_1, to encrypt the message.

For better understanding, we provide a pseudocode implementation of the encryption algorithm (see 5.2.1).

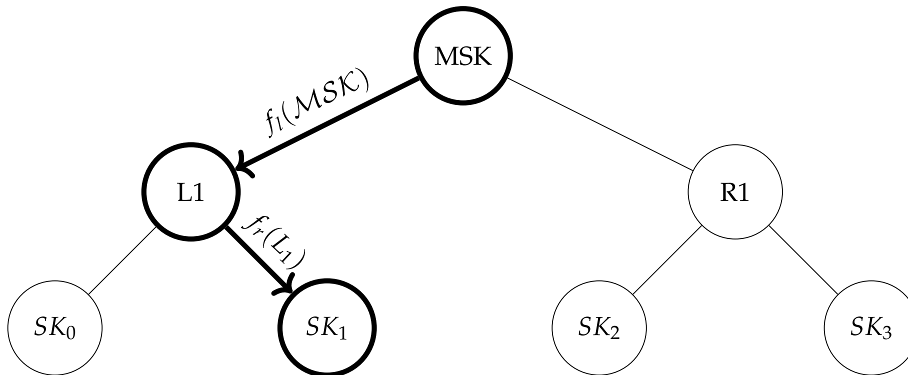


Figure 5.6: Example for key component retrieval for a tag τ using PBT-based PE with $n = 4$.

Puncturing

Revoking decryption capabilities for all messages encrypted using a tag τ consists of the following steps, the puncturing algorithm has to perform. Most intuitively, we need to delete the secret-key component. To achieve this, we do not only need to remove the leaf corresponding to the key component, but also all nodes which could be used to derive the component. This also includes the root of the tree. To still be able to access the other

Algorithm 9 Encryption

```
1:  function Encrypt(MSK, τ, M)
2:      b1b2 ← f(τ)                        ▷ get bit string corresponding to τ
3:      if b1 == 0 then
4:          SK_b1 ← f_l(MSK)
5:      else                                ▷ b1 == 1
6:          SK_b1 ← f_r(MSK)
7:      end if
8:      if b2 == 0 then
9:          SK_τ ← f_l(SK_b1)
10:     else                                ▷ b2 == 1
11:         SK_τ ← f_r(SK_b1)
12:     end if
13:     C ← Enc(SK_τ, M)
14:     return C
15: end function
```

key component, not corresponding to τ, we need to store all children of the nodes we deleted in the first step.

The simple example in figure 5.7 shows how an execution of the puncturing algorithm on a tag τ affects the underlying tree structure. Assuming τ corresponds to the secret-key component SK_1. Therefore, by puncturing on τ we need to delete all nodes which could be used to derive SK_1 (deletion indicated by crossing out the node). In this case these are the nodes MSK, L_1, and SK_1. In addition, we now need to store the nodes SK_0 and R_1 (depicted in bold) to still be able to derive all other secret-key components. As a result, the storage space needed to derive the secret-key components grows after puncturing.

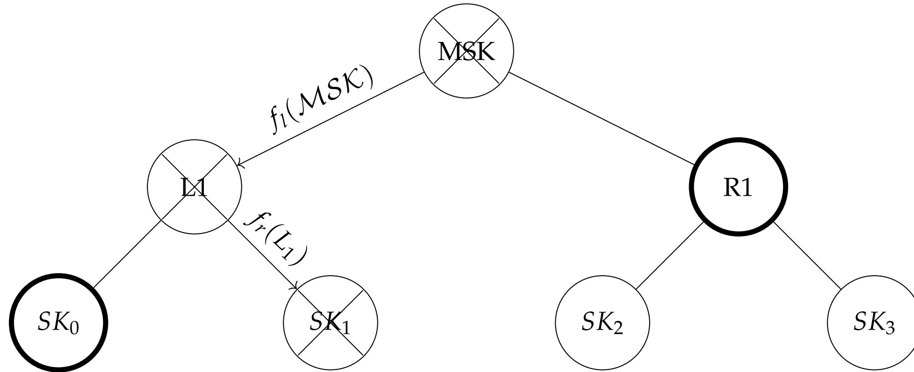


Figure 5.7: Example for puncturing on a tag τ using PBT-based PE with $n = 4$.

Decryption

Decryption of a message M encrypted under a tag τ follows a similar procedure as encryption. If the key is not punctured on τ we can retrieve the decryption key in the same way as we retrieved the encryption key for encryption described in Figure 5.6. If we punctured on the tag, the key retrieval will fail, since all nodes, using which we could have determined the key component, are deleted.

5.2.2 Theoretical analysis

Due to its hierarchical structure and the resulting need of storing intermediate nodes after puncture calls, a PBT-based PE scheme is able to achieve storage benefits compared to the naïve solution but is also sensitive to different puncturing orders. In the following, we describe how certain special orders affect the resulting secret-key storage of the approach.

- In order puncturing:

  Assuming the first tag corresponds to the leftmost (rightmost) and the last tag to the rightmost (leftmost) leaf of the tree, we end up storing at most $\log_2(n)$ nodes in a scheme supporting n tags. The same can be achieved if we puncture on the tags in reversed order (see A.2 for intuition). For the same reason, we can achieve this storage bound in a puncturing before next encryption ordering.

- Quasi-perfect ordering:

  Assuming a window size of w, where $w << n$ and n denotes the number of supported tags, i.e. the number of leaves the tree has. We now look at the sub-tree T' of our main tree, whereby T' is the smallest tree which includes all nodes in w as its leaves. To derive all nodes in T' we need a storage of at most $\mathcal{O}(w)$ since we could store each leaf node by itself [4]. Combining this observation with the upper bound achieved if we puncture in order, one can see that we increase the needed storage by $\mathcal{O}(w)$ at max in a quasi-perfect ordering. In conclusion, we can bound the needed storage space by $\mathcal{O}(log_2(n) + w)$.

- Worst-case:

  We achieve the worst-case storage requirement by puncturing on every second leaf in the tree. This results in needing to store the left $\frac{n}{2}$ leaves of the tree.

We can extract the following features achieved by a PBT-based PE scheme:

[4] Note that this will never happen in a real implementation. Therefore $\mathcal{O}(w)$ gives an upper bound for the needed storage.

- Hierarchical structure:

  Allows for storage reduction compared to the naïve solution.

- Achieves perfect correctness:

  Because the approach provides one unique key component for every supported tag.

- Benefits from special orderings:

  Special puncturing orders allow the approach to safe secret-key storage compared to the naïve solution.

Additionally, the approach has the following drawbacks:

- Hierarchical structure:

  Leads to varying secret-key storage (can increase and decrease). Results in longer time needed to access a key, therefore affecting the runtime of encryption, decryption and puncturing.

- No support for unbounded tag-space:

  Since we need to define the number of supported tags, n, in the initialisation of a PBT based PE scheme to define the function f which maps a tag to its corresponding key, an implementation with $n \to \infty$ is not possible.

A summary of all features and a comparison to the naïve solution is provided in figure 5.8. The PBT-based scheme not only achieves benefits from special puncturing orders which allows it to save on secret-key storage, but also achieve a better worst case storage requirement compared to the naïve scheme. Due to the hierarchical way in which we derive the key components associated to a tag, the efficiency of the encryption, decryption and puncturing algorithm is reduced.

Feature summary and comparison to naïve approach

Variable	Description
n	Number of supported tags
u	Number of unpunctured tags
t	Runtime of black-box SE.KeyGen() algorithm
e	Runtime of black-box SE.Enc() algorithm
d	Runtime of black-box SE.Dec() algorithm
w	Window size for quasi-perfect ordering

Feature	**PBT-based PE**	**Naïve SPE**
Secret-Key Storage		
Initial	$\mathcal{O}(1)$	$\Theta(n)$
During use	$\mathcal{O}(\frac{n}{2})$	$\Theta(u)$
Worst case	$\Theta(\frac{n}{2})$	$\Theta(n)$
Best case	$\Theta(\log_2(n))$	$\Theta(u)$
Computation time		
KeyGen()	$\mathcal{O}(t)$	$\mathcal{O}(nt)$
Encrypt()	$\mathcal{O}(e + \log_2(n))$	$\mathcal{O}(e)$
Puncture()	$\mathcal{O}(\log_2(n))$	$\mathcal{O}(1)$
Decrypt()	$\mathcal{O}(d + \log_2(n))$	$\mathcal{O}(d)$
Tag support		
Size of tag-space	Bounded	Bounded
Correctness		
Achieves	Perfect	Perfect
Special case benefits		
In order puncturing	Storage: $\mathcal{O}(\log_2(n))$	No benefits
Puncturing before next encryption	Storage: $\mathcal{O}(\log_2(n))$	No benefits
Quasi-perfect ordering	Storage: $\mathcal{O}(\log_2(n) + w)$	No benefits

Figure 5.8: Features achieved by a PBT-based PE scheme compared to the naïve SPE scheme. For each feature the better performing scheme is marked green and the worse performing is marked red.

Chapter 6

Blueprint

From the hitherto proposed constructions (presented in chapter 5) we observed that one can abstract a modal approach to building PE which makes the data structure used to build the key explicit. To be more precise, the data structure stores the various secret (public) key components which make up the secret (public) key of the scheme, whereby each component of the secret (public) key is a secret (public) key of an underlying encryption scheme which is used in a black-box fashion. By defining how we can access individual key components and remove them on puncture calls[1], we provide a framework based on a data structure which can be combined with different encryption schemes to arrive at a PE scheme. Such combinations can be used to adapt the used scheme even more to a specific application. For instance, using the naïve construction one can use a black-box encryption scheme which provides fast encryption and decryption, therefore speeding up the runtimes of the algorithms.

Part of the aim of this project was to construct a blueprint which could be used to filter out good candidate data structures for novel constructions of PE. However, during the analysis of different constructions, we discovered that very little is needed from the data structure itself. It suffices that keys can be stored and securely deleted. Instead, we found a requirement for a scheme to be able to achieve perfect correctness. Namely, the scheme needs to provide one unique key component for every supported tag (detailed explanation in section 6.1). This new discovery led to a more in-depth analysis of the key storage of PE schemes, and we found that regarding secret-key storage, the naïve solution is optimal for its class of perfectly correct, non-hierarchical PE schemes. As a result, we concluded that a new scheme can only achieve better storage requirements compared to the naïve solution by either not providing one unique key component for every supported tag, i.e. sacrificing perfect correctness, or not store all key components at the

[1]Where we assume the deletion to be secure.

same time, as in a PBT-based approach. We describe our findings in more detail in section 6.2.

6.1 Perfect correctness requirement

Achieving perfect correctness can be a crucial requirement for certain applications. We found a requirement for data-structure-based PE schemes which needs to be fulfilled for it to be able to achieve perfect correctness. Namely, it needs to provide at least one unique key component for every supported tag.

The intuition behind this requirement is as follows: To achieve perfect correctness a PE scheme must be able to decrypt any message encrypted under an unpunctured tag τ. If we assume τ has no unique key component, it is possible to puncture on τ by puncturing on all other tags τ_i which share a key component with τ.

This means that for any PE scheme achieving perfect correctness the required-secret key storage will be linear in the number of supported tags in the worst case.

Theorem 6.1 *A PE scheme using a tag-space $\mathcal{T}$ achieves perfect correctness if and only if it provides at least one unique key component for every tag $\tau \in \mathcal{T}$.*

Proof ($\Leftarrow$) Proof by contradiction:

Assume there exists a PE scheme using tag-space $\mathcal{T}$ and key-space $\mathcal{K}$ achieving perfect correctness but not providing at least one unique key component for every tag $\tau \in \mathcal{T}$.

This means there exists a tag $\tau' \in \mathcal{T}$ such that τ' has no unique key component. As a result, there has to exist a set $\hat{\mathcal{T}} \subseteq \mathcal{T} \setminus \{\tau'\}$ such that the set of key components $\mathcal{K}' \subseteq \mathcal{K}$ associated to τ' is a subset of the set of key components $\mathcal{K}_i$ associated to the tags $\tau_i \in \hat{\mathcal{T}}$, i.e. $\mathcal{K}' \subseteq \bigcup_{\tau_i \in \hat{\mathcal{T}}} \mathcal{K}_i$.

By puncturing on all tags $\tau_i \in \hat{\mathcal{T}}$ we delete all key components in the set $\bigcup_{\tau_i \in \hat{\mathcal{T}}} \mathcal{K}_i$ and since the set of key components associated to τ' is a subset of this set, i.e. $\mathcal{K}' \subseteq \bigcup_{\tau_i \in \hat{\mathcal{T}}} \mathcal{K}_i$, all key components associated to τ' get deleted, resulting in a secret-key punctured on τ' without ever puncturing on τ'. This means the scheme does not achieve perfect correctness.

We therefore arrive at a contradiction which proves our statement.

($\Rightarrow$) Proof by contradiction:

Assume there exists a PE scheme using tag-space $\mathcal{T}$ and key-space $\mathcal{K}$ providing at least one unique key component for every tag $\tau \in \mathcal{T}$ but not achieving perfect correctness.

From this follows that for all tags $\tau_i \in \mathcal{T}$ the set of key components $\mathcal{K}_i \subseteq \mathcal{K}$ associated to it contains at least one unique key component, i.e. $\forall \tau_i \in \mathcal{T}, |\mathcal{K}_i \setminus \bigcup_{\tau_j \in \mathcal{T} \setminus \{\tau_i\}} \mathcal{K}_j| \geq 1$.

Since we assumed that the scheme does not achieve perfect correctness, there exists a tag τ' and a set $\hat{\mathcal{T}} \subseteq \mathcal{T} \setminus \{\tau'\}$ such that by puncturing on all tags $\tau_i \in \hat{\mathcal{T}}$ we also puncture on τ'. This means the set of key components $\mathcal{K}'$ associated to τ' has to be a subset of the union of the set of key components associated to the tags $\tau_i \in \hat{\mathcal{T}}$, i.e. $\mathcal{K}' \subseteq \bigcup_{\tau_i \in \hat{\mathcal{T}}} \mathcal{K}_i$. And we therefore have $|\mathcal{K}' \setminus \bigcup_{\tau_j \in \hat{\mathcal{T}}} \mathcal{K}_j| = 0$.

We again arrived at a contradiction which proves our statement.

Since we have proven both directions, we can conclude that a PE scheme using a tag-space $\mathcal{T}$ achieves perfect correctness if and only if it provides at least one unique key component for every tag $\tau \in \mathcal{T}$. □

6.2 Storage reduction requirement

Efficiently reducing the required storage space of a PE scheme is one of the main goals for new constructions. This makes it even more interesting that we found a requirement necessary to achieve storage reduction compared to the naïve solution.

Theorem 6.2 *A PE scheme based on data structures with perfect correctness, supporting n tags can only achieve better key storage requirements compared to the naïve scheme for the same tag-space, if its underlying data structure is hierarchical.*

Remark:

- In the context of PE we consider a data structure to be hierarchical if it does not store all key components for all supported tags after initialization. For instance, we consider the Bloom filter underlying a BFE-based scheme to be non-hierarchical where in contrast the tree underlying a PBT-based approach is considered to be hierarchical[2].

Proof Proof by contradiction:

Assume we have a PE scheme which achieves perfect correctness and better storage requirements compared to the naïve approach, but its underlying data structure is not hierarchical.

From theorem 6.1 it follows that such a scheme needs to provide at least n unique key components. Since we assumed the scheme to be not hierarchical

[2] If the data structure stores some kind of seed values from which the key components get derived we consider it hierarchical if it does not store all seed values for all supported tags after initialization.

it needs to store n unique key components which results in a required key storage of at least $n * s$ bits, where s denotes the size of one key component, i.e. the storage S for the secret-key in this scheme is $S \geq n * s$.

The naïve approach supporting n tags stores exactly one key component for each tag, therefore requiring $N = n * s$ bits to store the secret-key.

We can see that we arrive at a contradiction since we assumed $S < N$ and derived $S \geq n * s = N$.

Therefore, we conclude that our initial assumption is false, what proves our statement. □

Remark:

- Dynamic PE schemes (introduced in chapter 7) can also reduce the required storage space in certain scenarios, since they allow to extend the supported tag-space if needed. Note however that at any point in time a dynamic PE scheme, currently supporting n_i tags, can only achieve better storage requirements compared to the naïve approach supporting n_i tags, if its underlying data structure is hierarchical as is proven by this theorem.

Corollary 6.3 *A PE scheme based on data structures with perfect correctness, supporting n tags can only achieve better key storage requirements compared to the naïve scheme by trading algorithm efficiency for it, i.e. at least one of the algorithms (encryption decryption or puncturing) will always be slower compared to the one of the naïve solution for some tags.*

Proof Proof by contradiction:

We have a PE scheme which achieves perfect correctness, supports n tags and achieves better key storage requirements then the naïve solution. Assume the algorithms of this scheme, i.e. encryption, decryption and puncturing, are always at least as efficient as the algorithms from the naïve scheme.

Our scheme needs to be based on a hierarchical data structure (as shown in theorem 6.2) and therefore there exist at least two tags τ_1 and τ_2 in the supported tag-space $\mathcal{T}$ whose associated key components are not directly stored inside the data structure but get derived from a shared 'seed value'[3]. For instance, any two tags and the root-key in a PBT-based scheme. Let's consider the following three cases:

[3]If only one such tag τ exists, it would mean that we store a 'seed value' from which we can derive the key associated to τ with the property that the seed value requires less storage than the computed key component. This would mean any message encrypted using τ is less secure, since an adversary would only need to brute-force the smaller seed value instead of the longer key component. For the same reason, the 'seed value' must be shared by the two tags.

- Encryption using τ_1 or τ_2:

  To encrypt a message under either τ_1 or τ_2 forces the derivation of the associated key component. This requires the application of some function f on the stored 'seed value' for at least one time. Assuming the computation of f for a particular value needs $z > 0$ time we need $\Theta(z + e)$ time to encrypt a message under the tag, whereby e denotes the time needed to encrypt a message using the black-box encryption scheme.

- Decryption using τ_1 or τ_2:

  Similar to encryption, decryption requires the derivation of the associated key component. Assuming the black-box decryption algorithm has a runtime of d, we end up needing $\Theta(z + d)$ time to decrypt the message.

- Puncturing on τ_1 or τ_2:

  Puncturing on either one of the two tags requires the deletion of the stored 'seed value'. Since we assumed our scheme to achieve perfect correctness, we need to derive the key component associated to the other tag prior to the deletion of the seed value. Therefore, puncturing one of these two tags requires $\Theta(z)$ time.

We assumed encryption, decryption and puncturing to be at least as efficient as the algorithms of the naïve solution, i.e. run in $\Theta(e)$, $\Theta(d)$ and $\mathcal{O}(1)$ respectively. We have shown that for some tags the algorithms need time $\Theta(z + e)$, $\Theta(z + d)$, and $\Theta(z)$ and therefore arrive at a contradiction.

This means our initial assumption is wrong, what proves our statement. □

Corollary 6.4 *Any non-hierarchical data-structure-based PE scheme achieving perfect correctness needs to store at least u unique key components during its use, if u denotes the current number of unpunctured tags.*

Proof This statement follows directly from theorem 6.1 and theorem 6.2. □

Corollary 6.5 *Any non-hierarchical data-structure-based PE scheme achieving perfect correctness cannot beat the naïve solution regarding key storage and algorithm efficiency, i.e. the naïve scheme is optimal for this class of PE schemes.*

Proof We prove the statement directly:

The optimality of the secret-key storage follows directly from theorem 6.2 and corollary 6.4.

The runtime of both the encryption and decryption algorithm depends on the black-box encryption scheme used. Assuming encrypting and decrypting take time e and d respectively, a successful encryption (decryption) needs

at least e (d) time. We therefore arrive at a lower bound of $\Omega(e)$ and $\Omega(d)$ for the runtime of the encryption and decryption algorithm, respectively. An encryption (decryption) call for a punctured tag returns in constant time in the naïve scheme. In conclusion, we can see that the runtime of the encryption and decryption algorithm in the naïve approach are optimal.

Since puncturing is possible in constant time in the naïve solution, it also achieves an optimal runtime for the puncturing algorithm. □

6.3 Consequence for new constructions

Our discoveries showed that new constructions need to provide at least one unique key component for every supported tag to achieve perfect correctness. Hence, to be able to achieve better storage bounds for the secret-key than linear in the number of supported tags a novel construction needs to be hierarchical, and therefore sacrifice algorithm efficiency, or give up on perfect correctness.

The search for novel construction therefore becomes the quest of finding a good trade-off between algorithm efficiency, required secret-key storage space and correctness. How good such a trade-off is will depend a lot on the application. For instance, we may not care about a small correctness error if we instead can achieve highly efficient algorithms and small storage requirements.

We now explore the secret-key storage, algorithm efficiency trade-off by considering ways in which a PE scheme can support unlimited tags (messages) while still having bounded key storage.

Chapter 7

Dynamic Puncturable Encryption

As shown in corollary 6.3 a PE scheme can only achieve better key storage requirements compared to the naïve solution if it trades some algorithm efficiency for it. This in addition to the optimality of the naïve solution for its class of PE schemes (as shown in corollary 6.5) means the search for new data structures suitable for PE construction becomes the quest of finding the optimal trade-off between storage reduction, algorithm efficiency and correctness.

To help us find data structures which achieve good trade-offs, we introduce a new class of PE schemes which we call Dynamic Puncturabel Encryption (DPE) schemes. The main idea behind DPE schemes is to initially only support a small part of the entire tag-space and extend it, and therefore the secret-key, once we used up all tags and require new ones. This allows us to save storage space while still providing rather efficient algorithms.

7.1 Syntax, correctness and security

To introduce this functionality, we extend the definition of a PE scheme with a new algorithm called KeyExt, the key extension algorithm. We arrive at the following definition:

Definition 7.1 *A tag-based* Dynamic Symmetric Puncturable Encryption (DSPE) *scheme with tag-space $\mathcal{T}$, message-space $\mathcal{M}$, key-space $\mathcal{K}$, and ciphertext-space $\mathcal{C}$, consists of a tuple of five algorithms (*DSPE.KeyGen, DSPE.Enc, DSPE.Dec, DSPE.Punct, DSPE.KeyExt*) with the following syntax:*

- $SK_0 \leftarrow_\$$ DSPE.KeyGen$(\lambda, n, params)$*: Given a security parameter λ and an initial number of tags n, where $n \in \mathbb{N} \cup \{\infty\}$ and ´∞´ corresponds to an unbounded number of tags as well as an array params holding additional parameters if needed, outputs an initial secret-key $SK_0 \in \mathcal{K}$.*

- $C \leftarrow_\$ \text{DSPE.Enc}(SK_i, M, \tau)$: *On input a secret-key* $SK_i \in \mathcal{K}$*, a message* $M \in \mathcal{M}$ *and a tag* $\tau \in \mathcal{T}$*, outputs a ciphertext* $C \in \mathcal{C} \cup \{\perp\}$*, where* $\perp$ *indicates that the encryption failed.*
- $M \leftarrow \text{DSPE.Dec}(SK_i, C, \tau)$: *Given a secret-key* $SK_i \in \mathcal{K}$*, a ciphertext* $C \in \mathcal{C}$*, and a tag* $\tau \in \mathcal{T}$*, corresponding to C, outputs* $M \in \{\mathcal{M}\} \cup \{\perp\}$*, where* $'\perp'$ *indicates that the decryption failed.*
- $SK_i \leftarrow \text{DSPE.Punct}(SK_{i-1}, \tau)$: *Takes as input a secret-key* $SK_{i-1} \in \mathcal{K}$ *and a tag* $\tau \in \mathcal{T}$ *and outputs a (new) secret-key* $SK_i \in \mathcal{K}$*.*
- $SK_i \leftarrow \text{DSPE.KeyExt}(SK_i, params)$: *On input a secret-key* SK_i *and an array of potentially needed additional arguments, outputs a extended version of the secret-key* SK_i*.*

Note that not all DSPE schemes need an explicit key extension algorithm. Schemes based on hierarchical data structures can implement the key extension implicitly in the encryption, decryption and puncturing algorithms, therefore not needing a key extension algorithm.

Using our definition given above, we define perfect correctness of a DSPE scheme as follows:

Definition 7.2 *A tag-based DSPE scheme given by (*DSPE.KeyGen, DSPE.Enc, DSPE.Dec, DSPE.Punct, DSPE.KeyExt*) with tag-space* $\mathcal{T}$*, message-space* $\mathcal{M}$*, key-space* $\mathcal{K}$*, and ciphertext-space* $\mathcal{C}$ *achieves perfect correctness, if for all* λ*, for all* $n \in \mathbb{N} \cup \{\infty\}$*, and for all* $\tau \in \mathcal{T}$*,* $SK_0 \leftarrow_\$ \text{DSPE.KeyGen}(\lambda, n, params)$ *and* $C \leftarrow_\$ \text{DSPE.Enc}(SK_i, M, \tau)$*, where* $M \in \mathcal{M}$*,* $C \in \mathcal{C}$*, and* $i \in \{0, \ldots, n\}$ *we have that:*

- $\exists$ *a finite sequence of key extensions calls,* $SK_0 \leftarrow \text{DSPE.KeyExt}(SK_0, params)$*, such that* $Pr[M == \text{DSPE.Dec}(SK_0, \text{DSPE.Enc}(SK_0, M, \tau), \tau)] = 1$.
- $\exists$ *a finite sequence of key extensions calls,* $SK_i \leftarrow \text{DSPE.KeyExt}(SK_i, params)$*, such that* $\forall$ *sequences of* DSPE.Punc *calls* $SK_i \leftarrow \text{DSPE.Punct}(SK_{i-1}, \tau_i)$*, for* $i = 1$ *to* n*, we have that* $Pr[M == \text{DSPE.Dec}(SK_i, C, \tau)] = 1$*, if* $\tau \notin \bigcup_i \tau_i$*.*

Remark:

- Note that the index i of the secret-key SK_i denotes the number of puncturings performed. We do not increment i on an extension call.

Additionally, we define a relaxed correctness notion which we call *relaxed correctness* that allows for a negligible correctness error as follows:

Definition 7.3 *A tag-based DSPE scheme given by (*DSPE.KeyGen, DSPE.Enc, DSPE.Dec, DSPE.Punct, DSPE.KeyExt*) with tag-space* $\mathcal{T}$*, message-space* $\mathcal{M}$*, key-space* $\mathcal{K}$*, and ciphertext-space* $\mathcal{C}$ *achieves relaxed correctness, if for all* λ*, for*

```
Game G^{fs-ind-cpa}_{DSPE}(λ, n)

1  b ←$ {0,1}
2  PT ← ∅; CS ← ∅; ChalT ← ∅
3  corrupted ← False; i ← 0
4  SK_0 ←$ DSPE.KeyGen(λ, n, params)
5  b* ←$ A^{Punct(·),Extension(·)}_{Corr(),Challenge(·,·,·)}()
6  Return b == b*

Challenge(M_0, M_1, τ):

7  if corrupted and (τ ∉ CS): Return ⊥
8  ChalT ← ChalT ∪ {τ}
9  C ←$ DSPE.Enc(SK_i, M_b, τ)
10 Return C

Punct(τ):

11 i ← i + 1
12 SK_i ← DSPE.Punct(SK_{i−1}, τ)
13 PT ← PT ∪ {τ}
14 Return

Corr():

15 if ChalT ⊈ PT: Return ⊥
16 corrupted ← True
17 CS ← PT
18 Return SK_i //current key

Extension(params):

19 SK_i ←DSPE.KeyExt(SK_i, params)
20 Return SK_i
```

Figure 7.1: Game formalizing fs-ind-cpa security of a DSPE scheme.

---

all $n \in \mathbb{N} \cup \{\infty\}$, and for all $\tau \in \mathcal{T}$, $SK_0 \leftarrow_{\$} \text{DSPE.KeyGen}(\lambda, n, params)$, and $C \leftarrow_{\$} \text{DSPE.Enc}(SK_i, M, \tau)$, where $M \in \mathcal{M}$, $C \in \mathcal{C}$, and $i \in \{0, \ldots, n\}$, there exists a negligible function $negl : \mathbb{N} \rightarrow \mathbb{R}$ such that:

- *$\exists$ a finite sequence of key extensions calls, $SK_0 \leftarrow \text{DSPE.KeyExt}(SK_0, params)$, such that $Pr[M == \text{DSPE.Dec}(SK_0, \text{DSPE.Enc}(SK_0, M, \tau), \tau)] = 1$.*
- *$\exists$ a finite sequence of key extensions calls, $SK_i \leftarrow \text{DSPE.KeyExt}(SK_i, params)$, such that for all sequences of DSPE.Punct calls $SK_i \leftarrow \text{DSPE.Punct}(SK_{i-1}, \tau_i)$, for $i = 1$ to n, we have that $Pr[M == \text{DSPE.Dec}(SK_i, C, \tau)] = 1 - negl(\lambda)$, if $\tau \notin \bigcup_i \tau_i$, i.e. we allow for a negligible correctness error introduced by puncturing calls.*

To define fs-ind-cpa the security of a DSPE scheme, we extend the game from figure 3.1 to also allow for key extension using the key extension algorithm. We present the new game in figure 7.1. Again using definition 2.7 we define the advantage an adversary has while playing this game as follows:

$$\mathbf{Adv}^{\text{fs-ind-cpa}}_{\text{DSPE}}(\mathcal{A}, \lambda, n) = 2 \left| Pr \left[True \Longleftarrow \mathbf{G}^{\text{fs-ind-cpa}}_{\text{DSPE}}(\lambda, n) \right] - \frac{1}{2} \right|.$$

Using it we arrive at the following definition for a fs-ind-cpa secure DSPE scheme:

Game $\mathbf{G}^{\text{fs-ind-cca}}_{\text{DSPE}}(\lambda, n)$

1 $b \leftarrow\$ \{0,1\}$; $ChalT \leftarrow \emptyset$
2 $\mathcal{PT} \leftarrow \emptyset$; $\mathcal{CS} \leftarrow \emptyset$; $\mathcal{CT} \leftarrow \emptyset$
3 $corrupted \leftarrow False$; $i \leftarrow 0$
4 $SK_0 \leftarrow\$ \text{DSPE.KeyGen}(\lambda, n, params)$
5 $b^* \leftarrow\$ \mathcal{A}^{\text{Punct}(\cdot),\text{Corr}(),\text{Extension}(\cdot)}_{\text{Dec}(\cdot,\cdot),\text{Challenge}(\cdot,\cdot,\cdot)}()$
6 Return $b == b^*$

$\text{Dec}(C, \tau)$:

7 if $C \in \mathcal{CT}$: Return $\perp$
8 $M \leftarrow \text{DSPE.Dec}(SK_i, C, \tau)$
9 Return M

$\text{Challenge}(M_0, M_1, \tau)$:

10 if *corrupted* and $(\tau \notin \mathcal{CS})$: Return $\perp$
11 $ChalT \leftarrow ChalT \cup \{\tau\}$
12 $C \leftarrow\$ \text{DSPE.Enc}(SK_i, M_b, \tau)$
13 $\mathcal{CT} \leftarrow \mathcal{CT} \cup \{C\}$
14 Return C

$\text{Punct}(\tau)$:

15 $i \leftarrow i + 1$
16 $SK_i \leftarrow \text{DSPE.Punct}(SK_{i-1}, \tau)$
17 $\mathcal{PT} \leftarrow \mathcal{PT} \cup \{\tau\}$
18 Return

$\text{Corr}()$:

19 if $ChalT \not\subseteq \mathcal{PT}$: Return: $\perp$
20 $corrupted \leftarrow True$
21 $\mathcal{CS} \leftarrow \mathcal{PT}$
22 Return SK_i //current key

$\text{Extension}(params)$:

23 $SK_i \leftarrow \text{DSPE.KeyExt}(SK_i, params)$
24 Return SK_i

Figure 7.2: Game formalizing fs-ind-cca security of a DSPE scheme.

Definition 7.4 *We consider a* Dynamic Symmetric Puncturable Encryption *scheme* fs-ind-cpa *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{fs-ind-cpa}}_{\text{DSPE}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

By extending the game from figure 3.2 to allow for key extension, we arrive at a game formalizing fs-ind-cca security for a DSPE scheme. The new game is depicted in figure 7.2. We arrive at the follwing notions for the advantage of an adversary has while playing the game.

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{DSPE}}(\mathcal{A}, \lambda, n) = 2\left| Pr\left[True \Longleftarrow \mathbf{G}^{\text{fs-ind-cca}}_{\text{DSPE}}(\lambda, n)\right] - \frac{1}{2}\right|.$$

From it follows the fs-ind-cca security of a DSPE scheme.

Definition 7.5 *We say a DSPE scheme is* fs-ind-cca *secure, if there exists a negligible function negl* : $\mathbb{N} \rightarrow \mathbb{R}$ *such that*

$$\mathbf{Adv}^{\text{fs-ind-cca}}_{\text{DSPE}}(\mathcal{A}, \lambda, n) \leq negl(\lambda).$$

7.2 Problems of DPE

One of the main challenges in constructing a DPE scheme is the question of how to maintain the same state on both sides of a communication. We illustrate the problem on a small example.

Alice and Bob communicate using a DSPE scheme. If Alice runs out of tags, she uses the key extension algorithm of the scheme to extend the tag-space. Assume she now sends two messages M_1 and M_2 under different, new tags τ_1 and τ_2 after she extended the tag-space. The message corresponding to τ_1 gets lost. Bob now receives a cyphertext associated to τ_2. How does Bob now know in which way he needs to extend the tag-space to derive the correct key component for τ_2? I.e. how does Bob know that the new key component he derived is the one Alice used for τ_2 and not the one for τ_1?

A similar problem arises when messages arrive out of order, and it is not easy to solve if one assumes we use arbitrary strings as tags.

Our solution to the problem is to only use DPE schemes whose tag-spaces consist of sequence numbers. This way, in a DSPE scheme, we can use deterministic algorithms on both sides of the communication which then assign a newly derived key component to its tag[1]. Additionally, if messages get lost or arrive out of order we can deduce how many intermediate keys we need to derive. For example, if the current tag-space supports sequence numbers $\{0, \ldots, 100\}$ and we receive a message associated to the tag 105, we can derive the keys associated to the tags $101, 102, 103, 104$ and 105 in the correct order. Note that by extending the secret-key with many new components at once could mitigate the problem of messages arriving out of order, but i does not solve the problem of an adversary blocking all messages associated to tags from a first extension and only letting through messages associated to tag from a second extension.

7.2.1 Unsuitability for PkPE

Recall that the main idea behind dynamic PE schemes is to extend the secret-key once we need to support more tags. By making the key extension algorithm deterministic, such an extension can be performed by both parties in a symmetric encryption scenario. This means that once the encrypting party detects that it has run out of tags, it can extend the secret-key by itself, encrypts the message using the newly derived secret-key and sends it to the decrypting party which then detects that it needs to extend the secret-key as well. Since both parties use the same deterministic extension algorithm, they derive the same key components.

[1] One could think of a random number generator initialized with the same seed on both sides of the communication. The first random number will then be assigned to the first tag and so on.

While this works for symmetric encryption, it cannot be applied to public-key encryption. An encrypting party, detecting that the public-key needs to be extended, cannot derive the new public-key by itself. Instead, it has to message the decrypting party that an extension is required. It can then extend the key and update its secret-key and the public-key accordingly. This introduces problems, such as large overheads for encryption once key extension is required. An adversary could repeatedly send extension requests to keep the decrypting party busy computing new key components, the message asking for key extension could be lost, corrupted or possibly blocked by an adversary. In addition to all of that, it is difficult for the decrypting party to monitor the public-key and detecting itself that extension is required. Since the public-key remains unchanged after a puncturing call, the only way for the decrypting party to check if key extension is required, is to rely on the tags received with encrypted messages. Assuming we use sequence numbers as tags, the decrypting party could store the highest sequence number received and depending on it, extend the public and secret-key before we would run out. This then forces us with the problem of the timing of such an extension, meaning how many tags can be used before we extend the key. Let tl denote the number of unused tags when we call the key extension. If one, or multiple, encrypting parties try to send $m > tl$ messages with different unused tags in a short amount of time, $m - tl$ of them can not be encrypted because there does not exist a key component for their corresponding tags.

All this points led us to the decision that DPE is not well suited for public-key cryptography, and we therefore decided to restrict our analysis to DSPE schemes only.

Chapter 8

New Construction Ideas

Using our new insights on requirements presented in chapter 6, we present new ideas on how to construct PE schemes. For each proposal, we describe the high-level idea, list potential benefits over the naïve solution, and point out possible drawbacks of each approach. To conclude the chapter, we present a list of (classes of) data structures which we classified as unsuited for PE schemes and briefly describe the reasons for it this decision.

8.1 Puncturable Encryption based on Ratcheting

This scheme is based on the *ratcheting* technique used to achieve forward secrecy in the Signal messaging protocol. We base our observation on the formal security analysis done by Cohn-Gordon et al. [5]. The scheme falls in the class of perfectly correct, hierarchical PE schemes.

To achieve forward secrecy, the high level idea behind the ratcheting mechanism is to use one key component for each tag. Although this sounds like the idea used in the naïve solution, the main difference lies in the way we get the key components, i.e. in context of PE a ratcheting-based PE scheme is hierarchical. The ratcheting mechanism uses a so-called RATCHET KEY and two one-way functions f_{mk} and f_{rk}. The key generation algorithm generates an initial ratchet key RK_0. From this, a message key can be derived using f_{mk}, the message key function. On input a ratchet key RK_i, f_{mk} outputs a message key MK_i which is used as the secret-key for encrypting a message.

The message keys are single-tag in the sense that they are used to encrypt a message only under a specific tag, namely the message key MK_i is used to encrypt the messages associated to the tag $\tau_i = i$ (recall that we assume to be using sequence numbers as tags as described in section 2.2.1). To allow encryption under more than a single tag, there is also a ratchet key function f_{rk}, which on input a ratchet key RK_i outputs the next ratchet key RK_{i+1}.

To puncture on a tag $\tau = i$ the ratchet key RK_{i+1} needs to be derived and the old ratchet key gets securely deleted. This forces the derivation of all message keys MK_j with $j < i$ to still achieve perfect correctness and guarantees that MK_i cannot be derived any more. If the currently stored ratchet key is RK_l with $l > i$ no new ratchet key needs to be derived and the stored message key MK_i gets securely deleted.

Due to its hierarchical nature, it is possible to achieve key storage reduction compared to the naïve solution using this approach. A PE scheme based on ratcheting could, especially for certain special puncturing orders, achieve far better storage requirements while still providing reasonably fast algorithms.

Figure 8.1 shows how the underlying data structure for the ratchet mechanism could look like if we used four different tags $\tau_0, \ldots, \tau_3$ and punctured on τ_0 and τ_3. One can see that after puncturing we are left with storing MK_1, MK_2, and the current ratchet key RK_4.

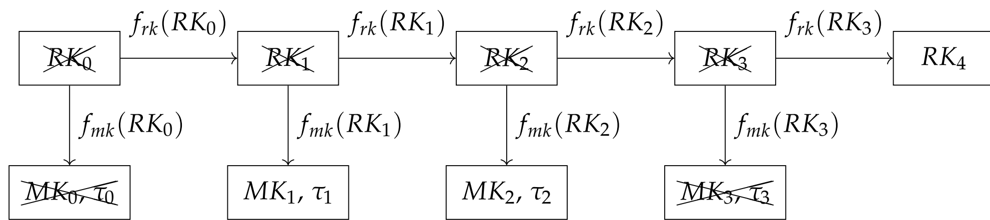


Figure 8.1: Illustration of the underlying data structure of a PE scheme based on ratcheting in a state where we punctured on tag τ_0 and τ_3. Shown are all parts which have been part of the scheme up until this state and how they got derived (using f_{mk} and f_{rk}). All crossed-out boxes indicate that the corresponding elements got deleted and are not part of the data structure any more due to puncturing calls.

8.2 Chained Perfect Binary Tree Puncturable Encryption

By using a ratchet key we can derive root keys for PBTs which would allow us to extend the supported tag-space of a PBT-based scheme dynamically, therefore this scheme falls into the class of dynamic PE schemes. We would start with one single PBT, which then functions as a standard PBT-based PE scheme, and add a second PBT once we run out of key components. We would therefore end up with a chain of trees which make up the secret-key of this new scheme.

Such a scheme needs two one-way functions f_{mk} and f_{rk}. The key generation algorithm outputs an initial ratchet key RK_0. From this, a ´root-key´ $Root_0$

can be derived using f_{mk}, the root-key function. On input a ratchet key RK_i, f_{mk} outputs a root-key $Root_i$, which is used as the root of a PBT T_i as in the original PBT-based scheme.

To allow more than one PBT to be used, there is also the ratchet key function f_{rk}, which on input a ratchet key RK_i outputs a new ratchet key RK_{i+1}. After each derivation of a new root $Root_i$ a new ratchet key RK_{i+1} is derived and the old version gets securely deleted.

By knowing how many tags each PBT supports (which is able to vary from tree to tree) a tag $\tau_i = i$ can get assigned to the correct tree which then is used to derive the corresponding key component as in a PBT-based approach (recall we assumme to use sequence numbers as tags as described in section 2.2.1).

Puncturing on a tag functions as in the PBT-based approach after finding its corresponding tree T_j.

Compared to a naïve solution, such a chained PBT approach could allow us to safe even more storage space, since we do not need to provide a tree for the entire tag-space from the start. Additionally, since the individual trees are smaller than one tree supporting the entire tag-space the involved algorithms can run faster compared to the original approach. Additionally, we would still be able to get storage benefits from special puncturing orders as we do using the standard approach.

The illustration in Figure 8.2 shows how such a chained PBT approach could look like. For both τ_0 and τ_1 the corresponding key is stored in the first tree T_0. The one corresponding to τ_l is stored in T_2 and the one corresponding to τ_m is in T_4. If we would run out of key components, we would use RK_4 to derive a new tree T_4.

8.3 Dynamic Perfect Binary Tree Puncturable Encryption

This approach functions as the PBT-based approach does, except we store a key component in every node of the tree. To do this we use a tree consisting of ratchet keys which we could derive in the same way as we derive the keys in a standard PBT-based approach, i.e. we use two one-way functions f_l and f_r to derive the left and right child inside the tree. To access a key associated to a tag we use these functions to derive the correct node inside the tree and then use an additional function f_{mk} to drive the actual message key from the ratchet key, i.e. given a ratchet key RK_i, f_{mk} outputs the message key MK_i which is used as the secret-key for encrypting a message.

Encryption, decryption and puncturing therefore function in the same way as they do in the PBT-based approach with the main difference that the

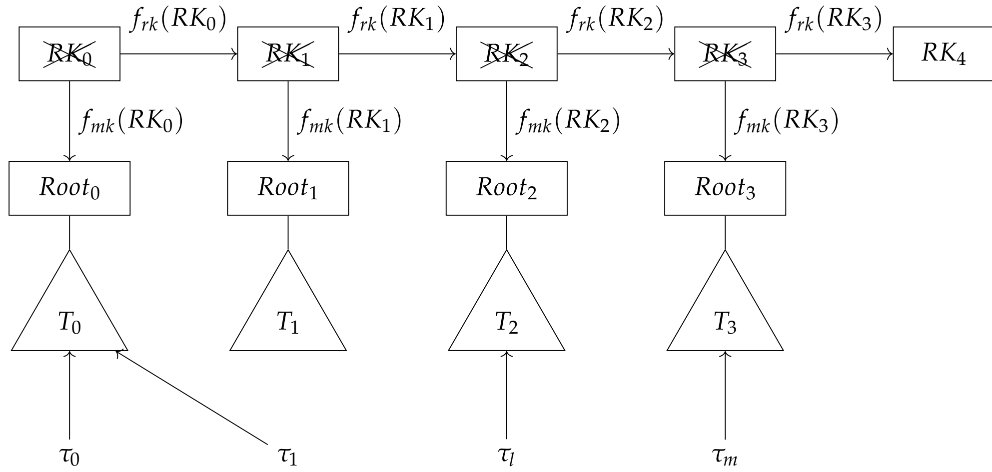


Figure 8.2: Illustration of the underlying data structures for a chained PBT-based approach currently supporting 4 trees. Shown are all parts of the scheme up until this state and how they got derived. Crossed out boxes indicate elements which have been used in the scheme but got deleted.

key component could not only be stored in a leaf node. Puncturing, compared to its functionality in the PBT-based approach, additionally requires the derivation of the children of deleted leaf nodes.

Such an approach potentially supports big tag-spaces while still providing rather efficient algorithms and a potentially smaller storage-space compared to the naïve solution and is an example of a dynamic PE scheme.

In figure 8.3 we illustrate how such a dynamic PBT-based scheme could look like.

Further analysis of the idea showed that such a scheme performs rather poorly regarding the required storage space. On a puncturing call, we need to calculate and store the children of the punctured node, otherwise we would not be able to derive all keys or extend the tag-support. This results in an increase of one key after each puncturing call, i.e. the required storage grows linearly in the number of puncturings performed.

Figure 8.4 illustrates how puncturing on a tag τ_5, associated to a current leaf node, effects the underlying tree structure of such a scheme. We compute the children of the deleted node using f_l and f_r to allow further expansion of the tree after puncturing. We mark all stored nodes in bold.

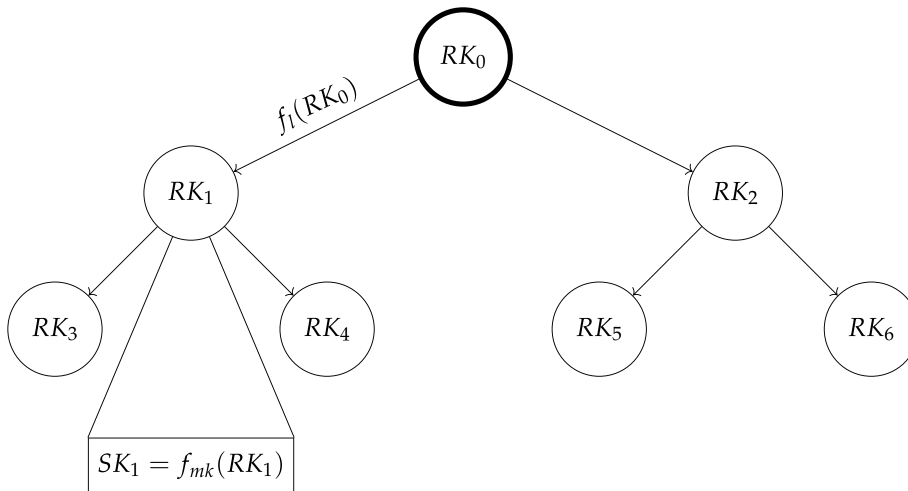


Figure 8.3: Illustartion of a dynamic PBT-based scheme. Shown, the message key (SK_1) derived from the 'root ratchet key' RK_0.

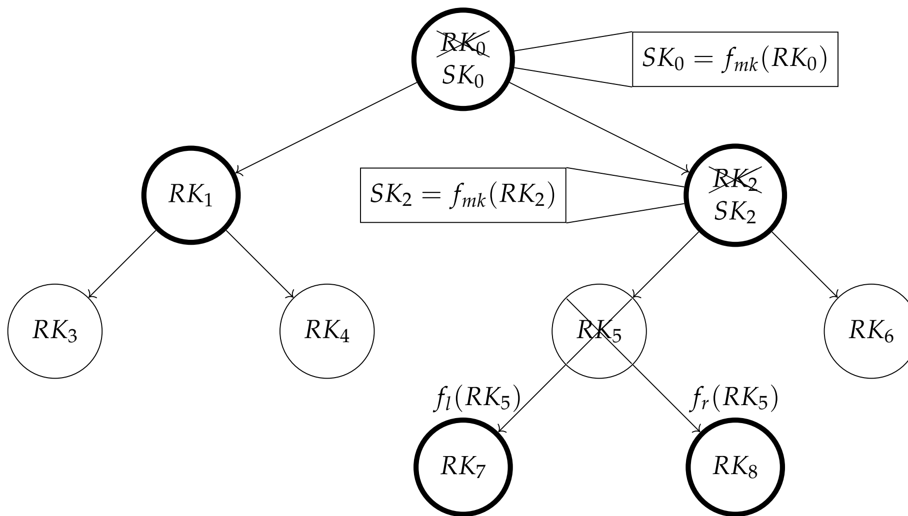


Figure 8.4: Example for puncturing on a ´leaf´ tag τ_5 in a dynamic PBT-based scheme.

8.4 BFE-based Puncturbale Encryption with Perfect Correctness

Recall the construction of PE based on BFE in section 5.1. Due to the probabilistic nature of a Bloom filter, we saw that the construction suffered from a lack of perfect correctness. Here we present an idea of how the BFE-based construction can be extended to achieve perfect correctness.

The approach functions in the same way the original BFE-based approach does, with the main difference being that we detect collisions prior to encryption. This means we check if there already exist a set of tags which use the same key components as our new tag τ, i.e. puncturing on them would result in puncturing on τ. If this is the case, we add a special key component for τ. We can therefore guarantee that we have at least one unique key component for every supported tag and can therefore achieve perfect correctness. The new scheme is therefore an example of a non-hierarchical, perfectly correct PE scheme.

Although this approach achieves perfect correctness, further analysis showed that it uses more storage space as the naïve solution and additionally all involved algorithms are slower compared to the ones from the naïve approach. We therefore conclude that such a scheme is not useful in practice.

8.5 Chained BFE-based Puncturable Encryption

A second idea to achieve perfect correctness with a BFE-based approach would be to chain multiple Bloom filters together. We would again check for a collision on the tags prior to encryption and if we encounter one we would use the next Bloom filter in the chain and check again. This step gets repeated until we encounter no collision and can encrypt our message. The scheme would therefore be part of the class of dynamic PE schemes.

As for the first extension, this scheme does not only have slower algorithms than the naïve solution but also requires more storage space, and we therefore conclude that it is not suitable for practical use.

8.6 Bloom Filter Cascade Puncturable Encryption

A PE scheme making use of Bloom filter cascades could also achieve perfect correctness by guaranteeing that for each tag, there exists one level in the cascade in which we do not encounter a collision on all computed hash values. The resulting scheme would be a non-hierarchical, perfectly correct PE scheme.

As for the other two BFE-extension also, this approach suffers from slow algorithms and needs more storage compared to the naïve scheme.

8.7 Puncturable Encryption through Re-Encryption

As any other encryption scheme, puncturable encryption suffers from the problem that any message could potentially be decrypted using brute force. In this approach we try to use this fact to base our puncturing algorithm

on, i.e. we relay on the amount of possible combinations to re-encrypt a message to puncture on a tag. The main idea is as follows: We provide a fixed number of keys k, $SK_0, \ldots, SK_{k-1}$ and a ´reencryption number´ r. To encrypt a message under a tag we associate r numbers, $n_0, \ldots, n_{r-1}$, between 0 and $k-1$ to the tag while keeping an order for these numbers. We then encrypt the message first under SK_{n_0}, the calculated ciphertext under SK_{n_1} and so on until we encrypted iteratively using the keys SK_{n_0} to $SK_{n_{r-1}}$. To then puncture on a tag, we remove the mapping from the tag to its sequence of numbers as well as ´remove the way to derive the sequence´.

We now consider different key lengths used for encryption today and calculate the number of needed re-encryptions to achieve a similar number of possible combinations (see figure 8.5).

Key Length	Number of used Keys						
	100	200	500	1000	2000	5000	10000
128	20	17	15	13	12	11	10
256	39	34	29	26	24	21	20
512	78	67	58	52	47	42	39
1024	155	134	115	103	94	84	78
2048	309	268	229	206	187	167	155
4096	617	536	457	412	374	334	309

Figure 8.5: Number of re-encryptions needed to achieve roughly the same number of possible combinations, as there are keys of length *Key Length*, using a multiple encryption approach with a defined number of used keys.

To achieve such a PE scheme, we need to find a way to assign a tag to a sequence of numbers in a pseudo-random way, such that the mechanism cannot be reversed. One idea to achieve this is to use an initial base vector $\mathcal{BV}$ holding r random numbers between 0 and $k-1$. We then use a ´matrix like´ structure which will hold vectors corresponding to tags. To compute the j-th key used to encrypt a message, we multiply the j-th entry of $\mathcal{BV}$ with the j-th element of the column vector corresponding to the tag, stored in the matrix, and take the result modulo k. The final number than corresponds to the index of the j-th key used in the encryption sequence. An example of this is shown in figure 8.6.

Such a PE scheme, which would fall into the class of dynamic PE schemes, could potentially achieve better storage requirements as the naïve solution, since its set of key components remains constant. Further analysis showed that this is not possible since now the arrays associated to the tags become the 'new key components' for this approach, i.e. if we for example only

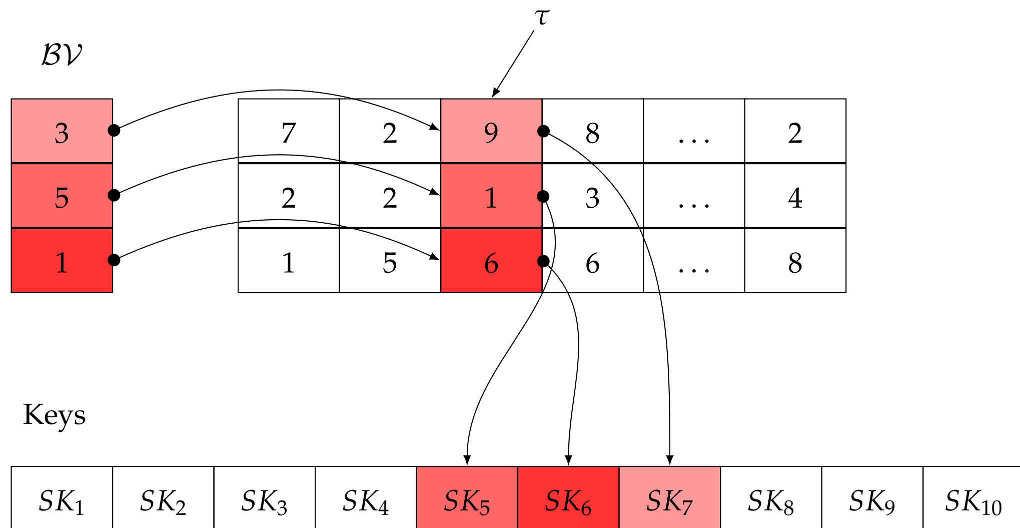


Figure 8.6: Example of encryption using $r = 3$ re-encryption rounds and $k = 10$ keys.

require 100 bits to store such an array we can only achieve the same security as with a 100 bit single key as the number of combinations needed to be tested to brute-force the scheme would be the same. This in addition to the fact that both encrypting and decrypting would require more time in such a scheme, led to the conclusion that a PE scheme based on re-encryption is not suitable for use in practice.

8.8 Lambda-Structures

By chaining together multiple ratcheting-based PE schemes, we would end up with a ´lambda-like´ structure and therefore another perfectly correct, hierarchical scheme..

Such an approach could work well in a scenario in which we have multiple communications, for which in every single one we puncture in order or in quasi-perfect order. Assume we have c communications, if we puncture in perfect order for each of these communications we would need to store c keys. If we puncture on a tag, we can use a one-way function to derive the next key in the sequence.

Assuming a quasi-perfect order for each communication with a window size of w_i for the $i - th$ communication, we would need a storage space of $\mathcal{O}(\sum_i w_i)$.

Note that we could chain together any kind of PE scheme and could even allow a mixture of different schemes, like chaining together ratcheting-based schemes and PBT-based schemes.

Scheme	Class	Goal	Decision
Ratcheting	Perfectly correct, hierarchical	Storage reduction	✓
Chained BFE	Dynamic	Storage reduction	×
Dynamic PBT	Dynamic	Storage reduction	×
Perfectly correct BFE	Perfectly correct, non-hierarchical	Perfect correctness	×
Chained BFE	Dynamic	Perfect correctness	×
BF Cascade	Perfectly correct, non-hierarchical	Perfect correctness	×
Re-encryption	Dynamic	Storage reduction	×
λ-structures	Perfectly correct, hierarchical	Storage reduction	✓
Unbounded BFE	Non-perfectly correct, non-hierarchical	Unbounded tag-space	×
Dynamic Naïve	Dynamic	Storage reduction	✓

Figure 8.7: Summary of newly explored PE schemes. For each approach we note its corresponding class of PE schemes, the goal we hoped to achieve with the scheme and if it got discarded (cross) or approved (checkmark).

8.9 BFE-based Puncturable Encryption with unbounded tag support

If we adapt the original BFE-based PE scheme to instead of deleting keys on a puncture call to replace them by a new one, we would end up with an approach having the same correctness error as the original BFE-based approach but with the added benefit that this scheme would support an unbounded number of tags. Such a scheme would, as the original BFE-based scheme, be non-hierarchical and not able to achieve perfect correctness.

Note that using this approach, we would lose the property of shrinking key size after puncturing compared to the original approach. Additionally, we would face the problem of deciding whether we are dealing with a new key (replaced by a puncturing call) or the original key on decryption which could greatly affect the efficiency of the decryption algorithm.

8.10 Dynamic Naïve Solution

Making the naïve solution dynamic by additionally providing a key extension algorithm to allow extension of the supported tag-space could allow saving on storage space by trading some algorithm efficiency for it, therefore providing a baseline to beat for new dynamic constructions. As the name suggests this scheme would fall into the class of dynamic PE schemes.

8.11 Exploration Summary

In figure 8.7 we briefly summaries the goal of each new construction, note the class of PE schemes the construction corresponds to, and denote whether we decided such a scheme could be useful in practice. We can see that a PE scheme based on ratcheting as well as the dynamic naïve solution are

the most promising new construction ideas and we therefore make a more detailed analysis of these two approaches in chapter 9.

8.12 Discarded data structures

In this section we list data structures we considered for construction of a PE scheme but decided not to use. We shortly describe the reasons why we think these data structures (or classes of data structures) are not suitable to implement a PE scheme.

8.12.1 Cuckoo Filters

Since cuckoo hashing (as described by N. Fleming [9]) changes the position of elements in the hash table and therefore also in the cuckoo filter, it cannot be used to assign a tag to its key component as it is done in the BFE-based approach. Therefore, one could use a cuckoo filter to test if a tag is already in use in a quite efficient way but needs to store the mapping from a tag to its key component in some different way.

The only way we could think of on how to use the cuckoo filter directly to store the mapping from a tag to its key component would be to fix the set of tags and insert all of them in the cuckoo filter. After successfully inserting all tags, we would use the key component associated to the entry the tag is mapped to in the filter. Using this idea, one would end up with a one-to-one mapping from each tag to a key component, and therefore basically the naïve solution.

In conclusion, a cuckoo filter may be a good solution for PE schemes which need an efficient method to check for already punctured tags, but it is an ineffective method to implement a PE scheme all by itself.

8.12.2 Morton Filters

Morton filters (as described by Breslow and Jayasena [2]) are optimized cuckoo filters, meaning they are faster and more space-efficient. Therefore, we come to the conclusion that morton filters are not suitable for implementing a PE scheme, for similar reasons we decided not to use cuckoo filters.

8.12.3 Counting Filters

Using counting filters, one could not only check if an element is part of a set but also get info about how many times an item is in a set. The only way we could think of on how to use the additional information provided by a counting filter is, for example in a BFE-based approach, to check if all keys associated to a new tag were already used before to encrypt messages

under different tags. This then could lead to the problem that we puncture on the new tag by calling the puncture algorithm on all the other tags (this is the reason for the correctness error in the original BFE-based approach). Since we can achieve such a collision detection with standard Bloom filters, as described in 8.4 we decided to not further investigate in the possibility of counting-filter-based PE schemes.

8.12.4 Self-balancing trees

Self-balancing trees, such as AVL-Trees, B-Trees, or Splay-trees, are not well suited as an underlying data structure for a PE scheme. This is due to their self-balancing property. The main intuition why trees are a suitable choice to implement a PE scheme is their hierarchical structure, which allows for compact storage. Since self-balancing trees reorder their elements to balance them self, the hierarchical structure can change and therefore keys could not be derived correctly any more.

8.12.5 Heaps

Since heaps, similar to self-balancing trees, reorder their elements, they can not provide compact storage as it is done in a PBT-based approach. Storing keys in nodes of a heap would require to store one key component for each tag. We would end up with an optimized naïve approach which could support an unbounded number of tags. Since there also exist a lot of other ideas on how to extend the naïve solution to support an unbounded number of tags we conclude that a PE scheme based solely on heaps is inefficient.

Note that we could use heaps to organize the mapping from arbitrary string tags to internally used tags, if one decides to support such tags. We then could store the mapping from user-tag to internally used tag in a node of the heap and give it an additional attribute such as creation time or a hierarchy level. Based on this attribute, we then could define a max or min heap to allow faster access to the correct key component of more recently used tags etc.

Chapter 9

Theoretical Constructions

From the new construction ideas presented in chapter 8 we decided that a dynamic naïve SPE scheme and a PE scheme based on ratcheting are the most promising candidates to achieve benefits over the naïve construction. In this chapter we present a theoretical construction of both approaches and based on it perform a theoretical analysis. We compare each approach to the naïve solution to get a direct comparison of their performance and highlight for each feature which scheme performs better.

9.1 Dynamic Naïve SPE

The dynamic naïve SPE scheme is a new naïve construction of dynamic PE. We adapted the original naïve construction, presented in chapter 4, to become a dynamic PE scheme, which we introduced in chapter 7.

This construction could save on secret-key storage since by being a dynamic PE scheme it does not support the entire tag-space from the beginning and therefore does not need to store all key components for all supported tags. Additionally, the scheme borrows the efficiency of the naïve scheme's algorithms for all tags which do not require key extension. Moreover, being able to extend the secret key allows this scheme to theoretically support an unbounded tag-space, which is not possible with hitherto used schemes.

In the following, we provide a more detailed description of the functionality of the scheme. We only provide a construction for symmetric encryption due to the problems which arise when using a DPE scheme for public-key encryption (as described in section 7.2).

A dynamic naïve SPE scheme initially supports a small tag-space $\mathcal{T}_{init}$ with $|\mathcal{T}_{init}| = n_{init}$, i.e. it initially supports n_{init} tags. Once it runs out of key components it uses a key extension algorithm to add support for an additional set of l tags.

Initialization

Initialization works in the same way as in the original naïve solution presented in chapter 4 with the main difference being that we only initialize the scheme for n_{init} tags.

Algorithm 10 Key Generation for dynamic naïve SPE

```
1: function KeyGen(λ, n_init, [])
2:     SK ← Array of length n_init                    ▷ Initialize array
3:     for i from 0 to n_init − 1 do                  ▷ Precompute key components
4:         SK[i] ←$ SE.KeyGen(λ)
5:     end for
6:     return SK
7: end function
```

Encryption, Decryption and Puncturing

Encryption, decryption and puncturing function in the same way as in the original naïve approach, but the algorithms additionally throw an error once they detect that key extension is required.

Algorithm 11 Encryption for for naïve dynamic SPE

```
1: function Encrypt(SK_i, M, τ)
2:     t ← getIndex(τ)                    ▷ Compute index τ corresponds to
3:     if t ≥ |SK_i| then                 ▷ Require key extension
4:         Error: Require key extension
5:         return ⊥
6:     end if
7:     if SK_i[t] == ⊥ then               ▷ Punctured on τ before
8:         return ⊥
9:     end if
10:    C ←$ SE.Enc(SK_i[t], M)            ▷ Encrypt message
11:    return C
12: end function
```

Algorithm 12 Decryption for naïve dynamic SPE

```
1: function DECRYPT(SK_i, C, τ)
2:     t ← getIndex(τ)                    ▷ Compute index τ corresponds to
3:     if t ≥ |SK_i| then                  ▷ Require key extension
4:         Error: Require key extension
5:         return ⊥
6:     end if
7:     if SK_i[t] == ⊥ then                ▷ Punctured on τ before
8:         return ⊥
9:     end if
10:    M ← SE.DEC(SK_i[t], C)              ▷ Decrypt message
11:    return M
12: end function
```

Algorithm 13 Puncturing for naïve dynamic SPE

```
1: function PUNCTURE(SK_i, τ)
2:     t ← getIndex(τ)                    ▷ Compute index τ corresponds to
3:     if t ≥ |SK_i| then                  ▷ Require key extension
4:         Error: Require key extension
5:         return ⊥
6:     end if
7:     SK_i[t] ← ⊥              ▷ Delete key component corresponding to τ
8:     return SK_i
9: end function
```

Key extension

The key extension algorithm takes as input the current secret-key, a number l, which denotes by how many key components extension is needed and a security parameter λ. It then extends the current secret-key by computing l new key components using the black-box key generation algorithm. Finally, it returns the new secret-key.

Algorithm 14 Key extension for naïve dynamic SPE

```
1: function KEYEXT(SK, [l, λ])
2:     s ← |SK|
3:     SK_new ← Array of length |SK| + l
4:     SK_new[0..s − 1] ← SK               ▷ Copy current secret-key
5:     for i from s to s + l − 1 do        ▷ Extend the key
6:         SK_new[i] ← SE.KEYGEN(λ)
7:     end for
8:     return SK_new
9: end function
```

9.1.1 Theoretical analysis

Compared to the standard naïve SPE scheme, the dynamic version can reduce the required storage whilst still providing fast algorithms for all tags which do not require key extension. Additionally, such a dynamic approach can gain benefits from both a puncturing before next encryption and a quasi-perfect ordering. For the first one, we only require to store $\mathcal{O}(max(n_{init}, l))$ key components at any point in time. Since we technically only need one key component in this scenario, choosing $l = n_{init} = 1$ would result in a scheme requiring $\mathcal{O}(1)$ secret-key storage[1]. For a quasi-perfect ordering with window size w, we can reduce the storage to $\mathcal{O}(max(n_{init}, w + l))$. This is the case since we technically would only need to store w key components at any point in time, but it is possible that certain tags of the current window require key extension, therefore increasing the needed storage.

We conclude the following list of benefits:

- Fast encryption, decryption and puncturing:

  For all tags which do not require key extension.

- Smaller secret-key size:

  The required secret-key storage is at most as large as in the original naïve solution, and even smaller in most cases.

- Achieves perfect correctness
- Can achieve benefits from special puncturing orders
- Unbounded tag support:

  An implementation for $n \to \infty$ is possible using this approach.

Additionally, we get the following list of drawbacks:

- Key size is linear in the current number of supported tags
- Slower algorithms for tags requiring key extension
- Non-deterministic key extension algorithm:

  Since key extension relies on the randomized key extension algorithm of the underlying SE scheme, an extended key would need to be securely distributed to all other parties involved in the communication. This is a major drawback of this approach.

We provide a full summary of the features achieved by this scheme as well as a comparison to the original naïve SPE solution (presented in chapter 4)

[1] In this scenario this approach is similar to a ratcheting-based PE scheme presented in section 9.2.

in figure 9.1. The dynamic naïve SPE construction beats the original naïve construction in almost any aspect except for the efficiency of the encryption, decryption and puncturing algorithms. It is important to note that these algorithms are only less efficient compared to the algorithms from the original naïve solution for tags which require key extension. For all other tags, the runtime of the algorithms is the same as in the naïve SPE scheme.

Feature summary and comparison

Variable	Description
n	Number of supported tags
n_{init}	Number of initially supported tags
n_{cur}	Number of currently supported tags
u	Number of unpunctured tags
u_{cur}	Number of unpunctured tags from the set of currently supported tags
l	Number of newly supported tags after key extension
t	Runtime of black-box SE.KeyGen() algorithm
e	Runtime of black-box SE.Enc() algorithm
d	Runtime of black-box SE.Dec() algorithm

Feature	Naïve DSPE	Naïve SPE
Secret-Key Storage		
Initial	$\Theta(n_{init})$	$\Theta(n)$
During use	$\Theta(u_{curr})$	$\Theta(u)$
Worst case	$\Theta(n)$	$\Theta(n)$
Best case	$\mathcal{O}(1)$	$\Theta(u)$
Computation time		
KeyGen()	$\mathcal{O}(n_{init}t)$	$\mathcal{O}(nt)$
Encrypt()	$\mathcal{O}(e+lt)$	$\mathcal{O}(e)$
Puncture()	$\mathcal{O}(1+lt)$	$\mathcal{O}(1)$
Decrypt()	$\mathcal{O}(d+lt)$	$\mathcal{O}(d)$
Tag support		
Size of tag-space	Possibly unbounded	Bounded
Correctness		
Achieves	Perfect	Perfect
Special case benefits		
In order puncturing	No benefits	No benefits
Puncturing before next encryption	Storage: $\mathcal{O}(max(n_{initi}, l))$	No benefits
Quasi-perfect ordering	Storage: $\mathcal{O}(max(n_{init}, w+l))$	No benefits

Figure 9.1: Features achieved by the naïve dynamic SPE scheme compared to a standard naïve SPE scheme. The better performing scheme is highlighted in green and the worse performing one in red.

9.2 Ratcheting

In contrast to the dynamic naïve SPE scheme presented in section 9.1 a PE scheme based on ratcheting could save on storage space due to its hierarchical structure. As a result of the way in which the individual message keys get derived, a scheme based on ratcheting could work especially well in a scenario where the tags get used up in sequence.

Recall, the main idea behind the ratcheting approach is to derive a key associated to a tag from a ratchet key. We now present a theoretical construction and analysis of this approach. We assume to have access to two one-way functions f_{mk} and f_{rk} which map a ratchet key to a message key and a new version of the ratchet key respectively. Additionally, we assume sequence numbers as the used tags, therefore the number of supported tags n directly defines the supported tag-space.

Key Generation

To initialize the scheme, we run the key generation algorithm. It takes a security parameter λ, a number of supported tags n and no additional arguments as input. It then initializes the ratchet key , RK, making use of the key generation algorithm of the black-box symmetric encryption scheme. Additionally it initializes the state variable $ratchetN$ which stores the current version number of the ratchet key. In a next step it initializes an empty array, KA, of size n, in which later on the keys are stored. Finally, it returns the secret-key consisting of KA, RK and $rachetN$.

Algorithm 15 Key Generation

1: **function** KeyGen($\lambda, n, []$)
2: $\quad RK \leftarrow_{\$}$ SE.KeyGen(λ) ▷ Initialize version 0 ratchet key
3: $\quad rachetN \leftarrow 0$
4: $\quad KA \leftarrow$ `Empty array of size` n
5: $\quad$ **return** $\{KA, RK, ratchetN\}$
6: **end function**

Encryption

To encrypt a message M under a tag τ the algorithm first computes the sequence number corresponding to τ. It then checks if it is higher or lower than the current ratchet key version stored in $ratchetN$. If it is lower we check the KA entry corresponding to the tag and either encrypt the message with the key stored there, or if we previously punctured on the tag and the entry holds $\perp$, encryption fails. Otherwise, i.e. the sequence number

is higher than the ratchet key version number, we iteratively use f_{rk} on the current ratchet key until we reach the ratchet key version corresponding to the tags sequence number and use f_{mk} to derive the message key, which the algorithm then uses to encrypt the message. Finally, it returns the encrypted message[2].

Algorithm 16 Encryption

```
1:  function Encrypt(SK, M, τ)
2:      t ← getIndex(τ)                       ▷ Compute index τ corresponds to
3:      if t < ratchetN then
4:          key ← KA[t]                       ▷ Access key component
5:          if key == ⊥ then                  ▷ Previously punctured on the tag
6:              return ⊥
7:          else
8:              C ← SE.Enc(key, M)
9:          end if
10:     else                                  ▷ Need to derive the key component
11:         temp ← ratchetN
12:         RK_temp ← RK
13:         while temp < t do
14:             RK_temp ← f_rk(RK_temp)       ▷ Evolve temporary ratchet key
15:             temp ← temp + 1
16:         end while
17:         key ← f_mk(RK_temp)               ▷ Derive encryption key
18:         C ← SE.Enc(key, M)
19:     end if
20:     return C
21: end function
```

Puncturing

Puncturing on a tag τ can happen in two different ways. Either the sequence number corresponding to the tag is smaller than the current ratchet key version, i.e. if we did not previously puncture on the same tag there is a key stored inside KA at the position indicated by the tag, in which case we remove the key from KA. Otherwise, the sequence number corresponding to the tag is larger than the ratchet key version, in which case we need to evolve the ratchet key until its version number is higher than the sequence

[2]Note that we store the current ratchet key version as well as all messages keys for unpunctured tags τ_i whose corresponding sequence numbers t_i have the property that $t_i < ratchetN$.

number corresponding to the tag and while doing this compute the keys for all tags corresponding to a sequence number between *ratchetN* and the one of the tag to be punctured on[3]. The algorithm then returns the new secret-key with updated values for *KA*, *ratchetN* and *RK*.

Algorithm 17 Puncturing

```
 1: function PUNCT(SK, τ)
 2:     t ← getIndex(τ)                    ▷ Compute index τ corresponds to
 3:     if t < ratchetN then
 4:         KA[t] ← ⊥                                ▷ Delete key component
 5:     else                                        ▷ Need to extend the key
 6:         temp ← ratchetN
 7:         while temp < t do
 8:             RK ← f_rk(RK)                           ▷ Evolve ratchet key
 9:             temp ← temp + 1
10:             KA[temp] ← f_mk(RK)       ▷ Derive and store key component
11:         end while
12:         KA[t] ← ⊥   ▷ Remove key component corresponding to the tag
13:         RK ← f_rk(RK)                               ▷ Evolve ratchet key
14:     end if
15:     ratchetN ← t + 1
16:     return {KA, RK, ratchetN}
17: end function
```

Decryption

The decryption algorithm works in the same way as the encryption algorithm does, with the only difference being that it uses the black-box decryption function instead of the encryption one.

9.2.1 Theoretical analysis

Due to its design, the secret-key storage of a PE scheme based on ratcheting only grows once we puncture on a tag whose corresponding sequence number is higher than the current ratchet-key-number *ratchetN*. This means if we never puncture at all, the secret-key storage will never grow.

In both an in order puncturing and a puncturing before next encryption scenario we can achieve constant storage since in both cases we only need to store one ratchet key and the ratchet-key-number. In the puncturing before

[3]This is to guarantee perfect correctness.

Algorithm 18 Decryption

```
1:  function DECRYPT(SK, C, τ)
2:      t ← getIndex(τ)                          ▷ Compute index τ corresponds to
3:      if t < ratchetN then
4:          key ← KA[t]                          ▷ Access key component
5:          if key == ⊥ then                     ▷ Previously punctured on the tag
6:              return ⊥
7:          else
8:              M ← SE.Dec(key, C)
9:          end if
10:     else
11:         temp ← ratchetN
12:         RK_temp ← RK
13:         while temp < t do
14:             RK_temp ← f_rk(RK_temp)          ▷ Evolve temporary ratchet key
15:             temp ← temp + 1
16:         end while
17:         key ← f_mk(RK_temp)                  ▷ Derive decryption key
18:         M ← SE.Dec(key, C)
19:     end if
20:     return M
21: end function
```

next encryption case, this can lead to slow algorithms since the time needed to derive the correct key grows the further away the sequence number corresponding to the tag gets. Such a problem does not occur in a puncturing before next encryption scenario, in which the used key can always be derived using the current ratchet key.

In a quasi-perfect ordering with window size w, we can achieve the same best and worst case storage requirements and algorithm runtimes as in a ratcheting-based PE scheme supporting $n = w$ tags.

We arrive at the following list of benefits:

- Smaller secret-key size:

  In most cases the required secret-key storage is smaller compared to the naïve solution, and in the worst case it will be at most as large as in a naïve scheme.

- Achieves perfect correctness
- Unbounded tag support

  An implementation using $n \to \infty$ is possible using this approach.

- Achieves benefits from special puncturing orders

Additionally, we get the following drawbacks:

- High dependency on puncturing order:

  The required secret-key storage as well as the runtime of the involved algorithms highly depend on the puncturing order.

- Slower encryption, decryption and puncturing:

  Due to the additional step of deriving the message key from the ratchet key the runtimes of the encryption, decryption and puncturing algorithms is even in the best case slower compared to the algorithms of the naïve solution.

We provide a full summary of the features achieved by this scheme as well as a comparison to the original naïve SPE solution (presented in chapter 4) in figure 9.2. As we can see, a ratcheting-based PE scheme can beat the naïve solution in almost all aspects. Its main drawback lies in the less efficient runtimes of the encryption, decryption and puncturing algorithms, although this result was to be expected as any PE scheme which is able to achieve better storage requirements compared to the naïve solution must have less efficient algorithms (as we proved in corollary 6.3).

Feature summary and comparison

Variable	Description
n, u	Number of supported tags, number of unpunctured tags
$ratchetN$	Current ratchet-key-number
u_{cur}	Number of unpunctured tags with sequence numbers $t < ratchetN$
f	Time needed to compute new ratchet key or message key, i.e. compute $f_{rk}()$ or $f_{mk}()$
seq_τ	Sequence number corresponding to tag τ
w	Window size of quasi-perfect ordering
t, e, d	Runtime of black-box SE.KeyGen(), SE.Enc(), SE.Dec() algorithm
z	$seq_\tau - ratchetN$

Feature	Ratcheting-based SPE	Naïve SPE
Secret-Key Storage		
Initial	$\mathcal{O}(1)$	$\Theta(n)$
During use	$\Theta(u_{cur})$	$\Theta(u)$
Worst case	$\Theta(n)$	$\Theta(n)$
Best case	$\mathcal{O}(1)$	$\Theta(u)$
Computation time		
KeyGen()	$\mathcal{O}(t)$	$\mathcal{O}(nt)$
Encrypt()	$\mathcal{O}(max(1, z+1) * f + e)$	$\mathcal{O}(e)$
Puncture()	$\mathcal{O}(max(1, 2z+1) * f)$	$\mathcal{O}(1)$
Decrypt()	$\mathcal{O}(max(1, z+1) * f + d)$	$\mathcal{O}(d)$
Tag support		
Size of tag-space	Possibly unbounded	Bounded
Correctness		
Achieves	Perfect	Perfect
Special case benefits		
In order puncturing	Storage: $\mathcal{O}(1)$	No benefits
Puncturing before next encryption	Storage: $\mathcal{O}(1)$ Encryption: $\mathcal{O}(f+e)$ Puncturing: $\mathcal{O}(f)$ Decryption: $\mathcal{O}(f+d)$	No benefits
Quasi-perfect ordering	Storage: $\mathcal{O}(w)$ Encryption: $\mathcal{O}(w * f + e)$ Puncturing: $\mathcal{O}(w * f)$ Decryption: $\mathcal{O}(w * f + d)$	No benefits

Figure 9.2: Features achieved by the SPE scheme based on ratcheting compared to naïve SPE. The better performing scheme is highlighted in green and the worse performing one in red.

Chapter 10

Discussion and Conclusion

We wanted to investigate how to find new constructions of PE and tried to come up with at least one new PE scheme which can beat the naïve solution in some regard. To this end, we analysed hitherto used schemes and the naïve solution and tried to extract relevant features from their underlying data structures to establish a blueprint which could be used to help find new constructions.

The analysis of hitherto used schemes in chapter 5 showed that a data structure does not need to provide much to be able to suit as a basis for a PE scheme. Providing key storage and a mechanism to delete key components is enough to function as a basis for a PE scheme.

This led to a closer investigation on how certain features for new schemes could potentially be improved compared to the naïve solution (presented in chapter 6). Since encryption, decryption and puncturing are possible in constant time in the naïve scheme, we focused on possibilities which could allow for storage reduction. The research led to a necessary requirement for a PE scheme to reduce storage space compared to the naive solution while still achieving perfect correctness, namely needing to by hierarchical. We also found that to be able to achieve perfect correctness, a PE scheme needs to provide at least one unique key component for every supported tag.

By trying to effectively reduce storage whilst maintaining reasonably fast algorithms, we came up with a new class of PE schemes which we called dynamic PE schemes (presented in chapter 7). The main difference between a standard PE scheme and a dynamic one is that dynamic schemes do not support the entire tag space for the beginning, i.e. after initialization the scheme supports only n_{init} tags, where $n_{init} < n$ and n denotes the size of the entire tag-space. Such dynamic schemes also allow support of theoretically unbounded tag-spaces, which is not possible with hitherto constructions.

Based on our new knowledge about PE schemes we came up with new

ideas on how to construct PE presented in chapter 8. From these ideas we selected the two most promising candidates, naïve DSPE and ratcheting, and presented a theoretical implementation and analysis of the approaches (see chapter 9). We found that both schemes are able to achieve better storage requirements compared to the naïve solution.

10.1 Discussion

Our two new constructions, naïve DSPE and PE based on ratcheting, are both able to outperform the storage requirements of the naïve solution by trading some algorithm efficiency for it. How good this trade-off ends up being highly depends on the puncturing order. Especially the ratcheting-based PE scheme is sensitive to changes in the puncturing order. Our analysis showed that such a scheme works especially well if we puncture in order or in a quasi-perfect order. Therefore, a ratcheting based PE scheme works well in applications in which we are likely to encounter such puncturing orders.

We provide a comparison between the naïve solution, the hitherto used constructions, BFE-based and PBT-based, as well as our two new constructions in figure 10.1. We can see that a PBT-based and a ratcheting-based PE scheme have the best initial storage requirement, namely both only store one single key, whereas a BFE-based scheme has the worst initial storage requirement. During use, the dynamic naïve solution and the ratcheting-based scheme outperform the other schemes. Their storage only depends on the number of punctured tags from their currently supported set of tags. These two schemes are also able to achieve the best best case storage bound.

The naïve solution remains unbeaten regarding algorithm efficiency for encryption, puncturing and decryption. Only the PBT-based scheme, the naïve DSPE scheme and the ratcheting-based scheme can achieve a better runtime for the key generation algorithm, whereby the PBT-based and the ratcheting based schemes achieve the best possible runtime (since they only need to initialize one key).

In the figure we also see that the ratcheting-based approach can profit the most from special cases. This is due to its high sensitivity to the order of puncturing calls. Also, we can see that a BFE-based scheme cannot outperform the naïve solution in any case.

For all presented constructions, we assumed to be using sequence numbers as tags. An analysis on how to efficiently associate arbitrary strings to tags is out of scope for this thesis. We restricted ourselves to PE schemes based on data structures. Also, we did not include any special analysis regarding *fully puncturable encryption*, as introduced by Derler et al. [7], which introduces a notion of positive puncturing calls.

Realizing the storage reduction achieved by the presented schemes is not an easy thing to do in practice due to the complexity of secure deletion. One would need to perform regular ´clean-ups´, either based on time passed or on the number of puncturings performed, to take advantage of the reduced key size. How this can efficiently be achieved is out of scope of this thesis.

Variable	Description
n	Number of supported tags
n_{init}	Number of initially supported tags, $n_{init} < n$
n_{cur}	Number of currently supported tags , $n_{cur} \leq n$
u	Number of unpunctured tags
$ratchetN$	Current ratchet-key-number
u_{cur1}	Number of unpunctured tags from the set of currently supported tags
u_{cur2}	Number of unpunctured tags with sequence numbers $t < ratchetN$
f	Time needed to compute new ratchet key or message key, i.e. compute $f_{rk}()$ or $f_{mk}()$
seq_{τ}	Sequence number corresponding to tag τ
w	Window size of quasi-perfect ordering
l	Number of newly supported tags after key extension
p	Upper bound on the false positive probability of the Bloom filter
k	Amount of hash functions used
h	Time needed to compute one hash value
s	Size of the Bloom filter, $s = -\frac{1}{\sqrt[ke]{1-\sqrt[k]{p}}-1}$
t	Runtime of black-box SE./PkPE.KeyGen() algorithm
e	Runtime of black-box SE./PkPE.Enc() algorithm
d	Runtime of black-box SE./PkPE.Dec() algorithm
z	$seq_{\tau} - ratchetN$

Feature	Naïve SPE/PkPE	BFE-based PE	PBT-based PE	Naïve DSPE	Ratcheting-based SPE
Secret-Key Storage					
Initial	$\Theta(n)$	$\Theta(s)$	$\mathcal{O}(1)$	$\Theta(n_{init})$	$\mathcal{O}(1)$
During use	$\Theta(u)$	$\mathcal{O}(s)$	$\mathcal{O}(\frac{n}{2})$	$\Theta(u_{cur1})$	$\Theta(u_{cur2})$
Worst case	$\Theta(n)$	$\Theta(s)$	$\Theta(\frac{n}{2})$	$\Theta(n)$	$\Theta(n)$
Best case	$\Theta(u)$	$\mathcal{O}(s)$	$\Theta(\log_2(n))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Computation time					
KeyGen()	$\mathcal{O}(nt)$	$\mathcal{O}(st)$	$\mathcal{O}(t)$	$\mathcal{O}(n_{init}t)$	$\mathcal{O}(t)$
Encrypt()	$\mathcal{O}(e)$	$\mathcal{O}(hke)$	$\mathcal{O}(e+\log_2(n))$	$\mathcal{O}(e+lt)$	$\mathcal{O}(max(1,z+1)*f+e)$
Puncture()	$\mathcal{O}(1)$	$\mathcal{O}(hk)$	$\mathcal{O}(\log_2(n))$	$\mathcal{O}(1+lt)$	$\mathcal{O}(max(1,2z+1)*f)$
Decrypt()	$\mathcal{O}(d)$	$\mathcal{O}(hkd)$	$\mathcal{O}(d+\log_2(n))$	$\mathcal{O}(d+lt)$	$\mathcal{O}(max(1,z+1)*f+d)$
Tag support					
Size of tag-space	Bounded	Bounded	Bounded	Possibly unbounded	Possibly unbounded
Correctness					
Achieves	Perfect	Non-negligible	Perfect	Perfect	Perfect
Special case benefits					
In order puncturing	No benefits	No benefits	Storage: $\mathcal{O}(\log_2(n))$	No benefits	Storage: $\mathcal{O}(1)$
Puncturing before next encryption	No benefits	No benefits	Storage: $\mathcal{O}(\log_2(n))$	Storage: $\mathcal{O}(max(n_{init},l))$	Storage: $\mathcal{O}(1)$ Encryption: $\mathcal{O}(f+e)$ Puncturing: $\mathcal{O}(f)$ Decryption: $\mathcal{O}(f+d)$
Quasi-perfect ordering	No benefits	No benefits	Storage: $\mathcal{O}(\log_2(n)+w)$	Storage: $\mathcal{O}(max(n_{init},w+l))$	Storage: $\mathcal{O}(w)$ Encryption: $\mathcal{O}(w*f+e)$ Puncturing: $\mathcal{O}(w*f)$ Decryption: $\mathcal{O}(w*f+d)$

Figure 10.1: Comparison of the naïve solution to hitherto used schemes, PBT-based and BFE-based, and to new schemes, naïve DSPE and ratcheting based PE. The best performing scheme is coloured green and the worst performing one is coloured red. Blue coloured schemes perform better and orange coloured schemes perform worse than the naïve solution for the specific feature.

10.2 Important Features, DSPE Conclusion and Future Work

Our analysis of hitherto used schemes showed that a data structure does not need to provide much to be able to suit as a basis for a PE scheme. Namely, any structure providing key storage and a mechanism to delete key components can be used to implement a PE scheme.

More interesting are the requirements we found to achieve certain benefits. To be able to reduce the secret-key storage and preserve perfect correctness one needs to use a hierarchical data structure and to guarantee perfect correctness a PE scheme needs to provide at least one unique key component for every supported tag. These two discoveries led to the conclusion that the naïve PE scheme is optimal regarding required storage space and algorithm efficiency for its class of non-hierarchical, non-dynamic PE schemes, achieving perfect correctness. We also showed that any PE scheme which achieves perfect correctness and better storage requirements compared to the naïve solution has to have less efficient algorithms for encryption, decryption and/or puncturing due to the need to first derive some key component(s) before encryption / decryption / puncturing can take place.

The new class of dynamic PE schemes we introduced is able to achieve storage benefits without using hierarchical data structures by not supporting the entire tag-space form the beginning, but rather increase it on demand. This introduces a hierarchy between initially supported tags and later on added ones. Therefore, any dynamic PE scheme can be considered to be hierarchical even if the underlying data structure is not (for example the naïve DSPE scheme).

As a continuation of this project, we would construct real implementations of our theoretically constructed schemes and analyse their performance in simulated scenarios. Additionally, analysing different use cases for PE to single out the most common puncturing orders could help find new application-specific schemes with good performance.

We conclude that further research should focus on dynamic PE and schemes based on hierarchical data structures and focus on achieving the best possible trade-off between storage reduction and algorithm efficiency, whilst keeping application specific performance in mind.

Appendix A

Appendix

A.1 BFE vs. naïve storage comparison

In the following plots (A.1) we compare the initial secret-key storage required for the naïve solution and a BFE-based PE scheme for different upper bounds on the false positive probability p of the Bloom filter and different amount of involved hash functions k.

A.2 In order puncturing in PBT based PE

The following example serves as an intuition on why the key size can be bounded by $log_2(n)$ while doing in order puncturing in a Perfect Binary Tree Puncturable Encryption scheme. We puncture from ´left to right´ and indicate the nodes we need to store at the current iteration by colouring them grey. Nodes that get deleted after a puncturing call get crossed out.

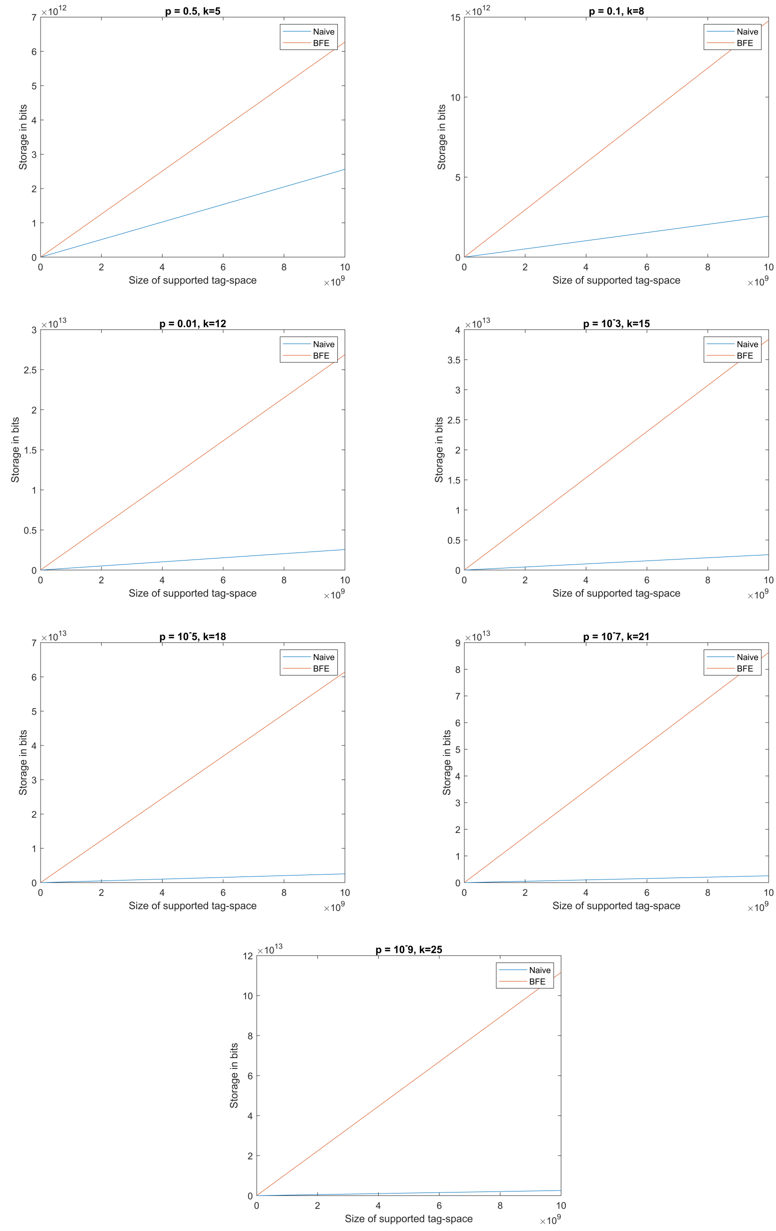
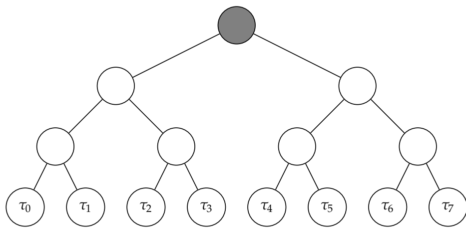
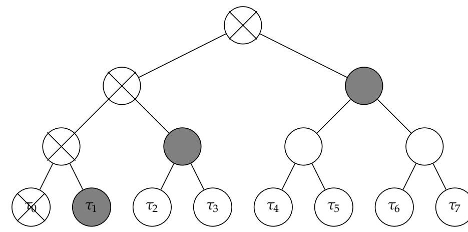


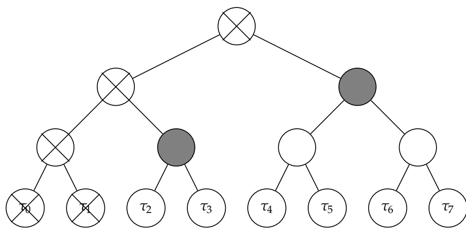
Figure A.1: Secret-key storage comparison of a BFE-based PE scheme achieving an upper bound on the false positive probability of the Bloom filter of p using k hash functions and the naïve solution. Both schemes are using 256-bit secret-keys.



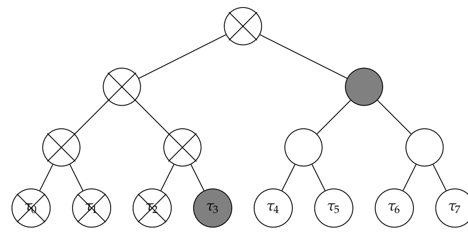
Un-punctured tree for $n = 8$ tags.



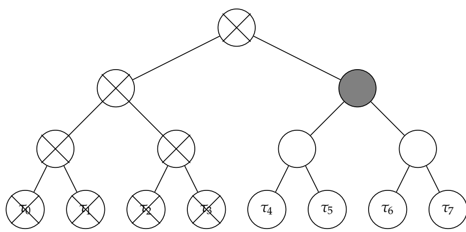
Punctured on the first tag τ_0.



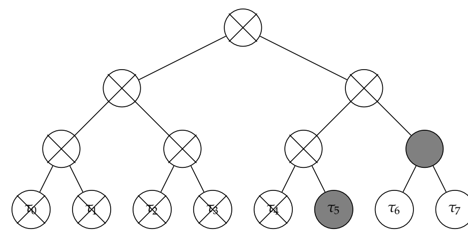
Punctured on τ_0 and τ_1.



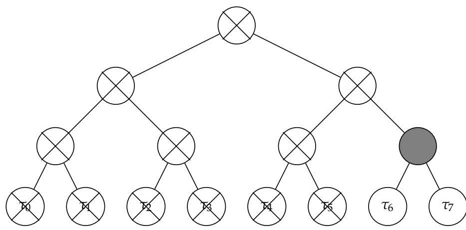
Punctured on τ_0, τ_1, and τ_2.



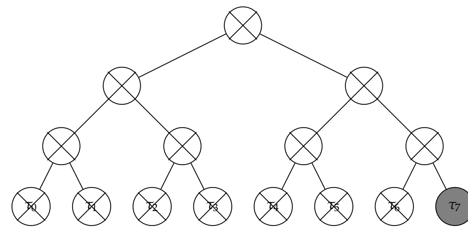
Punctured on τ_0, τ_1, τ_2, and τ_3.



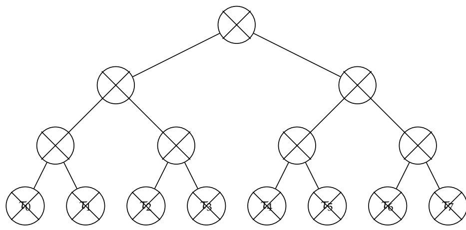
Punctured on τ_0, τ_1, τ_2, τ_3, and τ_4.



Punctured on τ_0, τ_1, τ_2, τ_3, τ_4, and τ_5.



Punctured on τ_0, τ_1, τ_2, τ_3, τ_4, τ_5, and τ_6.



Punctured on all tags.

Bibliography

[1] Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 117–150. Springer, Heidelberg, May 2019.

[2] Alex D. Breslow and Nuwan S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.*, 11(9):1041–1055, May 2018.

[3] Valerio Cini, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. CCA-secure (puncturable) KEMs from encryption with non-negligible decryption errors. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 159–190. Springer, Heidelberg, December 2020.

[4] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 1115–1127, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013.

[6] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors,

Advances in Cryptology – EUROCRYPT 2018, Part III, volume 10822 of *Lecture Notes in Computer Science*, pages 425–455. Springer, Heidelberg, April / May 2018.

[7] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. Cryptology ePrint Archive, Report 2018/321, 2018. https://eprint.iacr.org/2018/321.

[8] David Derler, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. I want to forget: Fine-grained encryption with full forward secrecy in the distributed setting. Cryptology ePrint Archive, Report 2019/912, 2019. https://eprint.iacr.org/2019/912.

[9] Noah Fleming. Cuckoo Hashing and Cuckoo Filters. 2018.

[10] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE Computer Society Press, May 2015.

[11] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 519–548. Springer, Heidelberg, April / May 2017.

[12] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. Cryptology ePrint Archive, Report 2017/223, 2017. http://eprint.iacr.org/2017/223.

[13] Shi-Feng Sun, Amin Sakzad, Ron Steinfeld, Joseph Liu, and Dawu Gu. Public-key puncturable encryption: Modular and compact constructions. Cryptology ePrint Archive, Report 2020/126, 2020. https://eprint.iacr.org/2020/126.

[14] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 763–780. ACM Press, October 2018.

[15] Willy Susilo, Dung Hoang Duong, Huy Quoc Le, and Josef Pieprzyk. Puncturable encryption: A generic construction from delegatable fully key-homomorphic encryption, 2020.

[16] M. Xia and L. Yin. A generic construction of puncturable encryption. In *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 120–124, 2019.

[17] T. V. Xuan Phuong, R. Ning, C. Xin, and H. Wu. Puncturable attribute-based encryption for secure data delivery in internet of things. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1511–1519, 2018.