



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Implementing a Forward-Secure Cloud Storage System

Master Thesis

Younis Khalil

August 12, 2023

Supervisors: Prof. Dr. Kenny Paterson, Dr. Felix Günther, Matilda Backendal

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



---

## Abstract

Cloud storage is used extensively, yet the promises of storage providers on security and privacy remain in doubt: most times, cloud storage providers retain control of the encryption keys used to encrypt data at rest. This means that the providers can theoretically access the stored data at any time. To mitigate the issue of trust, we aim to present a cloud storage system with forward-secure deletion, which makes use of a puncturable key-wrapping (PKW) scheme. Data is end-to-end encrypted with data encryption keys (DEKs) to provide fine-grained secure file storage, relying on cloud storage providers only for the availability of their service, but taking back control of security and privacy. The DEKs are protected using PKW, the forward security of which allows for an operation called “shred”, which not only deletes a file but also makes it irrecoverable by cryptographic means. We explore the feasibility of forward-secure file storage, evaluate the performance, and propose extensions to the PKW scheme to support file hierarchies.

---

## **Acknowledgements**

I would like to thank Prof. Dr. Kenny Paterson for making it possible for me to conduct this project in the Applied Cryptography Group and for supervising my thesis. I am also deeply thankful to my co-supervisors Matilda Backedal and Dr. Felix Günther for their continued support, their encouragement, and their poignant remarks and feedback. It was a truly enriching experience to work with such knowledgeable people.

---

## Acronyms

<b>AEAD</b>	Authenticated Encryption With Associated Data
<b>DEK</b>	Data Encryption Key
<b>FTP</b>	File Transfer Protocol
<b>GCS</b>	Google Cloud Storage
<b>GGM</b>	Goldreich-Goldwasser-Micali
<b>HPFS</b>	Hierarchical Protected File Storage
<b>hPKW</b>	Hierarchically Puncturable Key-Wrapping
<b>hPPRF</b>	Hierarchically Puncturable Pseudo-Random Function
<b>KEK</b>	Key Encryption Key
<b>PFS</b>	Protected File Storage
<b>PKW</b>	Puncturable Key-Wrapping
<b>PPRF</b>	Puncturable Pseudo-Random Function
<b>PRF</b>	Pseudo-Random Function
<b>PRG</b>	Pseudo-Random Generator



---

# Contents

---

<b>Contents</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries</b>	<b>5</b>
2.1 Notation . . . . .	5
2.2 Cryptographic primitives . . . . .	6
2.2.1 AEAD . . . . .	6
2.2.2 Pseudo-random generators (PRGs) and pseudo-random functions (PRFs) . . . . .	7
2.2.3 Puncturable pseudo-random functions (PPRFs) . . . . .	9
2.2.4 Puncturable key wrapping . . . . .	11
<b>3 Protected file storage</b>	<b>15</b>
3.1 PFS (original definition) . . . . .	15
3.1.1 Confidentiality and integrity . . . . .	16
3.1.2 Construction from PKW and AEAD . . . . .	19
3.1.3 Implementation . . . . .	19
3.1.4 Evaluation . . . . .	21
3.2 PFS without file identifiers (PFS <sup>+</sup> ) . . . . .	29
3.2.1 PFS <sup>+</sup> from PKW and AEAD . . . . .	31
3.2.2 Implementation . . . . .	32
3.2.3 Evaluation . . . . .	33
<b>4 PFS with file hierarchy</b>	<b>37</b>
4.1 Hierarchical protected file storage (HPFS) . . . . .	37
4.2 Constructing hierarchical protected file storage (HPFS) using hierarchically puncturable pseudo-random function (hPPRF)	41
4.2.1 Hierarchically puncturable PRF (hPPRF) . . . . .	41
4.2.2 An instantiation of hPPRF . . . . .	43

## CONTENTS

---

4.2.3	Hierarchically puncturable key-wrapping (hPKW) . . .	44
4.2.4	hPKW from AEAD and hPPRF . . . . .	46
4.2.5	Hierarchical protected file storage (HPFS) construction	46
4.3	Constructing HPFS from PKW . . . . .	49
4.3.1	Detailed explanation of the construction . . . . .	50
4.3.2	Security of the construction . . . . .	52
4.3.3	Single-directory key rotation . . . . .	52
4.4	Implementation . . . . .	52
4.4.1	Extension of the PKW library . . . . .	52
4.4.2	hPKW . . . . .	53
4.4.3	PKW-per-directory . . . . .	53
4.5	Evaluation . . . . .	53
<b>5</b>	<b>Discussion</b>	<b>57</b>
<b>6</b>	<b>Conclusion</b>	<b>61</b>
6.1	Future work . . . . .	61
	<b>Bibliography</b>	<b>63</b>



## Chapter 1

---

# Introduction

---

OneDrive, iCloud, Google Drive, Dropbox. Many commercial cloud storage services promise easy-to-use functionality and reliable, available, secure, and convenient file storage. Often, these services come with integrated backup functionality. And although some promises of security are frequently made, these services seldom offer end-to-end encryption of files, and the encryption keys used to encrypt the files ultimately remain in the hands of the service providers. This gives the cloud full access to the stored data. With end-to-end encryption, files are encrypted before they are uploaded to the cloud and the encryption keys remain in the hands of the users. Therefore, not using end-to-end encryption places complete trust in the providers of these services, which may not be desirable. Hackers may gain unauthorized access to a service [1] and the services must comply with search warrants, some of which may be forged by criminals [2]. Barring this trust, can these services still be used while regaining security and privacy for the data stored using them?

By using end-to-end encryption, file contents can be protected from unwanted inspection. There are services such as BoxCryptor [3], Mega [4], or NextCloud [5], which offer this functionality without a user having to perform encryption themselves. Recently, even iCloud has included the functionality, but it is not active by default [6]. However, there is still an issue of forward secrecy. The notion of forward secrecy in general describes a property of cryptographic systems in which the compromise of long-term secrets does not affect the security of past cryptographic operations. In the context of cloud storage, this means that a storage service can record all encrypted files it has ever seen, in the hopes that at some future point it will gain access to the encryption key. We require some guarantee that deleted files will truly be irrecoverable; even if the keying material is later leaked by some accident or compromised by a malicious actor.

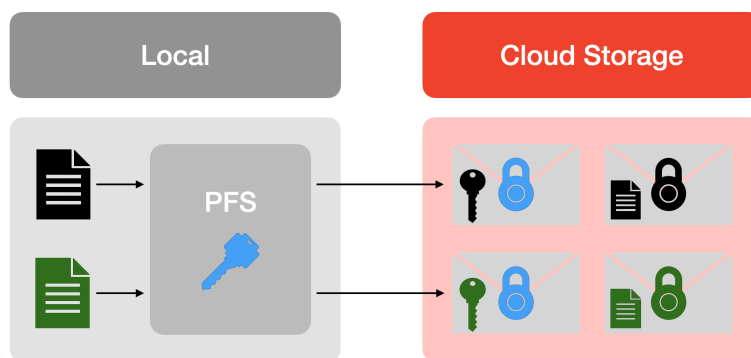
Previous work has focused on attacks against existing cloud storage solu-

tions [7, 8] or on designing an end-to-end encrypted storage system for the cloud in general [9, 10, 11]. Forward-security in the cloud setting has so far been investigated by [12], which focused on quantum security aspects of cloud storage, and in [13, 14], forward-security was investigated in relation to searchable encryption, a growing field of research for cloud storage. Burn-Box [15], gives “self-revocable” encryption, allowing for protection against compelled access. In this work, we focus on achieving forward security for deleted files in a cloud storage system, based on work on puncturable key-wrapping (PKW) [16]. Forward security for deleted files in cloud storage has not been implemented so far, to our knowledge.

This project aims to explore the feasibility of forward-secure cloud storage. Trust is placed in the promises of service providers to provide availability. But the system no longer relies on the cloud for security or privacy. The “secure cloud storage system” handles these aspects locally and provides fine-grained forward security for deleted files. Files can be “shredded” so that even in the case that the user’s decryption key is stolen, deleted files cannot be recovered — even if the cloud service accidentally kept backups of the encrypted files. This is possible because the shredding operation not only deletes a file, but also updates the user’s decryption key (which is located on the user’s device). The updated key is then unable to decrypt the shredded file. This means that the file in cloud storage can be deleted asynchronously (no connection to the cloud is needed). The shredding operation, which operates on just the key, can be local-only, with the physical (irretrievable) encrypted file remaining on the cloud until a periodic clean-up.

The fine-grained nature of deletion can be achieved by protecting the keys used to encrypt files (one key per file) with a key-wrapping [17] scheme. Key-wrapping was recently extended to the primitive “puncturable key-wrapping” (PKW) [16], which provides the promised forward security. The authors lay out a scheme using PKW to instantiate a secure file storage system with forward-secure deletion (protected file storage (PFS) [16]). Figure 1.1 shows at a basic level how PFS is used to securely store files in the cloud. To explore the feasibility, we implement the proposed scheme and test the performance. We present an optimization that requires an update of the syntax and security notions and significantly improves efficiency.

Often, a user may want to delete not only a handful of files but entire file (sub-)hierarchies instead. Consider a directory containing sensitive personal information that is no longer required after a project has been completed. In this case, it would be both cumbersome and somewhat inefficient to shred each file contained in the hierarchy individually. Therefore, this project also explores ways in which file hierarchies can be supported so that the shredding operation cannot be used exclusively at very fine-grained but at the same time at more coarse-grained levels.



**Figure 1.1:** Overview of the fundamental idea behind PFS: for each file, a DEK (shown in the same colour as the file) is used to encrypt it. Both the encrypted file and the wrapped DEK (wrapped using PKW) are stored on the cloud. The long-term secret key (shown in blue) used to wrap the DEKs is stored off the cloud.

In this project, we study the PFS approach and implement it to test its performance. Google Cloud Storage (GCS) is used as a service provider for the storage of encrypted files, but because cryptographic operations happen only locally, the cloud storage service is only for very basic functionality (i.e. uploading and downloading encrypted files). The choice of provider is rather arbitrary and is mainly based on ease of use. The evaluation pays special attention to time efficiency and key size, since these are important metrics for the system. System performance is measured by replaying file access patterns extracted from the log of public GitHub repositories to gain insight into performance on real-world usage data.

We will start by going over some preliminaries in Chapter 2, explaining the notation we use, and introducing the cryptographic primitives that are used to construct the PFS system (authenticated encryption with associated data (AEAD) [18], pseudo-random generators (PRGs), pseudo-random functions (PRFs), puncturable pseudo-random functions (PPRFs) [19, 20], and PKW [16]). Next, in Chapter 3, we will introduce the PFS scheme, along with its security definitions and show how it can be constructed by combining other cryptographic primitives, as presented in [16]. We then show how PFS was taken from this abstract description to a working implementation, describe how it was tested, and we present an optimization that strongly improves performance. We will then explore how the PFS scheme can be extended to support file hierarchies in Chapter 4 and present two constructions of the hierarchical primitive and an evaluation section which then compares these two constructions. In Chapter 5, the performance of the flat and hierarchical approach is compared and analysed. Finally, in Chapter 6, we summarize our findings and outline possible future research directions.

The implementations and resources used for the evaluations can be in-

## 1. INTRODUCTION

---

spected at <https://github.com/younisk/forward-secure-cloud-storage>.

## Chapter 2

---

# Preliminaries

---

We begin by introducing the notation used throughout the thesis. We will then state the definitions and security notions of cryptographic primitives in the way they are used in this work.

### 2.1 Notation

The notation is mostly standard, with some additions to simplify the stating of concepts that recur frequently.

We use the symbol  $\|$  to denote string concatenation,  $a \leftarrow b$  to denote the assignment of value  $b$  to variable  $a$ , and  $x \leftarrow_{\$} \mathcal{X}$  to denote that  $x$  is sampled from the set  $\mathcal{X}$  uniformly at random. Bitstrings of length  $n$  or of any length (including the empty string) are denoted by  $\{0, 1\}^n$  and  $\{0, 1\}^*$ , respectively. The special symbol  $\perp$  (pronounced *bot*) is used to denote rejection. We write  $[n]_t$  to denote the bitstring representation of length  $t$  of the number  $n \in \mathbb{N}$ .

In some instances, an associative map is used. We write  $A[\cdot] \leftarrow \perp$  to denote its empty instantiation and  $A[a] \leftarrow b$  to denote that the value of  $b$  is stored in the map  $A$ , indexed by  $a$ . In  $A[a] \leftarrow b$ ,  $a$  is called the “key” and  $b$  is called the “value”. Tuples are written enclosed with parentheses:  $(a, b, c)$ . To concatenate tuples, or to add a single element to the end, we use the  $+$  operator (e.g.  $(a, b) + (c, d)$  produces  $(a, b, c, d)$ ). As a shorthand,  $t += b$  may be used to express  $t = t + b$ , both for tuples and the standard arithmetic addition. The length of a tuple  $T$  is denoted with  $|T|$ . If a value is returned by an algorithm, but not used, it may be ignored by assigning it to  $\_$ , e.g.  $(v, \_) \leftarrow (a, b)$  ( $v$  gets the value of  $a$ ,  $b$  is ignored). As a shorthand, the union of two sets, as in  $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{T}$ , is often expressed as  $\mathcal{S} \leftarrow^{\cup} \mathcal{T}$ .

## 2.2 Cryptographic primitives

We present cryptographic primitives which lie at the heart of this work and introduce the notation we use when referring to them.

### 2.2.1 AEAD

Authenticated encryption with associated data (AEAD) is a symmetric encryption primitive that provides confidentiality for the encrypted plaintext, and integrity for both the plaintext and additional data (e.g. metadata encoding sender and receiver of a message). We restate the definition and the security games (Figure 2.1) of AEAD [18], as given in [16].

**Definition 2.1** (AEAD scheme). *An authenticated encryption with associated data scheme,  $\text{AEAD} = (\text{Enc}, \text{Dec})$ , is a pair of algorithms with four associated sets; the secret-key space  $\mathcal{SK}$ , the nonce space  $\mathcal{N}$ , the associated data space  $\mathcal{AD}$  and the message space  $\mathcal{M}$ . Also associated to AEAD is a ciphertext-length function  $\text{cl} : \mathbb{N} \rightarrow \mathbb{N}$ . The algorithms of AEAD operate as follows.*

- Via  $C \leftarrow \text{Enc}(sk, N, ad, M)$ , the deterministic encryption algorithm  $\text{Enc}$  on input the secret key  $sk \in \mathcal{SK}$ , a nonce  $N \in \mathcal{N}$ , associated data  $ad \in \mathcal{AD}$  and a message  $M \in \mathcal{M}$  produces a ciphertext  $C \in \{0, 1\}^{\text{cl}(|M|)}$ .
- Via  $M/\perp \leftarrow \text{Dec}(sk, N, ad, C)$ , the deterministic decryption algorithm  $\text{Dec}$  on input the secret key  $sk \in \mathcal{SK}$ , a nonce  $N \in \mathcal{N}$ , associated data  $ad \in \mathcal{AD}$  and a ciphertext  $C \in \{0, 1\}^*$  produces a message  $M \in \mathcal{M}$  or, to indicate failure, the special symbol  $\perp$ .

Correctness of a nonce-based AEAD scheme stipulates that  $\text{Dec}(sk, N, ad, \text{Enc}(sk, N, ad, M)) = M$  for all  $sk \in \mathcal{SK}$ ,  $N \in \mathcal{N}$ ,  $ad \in \mathcal{AD}$  and  $M \in \mathcal{M}$ .

We repeat the definition of AEAD ind\$-cpa security (confidentiality) and int-ctxt security (ciphertext integrity), as stated in [21]. Both notions are given in the multi-user setting [18].

**Definition 2.2** (AEAD confidentiality (ind\$-cpa)). *Let AEAD be a nonce-based AEAD scheme, and let game  $\mathbf{G}_{\text{AEAD}}^{\text{ind\$-cpa}}$  be defined as in Figure 2.1. We define the confidentiality (ind\$-cpa) advantage of an adversary  $\mathcal{A}$  against AEAD as:*

$$\text{Adv}_{\text{AEAD}}^{\text{ind\$-cpa}}(\mathcal{A}) = 2 \cdot \left| \Pr[\mathbf{G}_{\text{AEAD}}^{\text{ind\$-cpa}}(\mathcal{A}) \Rightarrow \text{true}] - \frac{1}{2} \right|.$$

**Definition 2.3** (AEAD integrity (int-ctxt)). *Let AEAD be a nonce-based AEAD scheme, and let game  $\mathbf{G}_{\text{AEAD}}^{\text{int-ctxt}}$  be defined as in Figure 2.1. We define the ciphertext integrity (int-ctxt) advantage of an adversary  $\mathcal{A}$  against AEAD as:*

$$\text{Adv}_{\text{AEAD}}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathbf{G}_{\text{AEAD}}^{\text{int-ctxt}}(\mathcal{A}) \Rightarrow \text{true}].$$

<p><b>Game <math>G_{\text{AEAD}}^{\text{ind\\$-cpa}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}; u \leftarrow 0</math></li> <li>2 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{NEW, Ro\\$}}()</math></li> <li>3 Return <math>b^* = b</math></li> </ol> <p><b>NEW():</b></p> <ol style="list-style-type: none"> <li>4 <math>u++; sk_u \leftarrow_{\\$} \mathcal{SK}</math></li> <li>5 <math>\mathcal{S}_{N,u} \leftarrow \emptyset</math></li> </ol> <p><b>Ro\$(i, N, ad, M)\$:</b></p> <ol style="list-style-type: none"> <li>6 If <math>N \in \mathcal{S}_{N,i}</math>:</li> <li>7     Return <math>\perp</math></li> <li>8 <math>\mathcal{S}_{N,i} \stackrel{\cup}{\leftarrow} \{N\}</math></li> <li>9 <math>C_0 \leftarrow_{\\$} \{0, 1\}^{\text{cl} M }</math></li> <li>10 <math>C_1 \leftarrow \text{Enc}(sk_i, N, ad, M)</math></li> <li>11 Return <math>C_b</math></li> </ol>	<p><b>Game <math>G_{\text{AEAD}}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>12 win <math>\leftarrow</math> false; <math>u \leftarrow 0</math></li> <li>13 <math>\mathcal{A}^{\text{NEW, ENC, DEC}}()</math></li> <li>14 Return win</li> </ol> <p><b>NEW():</b></p> <ol style="list-style-type: none"> <li>15 <math>u++; sk_u \leftarrow_{\\$} \mathcal{SK}</math></li> <li>16 <math>\mathcal{N}_u \leftarrow \emptyset; \mathcal{S}_{\text{NadC},u} \leftarrow \emptyset</math></li> </ol> <p><b>ENC(i, N, ad, M):</b></p> <ol style="list-style-type: none"> <li>17 If <math>N \in \mathcal{S}_{N,i}</math>:</li> <li>18     Return <math>\perp</math></li> <li>19 <math>\mathcal{S}_{N,i} \stackrel{\cup}{\leftarrow} \{N\}</math></li> <li>20 <math>C \leftarrow \text{Enc}(sk_i, N, ad, M)</math></li> <li>21 <math>\mathcal{S}_{\text{NadC},i} \stackrel{\cup}{\leftarrow} \{N, ad, C\}</math></li> <li>22 Return <math>C</math></li> </ol> <p><b>DEC(i, N, ad, C):</b></p> <ol style="list-style-type: none"> <li>23 <math>M \leftarrow \text{Dec}(sk_i, N, ad, C)</math></li> <li>24 If <math>M \neq \perp \wedge (N, ad, C) \notin \mathcal{S}_{\text{NadC},i}</math>:</li> <li>25     win <math>\leftarrow</math> true</li> <li>26 Return <math>M</math></li> </ol>
--	---

**Figure 2.1:** Game formalizations for confidentiality (ind\\$-cpa) and integrity (int-ctxt) of AEAD. Code with grey font prevents trivial attacks.

### 2.2.2 PRGs and PRFs

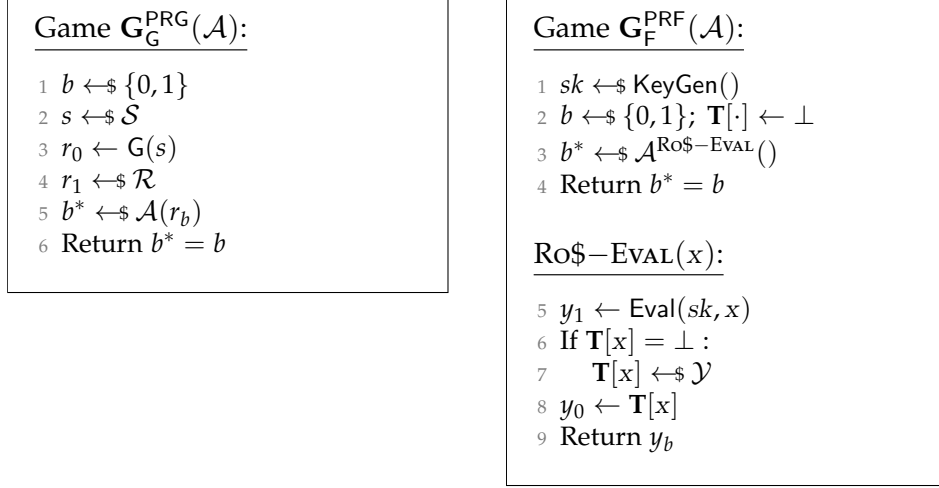
We present the definition of a pseudo-random generator (PRG), adapted from [22].

**Definition 2.4** (Pseudo-random generator). *A pseudo-random generator  $G : \mathcal{S} \rightarrow \mathcal{R}$  is an algorithm that on an input  $s \in \mathcal{S}$  (called a seed), deterministically computes an output  $r \in \mathcal{R}$ .  $\mathcal{S}$  and  $\mathcal{R}$  are finite.*

We provide the PRG security game in Figure 2.2.

**Definition 2.5** (PRG security). *We define the advantage of an adversary  $\mathcal{A}$  against the PRG security of  $G$  as:*

$$\text{Adv}_G^{\text{PRG}}(\mathcal{A}) = 2 \cdot \left| \Pr \left[ G_G^{\text{PRG}}(\mathcal{A}) \Rightarrow \text{true} \right] - \frac{1}{2} \right|.$$



**Figure 2.2:** Left: The PRG-security game in which the adversary has to distinguish between the real world ( $b = 0$ ), where it is given a real evaluation of  $G$ , and the random world ( $b = 1$ ), in which it is a random element from the output space. Right: the PRF-security game, in which the adversary can repeatedly query a challenge oracle and either gets truly random values ( $b = 0$ ) or pseudo-random values ( $b = 1$ ).

We also provide the definition of a pseudo-random function (PRF), in a non-standard way, to more seamlessly integrate with later definitions.

**Definition 2.6** (Pseudo-random function). A pseudo-random function,  $F = (\text{KeyGen}, \text{Eval})$ , is a pair of algorithms with three associated sets; the secret-key space  $\mathcal{SK}$ , the domain  $\mathcal{X}$ , and the range  $\mathcal{Y}$ .

- Via  $sk \leftarrow_{\$} \text{KeyGen}()$ , the probabilistic key generation algorithm  $\text{KeyGen}$ , taking no input, outputs the secret key  $sk \in \mathcal{SK}$ .
- Via  $y \leftarrow \text{Eval}(sk, x)$ , the function evaluation algorithm  $\text{Eval}$ , on input the secret key  $sk \in \mathcal{SK}$  and an element  $x \in \mathcal{X}$  outputs  $y \in \mathcal{Y}$ .

**Definition 2.7** (PRF security). We define the advantage of an adversary  $\mathcal{A}$  against the pseudorandomness PRF of  $F$  as

$$\text{Adv}_F^{\text{PRF}}(\mathcal{A}) = 2 \cdot \left| \Pr \left[ G_F^{\text{PRF}}(\mathcal{A}) \Rightarrow \text{true} \right] - \frac{1}{2} \right|.$$

### PRF from PRG

We present the well-known construction of a PRF from a PRG [23], and will refer to it as the Goldreich-Goldwasser-Micali (GGM) construction. Let  $G : \{0, 1\}^k \rightarrow \{0, 1\}^{2k}$ , a length-doubling PRG. We will use  $G_0, G_1$  to refer to



the left or the right half of  $G$ , respectively.

$$G(s) = G_0(s) || G_1(s)$$

We construct a PRF  $F : \{0,1\}^l \rightarrow \{0,1\}^k$ , taking as input  $x = (x_1, \dots, x_l) \in \{0,1\}^l$  as

$$F(sk, x) = G_{x_l}(G_{x_{l-1}}(\dots G_{x_2}(G_{x_1}(sk))))).$$

The function  $F$  chains evaluations of  $G$  together, the bits of the element  $x$  are used as an index to choose which half of  $G$  is used as a seed for the next evaluation of  $G$ . This can be imagined as a binary tree with nodes that have labels that are derived from an initial seed. The root of the tree has  $sk$  as the value, the left child of the root  $G_0(sk)$  and the right child of the root has value  $G_1(sk)$ . Nodes are labelled by the bitstring that leads to them.

### 2.2.3 Puncturable pseudo-random functions (PPRFs)

Puncturable pseudo-random functions (PPRFs) [19, 20, 24] are PRFs that can be punctured on elements of their domain, which makes it impossible to derive the corresponding elements in the range, thereby providing forward security for those values. We will present the syntax and security games for PPRFs and the construction of a PPRF from a PRG based on the GGM PRF construction [23].

We start by providing the definition of a PPRF as presented in [16].

**Definition 2.8** (PPRF). *A puncturable pseudo-random function,  $\text{PPRF} = (\text{KeyGen}, \text{Eval}, \text{Punc})$ , is a triple of algorithms with three associated sets; the secret-key space  $\mathcal{SK}$ , the domain  $\mathcal{X}$  and the range  $\mathcal{Y}$ .*

- Via  $sk \leftarrow \$ \text{KeyGen}()$ , the probabilistic key generation algorithm  $\text{KeyGen}$ , taking no input, outputs the secret key  $sk \in \mathcal{SK}$ .
- Via  $y / \perp \leftarrow \text{Eval}(sk, x)$ , the function evaluation algorithm  $\text{Eval}$ , on input the secret key  $sk$  and an element  $x \in \mathcal{X}$  outputs  $y \in \mathcal{Y}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{Punc}(sk, x)$ , the deterministic puncturing algorithm  $\text{Punc}$ , on input the secret key  $sk$  and an element  $x \in \mathcal{X}$  outputs an updated secret key  $sk' \in \mathcal{SK}$ .

For correctness, we require that for all  $sk \in \mathcal{SK}$  and all  $x, y \in \mathcal{X}$ :

- $\Pr[\text{Eval}(sk_0, x) \neq \perp \mid sk_0 \leftarrow \$ \text{KeyGen}()] = 1$ .
- If  $sk' \leftarrow \text{Punc}(sk, x)$  and  $y \neq x$ , then  $\text{Eval}(sk, y) = \text{Eval}(sk', y)$ .

<p><b>Game <math>\mathbf{G}_{\text{PPRF}}^{\text{fpr-ro}\\$}(\mathcal{A}), \mathbf{G}_{\text{PPRF}}^{\text{fpr-rro}\\$}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}; u \leftarrow 0; \mathbf{T}[\cdot, \cdot] \leftarrow \perp</math></li> <li>2 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{NEW, Ro}\\$, \text{EVAL}, \text{CORR, PUNC}}()</math></li> <li>3 Return <math>b^* = b</math></li> </ol> <p><b>NEW()</b></p> <ol style="list-style-type: none"> <li>4 <math>u++</math>; <math>sk_u \leftarrow_{\\$} \text{KeyGen}()</math></li> <li>5 <math>\mathcal{C}_u, \mathcal{E}_u, \mathcal{P}_u \leftarrow \emptyset</math></li> </ol>	<p><b>Ro\$\\$-EVAL(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>10 If <math>x \in \mathcal{E}_i</math> or <math>\text{corr}_i</math>: return <math>\perp</math></li> <li>11 <math>y_1 \leftarrow \text{Eval}(sk_i, x)</math></li> <li>12 If <math>y_1 = \perp</math>: return <math>\perp</math></li> <li>13 If <math>\mathbf{T}[i, x] = \perp</math>:</li> <li>14     <math>\mathbf{T}[i, x] \leftarrow_{\\$} \mathcal{Y}</math></li> <li>15 <math>y_0 \leftarrow \mathbf{T}[i, x]</math></li> <li>16 <math>\mathcal{C}_i \stackrel{\cup}{\leftarrow} \{x\}</math></li> <li>17 Return <math>y_b</math></li> </ol> <p><b>PUNC(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>18 <math>sk_i \leftarrow \text{Punc}(sk_i, x)</math></li> <li>19 <math>\mathcal{P}_i \stackrel{\cup}{\leftarrow} \{x\}</math></li> </ol> <p><b>CORR(<math>i</math>):</b></p> <ol style="list-style-type: none"> <li>20 If <math>\mathcal{C}_i \not\subseteq \mathcal{P}_i</math>:</li> <li>21     Return <math>\perp</math></li> <li>22 <math>\text{corr}_i \leftarrow \text{true}</math></li> <li>23 Return <math>sk_i</math></li> </ol>
<p><b>EVAL(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>6 If <math>x \in \mathcal{C}_i</math>: return <math>\perp</math></li> <li>7 <math>y \leftarrow \text{Eval}(sk_i, x)</math></li> <li>8 <math>\mathcal{E}_i \stackrel{\cup}{\leftarrow} \{x\}</math></li> <li>9 Return <math>y</math></li> </ol>	

**Figure 2.3:** Security game for PPRF with adversary  $\mathcal{A}$  having access to oracles Ro\$\\$-EVAL, PUNC, CORR in the game  $\mathbf{G}_{\text{PPRF}}^{\text{fpr-ro}\$}(\mathcal{A})$ . In the game  $\mathbf{G}_{\text{PPRF}}^{\text{fpr-rro}\$}(\mathcal{A})$ , it additionally has access to EVAL.

- If  $sk' \leftarrow \text{Punc}(sk, x)$ , then  $\text{Eval}(sk', x) = \perp$ .

The security games for a PPRF against an adversary are shown in Figure 2.3. They are in the multi-user setting [25] and capture the security notions fpr-ro\$\\$ and fpr-rro\$\\$ [16], which describe a combination of forward security and pseudo-randomness.

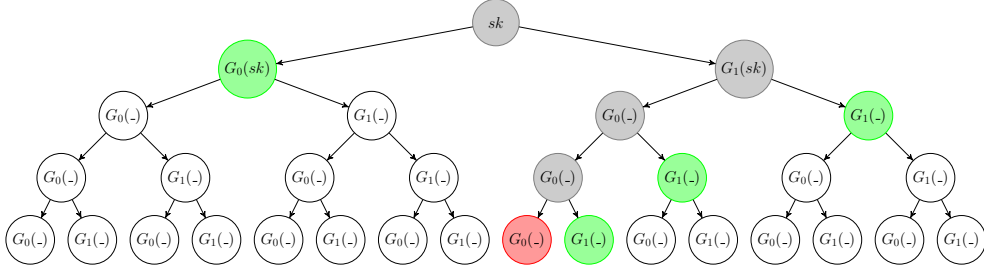
**Definition 2.9** (PPRF security (fpr-ro\$\\$, fpr-rro\$\\$)[16]). *We define the advantage of an adversary  $\mathcal{A}$  against the forward pseudorandomness  $X \in \{\text{fpr-ro}\$, \text{fpr-rro}\$ \}$  of PPRF as*

$$\text{Adv}_{\text{PPRF}}^X(\mathcal{A}) = 2 \cdot \left| \Pr \left[ \mathbf{G}_{\text{PPRF}}^X(\mathcal{A}) \Rightarrow \text{true} \right] - \frac{1}{2} \right|.$$

### PPRF from PRG

A possible construction of a PPRF can be achieved by adapting the GGM construction [23], that is shown above (Section 2.2.2).

To extend this instantiation of a PRF to match the syntax of a PPRF, a Punc algorithm is needed. Puncturing the GGM construction on  $x$  requires the



**Figure 2.4:** Example of a GGM tree of depth 4, punctured on element '1000'. The node labelled by the punctured element is shown in red, grey is used to indicate which nodes cannot be part of the key because of the puncture, the green nodes make up the updated secret key. To show how the values of the nodes are derived, the function used to generate the node value is shown inside the node. The symbol “-” denotes the use of the value of the parent node.

“removal” of the leaf node labelled  $x$ . We first note that the value of the node labelled  $x$  should not be recoverable after  $sk$  has been punctured on  $x$ . So, the updated secret key  $sk'$  cannot contain any predecessors of it (because then the  $x$  node could be derived from these). The rest of the tree should still be derivable, since it must still be possible to evaluate unpunctured elements. The secret key of a PPRF is therefore not the root secret  $sk$ , as it is for the PRF, but a set of nodes, each node being an unpunctured leaf node or the root of a subtree with only unpunctured leaf nodes. Observe that the nodes which make up the co-path from the root of a subtree to the leaf node define subtrees which make all nodes in the subtree except for the punctured node derivable. Consequently, to puncture  $x$ , the secret key  $sk$  is modified so that the node  $n_x$  defining the (unpunctured) subtree containing  $x$  is replaced by the co-path from  $n_x$  to  $x$ . As an example, Figure 2.4 shows a GGM PPRF tree of depth four, in which a single value is punctured. Before puncturing, the node containing “ $sk$ ” was the secret key. After puncturing, the key consists of the green nodes.

#### 2.2.4 Puncturable key wrapping

Key wrapping is a technique that allows safe storage and transport of keys under the protection of another (master) key. The primitive was introduced in [17], and extended in [16] to make the wrapped keys puncturable. Here, we repeat the syntax and security notions that are used to describe the PKW scheme given in [16].

**Definition 2.10** (PKW scheme). *A puncturable key-wrapping scheme consists of four algorithms  $\text{PKW} = (\text{Keygen}, \text{Wrap}, \text{Unwrap}, \text{Punc})$  with four associated sets: the secret-key space  $\mathcal{SK}$ , the tag space  $\mathcal{T}$ , the header space  $\mathcal{H}$  and the wrap-key space  $\mathcal{K}$ . Associated to the scheme is a ciphertext-length function  $\text{cl} : \mathcal{N} \rightarrow \mathcal{N}$ .*

- Via  $sk \leftarrow \text{KeyGen}()$ , the probabilistic key generation algorithm  $\text{KeyGen}$ , taking no input, outputs a secret key  $sk \in \mathcal{SK}$ .

- Via  $C/\perp \leftarrow \text{Wrap}(sk, T, H, K)$ , the deterministic wrapping algorithm  $\text{Wrap}$  on input a secret key  $sk \in \mathcal{SK}$ , a tag  $T \in \mathcal{T}$ , a header  $H \in \mathcal{H}$  and a key  $K \in \mathcal{K}$  outputs a ciphertext  $C \in \{0, 1\}^{\text{cl}(|K|)}$  or, to indicate failure,  $\perp$ .
- Via  $K/\perp \leftarrow \text{Unwrap}(sk, T, H, C)$ , the deterministic unwrapping algorithm  $\text{Unwrap}$  on input a secret key  $sk \in \mathcal{SK}$ , a tag  $T \in \mathcal{T}$ , a header  $H \in \mathcal{H}$  and a ciphertext  $C \in \{0, 1\}^*$  returns a key  $K \in \mathcal{K}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{Punc}(sk, T)$ , the deterministic puncturing algorithm  $\text{Punc}$  on input a secret key  $sk \in \mathcal{SK}$  and a tag  $T \in \mathcal{T}$  returns a potentially updated secret key  $sk' \in \mathcal{SK}$ .

### Confidentiality

Confidentiality is defined by the notions  $\text{find\$-cpa}$  and  $\text{find\$-rcpa}$ , both in the multi-user setting (Figure 2.5). We provide a brief overview of the oracles, the full explanation is found in [16]. The  $\text{find\$-cpa}$  notion is a form of indistinguishability from random bits ( $\text{ind\$-cpa}$ ) with forward security. It provides the adversary with a challenge oracle  $\text{RO\$-WRAP}$  to capture the indistinguishability, the  $\text{PUNC}$  oracle to perform punctures, and with a corruption oracle  $\text{CORR}$  to capture forward secrecy. For the  $\text{find\$-rcpa}$  notion, the adversary is additionally provided access to an oracle that returns real wrappings. We refer to the original work for the relations between these security notions.

**Definition 2.11** (PKW confidentiality). *Let PKW be a puncturable key-wrapping scheme. We define the advantage of an adversary  $\mathcal{A}$  against the forward indistinguishability  $X \in \{\text{find\$-cpa}, \text{find\$-rcpa}\}$  of PKW as*

$$\text{Adv}_{\text{PKW}}^X(\mathcal{A}) = 2 \left| \Pr[\mathbf{G}_{\text{PKW}}^X(\mathcal{A}) \Rightarrow \text{true}] - \frac{1}{2} \right|.$$

### Integrity

The security game definition of  $\text{int-ctxt}$  for a PKW scheme (Figure 2.6) is very closely related to the  $\text{int-ctxt}$  notion of AEAD. The objective of the adversary is to forge a valid ciphertext  $C$  (i.e. unwrapping does not result in  $\perp$ ) for a tag  $T$  and header  $H$ . Again, we refer to [16] for the details.

**Definition 2.12** (PKW integrity). *Let PKW be a puncturable key-wrapping scheme. We define the advantage of an adversary  $\mathcal{A}$  against the integrity of ciphertexts  $\text{int-ctxt}$  of PKW as*

$$\text{Adv}_{\text{PKW}}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathbf{G}_{\text{PKW}}^{\text{int-ctxt}}(\mathcal{A}) \Rightarrow \text{true}].$$

<p><b>Game <math>G_{\text{PKW}}^{\text{find\\$-cpa}}(\mathcal{A}), G_{\text{PKW}}^{\text{find\\$-rcpa}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}; u \leftarrow 0</math></li> <li>2 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{Ro\\$-WRAP, WRAP, PUNC, CORR, NEW}}()</math></li> <li>3 <b>Return</b> <math>b^* = b</math></li> </ol> <p><b>NEW()</b></p> <ol style="list-style-type: none"> <li>4 <math>u++</math></li> <li>5 <math>sk_u \leftarrow_{\\$} \text{KeyGen}()</math></li> <li>6 <math>\mathcal{S}_{PT,u}, \mathcal{S}_{\\$T,u}, \mathcal{S}_{T,u} \leftarrow \emptyset</math></li> <li>7 <math>\text{corr}_u \leftarrow \text{false}</math></li> </ol> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><b>WRAP(<math>i, T, H, K</math>):</b></p> <ol style="list-style-type: none"> <li>8 <b>If</b> <math>T \in \mathcal{S}_{T,i}</math> <b>then return</b> <math>\perp</math></li> <li>9 <math>C \leftarrow \text{Wrap}(sk_i, T, H, K)</math></li> <li>10 <math>\mathcal{S}_{T,i} \stackrel{\cup}{\leftarrow} \{T\}</math></li> <li>11 <b>Return</b> <math>y</math></li> </ol> </div>	<p><b>Ro\\$-WRAP(<math>i, T, H, K</math>):</b></p> <ol style="list-style-type: none"> <li>12 <b>If</b> <math>T \in \mathcal{S}_{T,i} \vee \text{corr}_i = \text{true}</math>:</li> <li>13     <b>Return</b> <math>\perp</math></li> <li>14 <math>C_1 \leftarrow \text{Wrap}(sk_i, T, H, K)</math></li> <li>15 <b>If</b> <math>C_1 = \perp</math>:</li> <li>16     <b>Return</b> <math>\perp</math></li> <li>17 <math>C_0 \leftarrow_{\\$} \{0, 1\}^{\text{cl}( K )}</math></li> <li>18 <math>\mathcal{S}_{\\$T,i} \stackrel{\cup}{\leftarrow} \{T\}; \mathcal{S}_{T,i} \stackrel{\cup}{\leftarrow} \{T\}</math></li> <li>19 <b>Return</b> <math>C_b</math></li> </ol> <p><b>CORR(<math>i</math>):</b></p> <ol style="list-style-type: none"> <li>20 <b>If</b> <math>\mathcal{S}_{PT,i} \not\subseteq \mathcal{S}_{\\$T,i}</math>:</li> <li>21     <b>Return</b> <math>\perp</math></li> <li>22 <math>\text{corr}_i \leftarrow \text{true}</math></li> <li>23 <b>Return</b> <math>sk_i</math></li> </ol> <p><b>PUNC(<math>T</math>):</b></p> <ol style="list-style-type: none"> <li>24 <math>sk_i \leftarrow \text{Punc}(sk_i, T)</math></li> <li>25 <math>\mathcal{S}_{PT,i} \stackrel{\cup}{\leftarrow} \{T\}</math></li> </ol>
---	--

**Figure 2.5:** Confidentiality and forward security (find\\$-cpa, find\\$-rcpa) games for puncturable key-wrapping scheme PKW. We omit the definition of find\\$-lcpa [16].

<p><b>Game <math>G_{\text{PKW}}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>\text{win} \leftarrow \text{false}; u \leftarrow 0</math></li> <li>2 <math>\mathcal{A}^{\text{NEW, WRAP, UNWRAP, PUNC}}()</math></li> <li>3 <b>Return</b> <math>\text{win}</math></li> </ol> <p><b>NEW()</b></p> <ol style="list-style-type: none"> <li>4 <math>u++; sk_u \leftarrow_{\\$} \text{KeyGen}()</math></li> <li>5 <math>\mathcal{S}_{THC,u}, \mathcal{S}_{T,u}, \mathcal{S}_{PT,u} \leftarrow \emptyset</math></li> </ol> <p><b>PUNC(<math>T</math>):</b></p> <ol style="list-style-type: none"> <li>6 <math>sk_i \leftarrow \text{Punc}(sk_i, T)</math></li> <li>7 <math>\mathcal{S}_{PT,i} \stackrel{\cup}{\leftarrow} \{T\}</math></li> </ol>	<p><b>WRAP(<math>i, T, H, K</math>):</b></p> <ol style="list-style-type: none"> <li>8 <b>If</b> <math>T \in \mathcal{S}_{T,i}</math> <b>then return</b> <math>\perp</math></li> <li>9 <math>C \leftarrow \text{Wrap}(sk_i, T, H, K)</math></li> <li>10 <b>If</b> <math>C = \perp</math> : <b>return</b> <math>\perp</math></li> <li>11 <math>\mathcal{S}_{THC,i} \stackrel{\cup}{\leftarrow} \{(T, H, C)\}; \mathcal{S}_{T,i} \stackrel{\cup}{\leftarrow} \{T\}</math></li> <li>12 <b>Return</b> <math>C</math></li> </ol> <p><b>UNWRAP(<math>i, T, H, C</math>):</b></p> <ol style="list-style-type: none"> <li>13 <math>K \leftarrow \text{Unwrap}(sk_i, T, H, C)</math></li> <li>14 <b>If</b> <math>K \neq \perp \wedge ((T, H, C) \notin \mathcal{S}_{THC,i} \vee T \in \mathcal{S}_{PT,i})</math>:</li> <li>15     <math>\text{win} \leftarrow \text{true}</math></li> <li>16 <b>Return</b> <math>K</math></li> </ol>
---	--

**Figure 2.6:** Ciphertext integrity (int-ctxt) game for puncturable key-wrapping scheme PKW.

<p><u>PKW[PPRF, AEAD]:</u></p> <p><u>PKW.KeyGen():</u></p> <ol style="list-style-type: none"> <li>1 Return PPRF.KeyGen()</li> </ol> <p><u>PKW.Wrap(<math>sk_p, T, H, K</math>):</u></p> <ol style="list-style-type: none"> <li>2 <math>sk_a \leftarrow \text{PPRF.Eval}(sk_p, T)</math></li> <li>3 <math>C \leftarrow \text{AEAD.Enc}(sk_a, N_0, H, K)</math></li> <li>4 Return <math>C</math></li> </ol>	<p><u>PKW.Unwrap(<math>sk_p, T, H, C</math>):</u></p> <ol style="list-style-type: none"> <li>5 <math>sk_a \leftarrow \text{PPRF.Eval}(sk_p, T)</math></li> <li>6 <math>K \leftarrow \text{AEAD.Dec}(sk_a, N_0, H, C)</math></li> <li>7 Return <math>K</math></li> </ol> <p><u>PKW.Punc(<math>sk_p, T</math>):</u></p> <ol style="list-style-type: none"> <li>8 <math>sk'_p \leftarrow \text{PPRF.Punc}(sk_p, T)</math></li> <li>9 Return <math>sk'_p</math></li> </ol>
---	--

**Figure 2.7:** Construction of PKW from PPRF and AEAD. The constant  $N_0$  is set to the all-zero nonce.

### Construction from AEAD and PPRF

The construction of PKW from AEAD and PPRF proposed in [16] is shown in Figure 2.7. In Wrap, it uses the PPRF to derive key encryption keys (KEKs) for the given tag and uses the KEK to encrypt the key passed as argument with the AEAD scheme. For Unwrap, again the PPRF is evaluated for the given tag, and the ciphertext is decrypted using the derived KEK. So, the tag space of the PKW scheme is equal to the domain of the PPRF. The puncturing operation is delegated to the PPRF.

We give some intuition on why it is secure:

**Confidentiality** Because the keys given to wrap are AEAD encrypted under a key derived by the PPRF, puncturing the PPRF on the tag that was used for wrapping makes the key undervivable and thus gives the forward security needed for find\$-cpa. The indistinguishability from random bits is provided by AEAD.

**Integrity** Similarly to confidentiality, ciphertext integrity (int-ctxt) is provided by the AEAD scheme.

---

## Protected file storage

---

The primary objective of this project is the development of a forward-secure file storage system that relies on puncturable key-wrapping to provide forward security for deleted files. To achieve this, a library produced in a previous project offering a puncturable key-wrapping implementation [26], as described in [16], is used and further enhanced. Puncturable key-wrapping is used to wrap individual data encryption keys (DEKs) for each file stored in the system. To delete a file (an operation we call “shred”), the corresponding DEK is punctured.

We will introduce the syntax and security notions used to describe a protected file storage (PFS) scheme as shown in [16] and present a new optimization which requires an update to the syntax. For both versions, we also present constructions and implementations and evaluate their performance.

### 3.1 PFS (original definition)

A protected file storage (PFS) is defined by the following syntax, as stated in [16].

**Definition 3.1** (PFS scheme). *A protected file storage scheme  $\text{PFS} = (\text{Setup}, \text{EncFile}, \text{DecFile}, \text{ShredFile}, \text{RotKey})$  is a 5-tuple of algorithms with four associated sets; the secret key space  $\mathcal{SK}$ , the file space  $\mathcal{F}$ , the file identifier space  $\mathcal{I}$  and the header space  $\mathcal{H}$ . Associated to the PFS is a ciphertext-length function  $\text{cl} : \mathbb{N} \rightarrow \mathbb{N}$ .*

- Via  $sk \leftarrow \$ \text{Setup}()$ , the probabilistic setup algorithm  $\text{Setup}$ , taking no input, produces a secret key  $sk \in \mathcal{SK}$ .
- Via  $(id, h, C) / \perp \leftarrow \$ \text{EncFile}(sk, F)$ , the randomized file encryption algorithm  $\text{EncFile}$  on input the secret key  $sk \in \mathcal{SK}$  and a plaintext file  $F \in \mathcal{F}$  produces a file identifier  $id \in \mathcal{I}$ , a header  $h \in \mathcal{H}$  and a ciphertext  $C \in \{0, 1\}^{\text{cl}(|F|)}$  or, to indicate failure,  $\perp$ .

- Via  $F/\perp \leftarrow \text{DecFile}(sk, id, h, C)$ , the deterministic file decryption algorithm  $\text{DecFile}$  on input the key  $sk \in \mathcal{SK}$ , a file identifier  $id \in \mathcal{I}$ , a header  $h \in \mathcal{H}$ , and a ciphertext  $C \in \{0,1\}^*$  returns a file plaintext  $F \in \mathcal{F}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{ShredFile}(sk, id)$ , the deterministic shredding algorithm  $\text{ShredFile}$  on input the secret key  $sk \in \mathcal{SK}$  and a file identifier  $id \in \mathcal{I}$  returns the updated secret key  $sk' \in \mathcal{SK}$ .
- Via  $(sk', (h'_1, \dots, h'_l)) / (sk', \perp) \leftarrow \text{RotKey}(sk, ((id_1, h_1), \dots, (id_l, h_l)))$ , the randomized key-rotation algorithm  $\text{RotKey}$  on input the secret key  $sk \in \mathcal{SK}$  and a list containing file identifier-header pairs  $((id_1, h_1), \dots, (id_l, h_l)) \in (\mathcal{I} \times \mathcal{H})^*$  returns the potentially updated secret key  $sk' \in \mathcal{SK}$  and a list of updated headers  $(h'_1, \dots, h'_l) \in \mathcal{H}^*$  or, to indicate failure,  $\perp$ .

We restate the definition of correctness from [16].

**Correctness** For correctness of a PFS scheme, we require that an encrypted file can always be decrypted, unless it has been shredded. This is conditioned on the fact that the corresponding header is updated in key rotations, since omitting the header during key rotation effectively deletes the file.

### 3.1.1 Confidentiality and integrity

The security notion for confidentiality is  $\text{find\$-rcpa}$  [16] (Figure 3.1), which captures both forward security and indistinguishability from random bits of the tuple  $(id, h, C)$  under real and real-or-random chosen plaintext attack (the adversary gets access to both a real-or-random encryption oracle and a real encryption oracle). Integrity is modelled with the notion  $\text{int-ctxt}$  (Figure 3.2), in which the adversary wins if it manages to forge a ciphertext that decrypts without error. For the details, we refer to [16].

**Definition 3.2** (PFS confidentiality [16]). *Let PFS be a protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the  $\text{find\$-rcpa}$  security of PFS as*

$$\mathbf{Adv}_{\text{PFS}}^{\text{find\$-rcpa}}(\mathcal{A}) = 2 \left| \Pr[\mathbf{G}_{\text{PFS}}^{\text{find\$-rcpa}}(\mathcal{A}) \Rightarrow \text{true}] - \frac{1}{2} \right|.$$

**Definition 3.3** (PFS integrity [16]). *Let PFS be a protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the  $\text{int-ctxt}$  security of PFS as*

$$\mathbf{Adv}_{\text{PFS}}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathbf{G}_{\text{PFS}}^{\text{int-ctxt}}(\mathcal{A}) \Rightarrow \text{true}].$$

We will give an overview of the oracles to which the adversary has access for the  $\text{find\$-rcpa}$  security game. It can call the  $\text{Ro\$-ENC}$  (“challenge”) or



<p><b>Game <math>G_{\text{PFS}}^{\text{find}\\$-\text{rcpa}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}; sk \leftarrow_{\\$} \text{Setup}()</math></li> <li>2 <math>L_{\text{Ro}\\$} \leftarrow (); L_{\text{Enc}} \leftarrow ()</math></li> <li>3 <math>\mathcal{S}_{\\$id} \leftarrow \emptyset; \text{corr} \leftarrow \text{false}</math></li> <li>4 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{Ro}\\$-\text{ENC}, \text{ENC}, \text{SHRED}, \text{CORR}, \text{ROTKEY}}()</math></li> <li>5 <b>Return</b> <math>b^* = b</math></li> </ol> <p><b><u>Ro\$\text{-ENC}(F)</u>:</b></p> <ol style="list-style-type: none"> <li>6 <b>If</b> <math>\text{corr} = \text{true}</math> <b>then return</b> <math>\perp</math></li> <li>7 <math>(id_1, h_1, C_1) \leftarrow_{\\$} \text{EncFile}(sk, F)</math></li> <li>8 <b>If</b> <math>(id_1, h_1, C_1) = \perp</math>:</li> <li>9     <b>Return</b> <math>\perp</math></li> <li>10 <math>id_0 \leftarrow_{\\$} \mathcal{I}; h_0 \leftarrow_{\\$} \mathcal{H}</math></li> <li>11 <math>C_0 \leftarrow_{\\$} \{0, 1\}^{\text{cl}( F )}</math></li> <li>12 <math>L_{\text{Ro}\\$} += (id_b, h_b)</math></li> <li>13 <math>\mathcal{S}_{\\$id} \stackrel{\cup}{\leftarrow} \{id_b\}</math></li> <li>14 <b>Return</b> <math>(id_b, h_b, C_b)</math></li> </ol> <p><b><u>ENC(F)</u>:</b></p> <ol style="list-style-type: none"> <li>15 <math>(id, h, C) \leftarrow_{\\$} \text{EncFile}(sk, F)</math></li> <li>16 <math>L_{\text{Enc}} += (id, h)</math></li> <li>17 <b>Return</b> <math>(id, h, C)</math></li> </ol>	<p><b><u>SHRED(id)</u>:</b></p> <ol style="list-style-type: none"> <li>18 <math>sk \leftarrow \text{ShredFile}(sk, id)</math></li> <li>19 <math>L_{\text{Ro}\\$} -= (id, *); L_{\text{Enc}} -= (id, *);</math></li> <li>20 <math>\mathcal{S}_{\\$id} \leftarrow \mathcal{S}_{\\$id} \setminus \{id\}</math></li> </ol> <p><b><u>ROTKEY()</u>:</b></p> <ol style="list-style-type: none"> <li>21 <math>l_{\\$} \leftarrow  L_{\text{Ro}\\$} ; l_r \leftarrow  L_{\text{Enc}} </math></li> <li>22 <math>((id_1, h_1), \dots, (id_{l_{\\$}}, h_{l_{\\$}})) \leftarrow L_{\text{Ro}\\$}</math></li> <li>23 <math>((id_{l_{\\$}+1}, h_{l_{\\$}+1}), \dots, (id_{l_{\\$}+l_r}, h_{l_{\\$}+l_r})) \leftarrow L_{\text{Enc}}</math></li> <li>24 <b>If</b> <math>b = 0</math>:</li> <li>25     <b>For</b> <math>i \leftarrow 1</math> <b>to</b> <math>l_{\\$}</math> <b>do</b> <math>h'_i \leftarrow_{\\$} \mathcal{H}</math></li> <li>26     <math>(sk, (h_{l_{\\$}+1}, \dots, h_{l_{\\$}+l_r})) \leftarrow_{\\$} \text{RotKey}(sk, L_{\text{Enc}})</math></li> <li>27     <b>If</b> <math>(h_{l_{\\$}+1}, \dots, h_{l_{\\$}+l_r}) = \perp</math> <b>then return</b> <math>\perp</math></li> <li>28 <b>If</b> <math>b = 1</math>:</li> <li>29     <math>(sk, (h_1, \dots, h_{l_{\\$}+l_r})) \leftarrow_{\\$} \text{RotKey}(sk, L_{\text{Ro}\\$}    L_{\text{Enc}})</math></li> <li>30     <b>If</b> <math>(h_1, \dots, h_{l_{\\$}+l_r}) = \perp</math> <b>then return</b> <math>\perp</math></li> <li>31 <math>L_{\text{Ro}\\$} \leftarrow ((id_1, h_1), \dots, (id_{l_{\\$}}, h_{l_{\\$}}))</math></li> <li>32 <math>L_{\text{Enc}} \leftarrow ((id_{l_{\\$}+1}, h_{l_{\\$}+1}), \dots, (id_{l_{\\$}+l_r}, h_{l_{\\$}+l_r}))</math></li> <li>33 <math>\text{corr} \leftarrow \text{false}</math></li> <li>34 <b>Return</b> <math>L_{\text{Ro}\\$}    L_{\text{Enc}}</math></li> </ol> <p><b><u>CORR()</u>:</b></p> <ol style="list-style-type: none"> <li>35 <b>If</b> <math>\mathcal{S}_{\\$id} \neq \emptyset</math> <b>then return</b> <math>\perp</math></li> <li>36 <math>\text{corr} \leftarrow \text{true}</math></li> <li>37 <b>Return</b> <math>sk</math></li> </ol>
--	--

**Figure 3.1:** Confidentiality and forward security (find\$\text{-rcpa}\$) game for protected file storage scheme PFS. Lists  $L_{\text{Ro}\$}$  and  $L_{\text{Enc}}$  keep track of headers currently in the system for the sake of key rotation. We write  $M -= (id, *)$  to denote removing a any pair from a list  $M$  in which the first item is  $id$ .

ENC (“encryption”) oracle to receive a real-or-random or a real encryption of a file. By calling the SHRED oracle, it can forward-securely delete a previously encrypted file. The adversary can gain access to the key by calling the corruption oracle CORR, but only after it has deleted all files encrypted with the challenge oracle. Once the key has been corrupted, the challenge oracle cannot be called until the key rotation oracle, ROTKEY (which replaces the secret key with a new secret key) has been called. To win the game, the adversary has to distinguish whether the challenge oracle provides real encryptions or random values.

In the int-ctxt game, the adversary has access to an encryption oracle, ENC, a decryption oracle, DEC, a shred oracle, SHRED, and a key rotation oracle,

<p><b>Game <math>\mathbf{G}_{\text{PFS}}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>\text{win} \leftarrow \text{false}; sk \leftarrow \\$ \text{Setup}()</math></li> <li>2 <math>\mathcal{S} \leftarrow \emptyset</math></li> <li>3 <math>\mathcal{A}^{\text{ENC,DEC,SHRED,ROTKEY}}()</math></li> <li>4 <b>Return</b> win</li> </ol> <p><b>ENC(<math>F</math>):</b></p> <ol style="list-style-type: none"> <li>5 <math>(id, h, C) \leftarrow \\$ \text{EncFile}(sk, F)</math></li> <li>6 <math>\mathcal{S} \stackrel{\cup}{\leftarrow} \{(id, h, C)\}</math></li> <li>7 <b>Return</b> <math>(id, h, C)</math></li> </ol> <p><b>SHRED(<math>id</math>):</b></p> <ol style="list-style-type: none"> <li>8 <math>sk \leftarrow \text{ShredFile}(sk, id)</math></li> <li>9 <math>\mathcal{S} \leftarrow \mathcal{S} \setminus \{(id, *, *)\}</math></li> </ol>	<p><b>DEC(<math>id, h, C</math>):</b></p> <ol style="list-style-type: none"> <li>10 <math>F \leftarrow \text{DecFile}(sk, id, h, C)</math></li> <li>11 <b>If</b> <math>(id, h, C) \notin \mathcal{S}</math> <b>and</b> <math>F \neq \perp</math>:</li> <li>12     <math>\text{win} \leftarrow \text{true}</math></li> <li>13 <b>Return</b> <math>F</math></li> </ol> <p><b>ROTKEY(<math>((id_1, h_1), \dots, (id_l, h_l))</math>):</b></p> <ol style="list-style-type: none"> <li>14 <math>(sk, (h'_1, \dots, h'_l)) \leftarrow \\$ \text{RotKey}(sk, ((id_1, h_1), \dots, (id_l, h_l)))</math></li> <li>15 <b>If</b> <math>(h'_1, \dots, h'_l) = \perp</math></li> <li>16     <b>Return</b> <math>\perp</math></li> <li>17 <math>\mathcal{S}_{\text{new}} \leftarrow \emptyset</math></li> <li>18 <b>For</b> <math>(id, h, C) \in \mathcal{S}</math> <b>do</b>:</li> <li>19     <b>If</b> <math>\exists i \in \{1, \dots, l\}</math>, s.t.               <math>(id, h) = (id_i, h_i)</math></li> <li>20         <math>\mathcal{S}_{\text{new}} \stackrel{\cup}{\leftarrow} \{(id, h'_i, C)\}</math></li> <li>21 <math>\mathcal{S} \leftarrow \mathcal{S}_{\text{new}}</math></li> <li>22 <b>Return</b> <math>(h'_1, \dots, h'_l)</math></li> </ol>
---	---

**Figure 3.2:** Ciphertext integrity (int-ctxt) game for protected file storage scheme PFS as presented in [16]. The notation  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(id, *, *)\}$  designates the removal of all tuples of which the first element is  $id$  from the set  $\mathcal{S}$ .

ROTKEY. To win the game, the adversary has to pass a tuple to the decryption oracle that was not produced by the encryption oracle, but still decrypts without error.

### Adversarial cloud service

In the security games (Figures 3.1 and 3.2), the adversary has extensive control over the system. It can encrypt files, shred files, rotate the key, and for the int-ctxt game, also decrypt files. A realistic adversary in the cloud storage setting is a malicious cloud storage service, which, while providing excellent availability for stored data, tries to glean as much insight as possible on the data that it stores. Reasons for this could range from hackers gaining access to the cloud service [1] to government-sanctioned dragnet operations [27] or even forged search warrants [2]. A malicious cloud service has significantly fewer capabilities than the adversary in the security games. It only sees ciphertexts and headers, and cannot issue any commands to the local client (it cannot tell the client to shred a file, for instance). The security notions are therefore stronger than what is needed for a threat model in which there is a malicious cloud service, and encode a strong form of privacy for the stored ciphertexts. However, it means that there is a gap between the threat model for a forward-secure cloud storage and the definition

of PFS security, of which the exploration is out of scope for this thesis.

### 3.1.2 Construction from PKW and AEAD

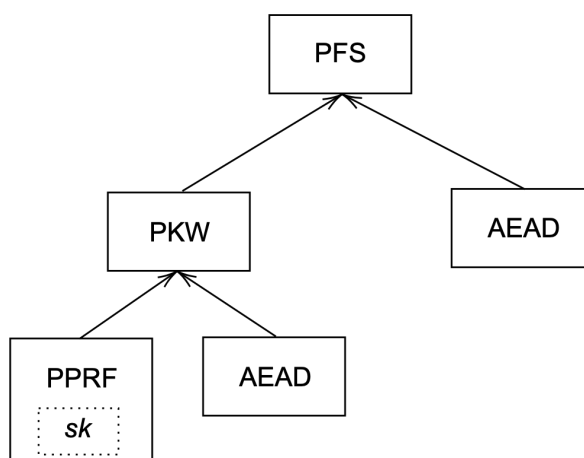
For completeness, here we repeat the construction of PFS from the composition of PKW and AEAD (Figure 3.3), as shown in [16]. In short, the PKW scheme is used to protect the DEKs, which are used in the AEAD scheme to encrypt and decrypt the files. The shredding operation is delegated to the PKW, where the PKW secret key for the file in question is punctured on the tag corresponding to the file. This updates the secret key  $sk$ , so that future encryptions or decryptions are not possible. We note that the random choice of a file identifier (line 8 of EncFile in Figure 3.3) may result in a collision (a file identifier being reused to encrypt a new file). The games do not specify this, the system must disallow it, however.

<p><u>Setup()</u> :</p> <ol style="list-style-type: none"> <li>1 Return PKW.KeyGen()</li> </ol> <p><u>DecFile(<math>sk, id, h, N    C</math>):</u></p> <ol style="list-style-type: none"> <li>2 <math>K \leftarrow</math> PKW.Unwrap(<math>sk, id, \epsilon, h</math>)</li> <li>3 If <math>K = \perp</math> :</li> <li>4   Return <math>\perp</math></li> <li>5 <math>F \leftarrow</math> AEAD.Dec(<math>K, N, \epsilon, C</math>)</li> <li>6 Return <math>F</math></li> </ol> <p><u>ShredFile(<math>sk, id</math>):</u></p> <ol style="list-style-type: none"> <li>7 <math>sk' \leftarrow</math> PKW.Punc(<math>sk, id</math>)</li> <li>8 Return <math>sk'</math></li> </ol>	<p><u>EncFile(<math>sk, F</math>):</u></p> <ol style="list-style-type: none"> <li>9 <math>K \leftarrow</math> <math>\{0, 1\}^k</math>; <math>id \leftarrow</math> <math>\{0, 1\}^t</math></li> <li>10 <math>h \leftarrow</math> PKW.Wrap(<math>sk, id, \epsilon, K</math>)</li> <li>11 If <math>h = \perp</math> :</li> <li>12   Return <math>\perp</math></li> <li>13 <math>N \leftarrow</math> <math>\{0, 1\}^n</math></li> <li>14 <math>C \leftarrow</math> AEAD.Enc(<math>K, N, \epsilon, F</math>)</li> <li>15 Return (<math>id, h, N    C</math>)</li> </ol> <p><u>RotKey(<math>sk_{old}, ((id_1, h_1), \dots, (id_l, h_l))</math>):</u></p> <ol style="list-style-type: none"> <li>16 <math>sk_{new} \leftarrow</math> PKW.KeyGen()</li> <li>17 For <math>i \leftarrow 1</math> to <math>l</math> do:</li> <li>18   <math>K_i \leftarrow</math> PKW.Unwrap(<math>sk_{old}, id_i, \epsilon, h_i</math>)</li> <li>19   <math>h'_i \leftarrow</math> PKW.Wrap(<math>sk_{new}, id_i, \epsilon, K_i</math>)</li> <li>20   If <math>h'_i = \perp</math> then return (<math>sk_{old}, \perp</math>)</li> <li>21 Return (<math>sk_{new}, ((id_1, h'_1), \dots, (id_l, h'_l))</math>)</li> </ol>
---	--

**Figure 3.3:** PFS construction achieving find $\$$ -rcpa security from composition of PKW scheme PKW and AEAD scheme AEAD as shown in [16]. The key-wrap space of the PKW scheme is  $\{0, 1\}^k$ , the tag space is  $\{0, 1\}^t$ . The AEAD scheme has nonce space  $\{0, 1\}^n$  and key space  $\{0, 1\}^k$ . The instantiated PFS scheme therefore has file identifier space  $\mathcal{I} = \{0, 1\}^t$ , header space  $\mathcal{H} = \{0, 1\}^{\text{PKW.cl}(k)}$  and an associated ciphertext length of  $\text{PFS.cl}(|F|) = \text{AEAD.cl}(|F|) + n$ .

### 3.1.3 Implementation

PFS is implemented in C++ using a PKW library [26] and an AEAD implementation (AES-GCM) from CryptoPP [28], based on the construction in Figure 3.3. We show an overview of how the primitives are combined in



**Figure 3.4:** Diagram showing how the primitives PPRF, PKW, and AEAD are combined to instantiate PFS. The secret key  $sk$  of the PFS is really the secret key of the PPRF.

Figure 3.4. The storage of encrypted files and headers is done with Google Cloud Storage (GCS), because of ease of integration (a C++ library is provided by Google [29]) and a free evaluation period of three months. To map the file identifier (randomly sampled, see line 8 of Figure 3.3) assigned by PFS during encryption to the corresponding file name, a lookup table in the form of an associative map is used. The PKW used to instantiate PFS uses tags and keys of bitlength 256. The tag length is chosen so that a tag collision will only become a problem after  $\approx 2^{128}$  files are added (the birthday bound applies). This number is large enough to support as many files as needed in any file system and is selected as an upper bound. The header and the file produced by the encryption operation are stored in the cloud under the identifier and an appropriate suffix (the encrypted file is stored with  $\langle id \rangle.f$ , the header with  $\langle id \rangle.h$ ). The ciphertext and header are stored separately because for key rotation, only the headers are needed, and downloading both the ciphertext and the header would be inefficient.

Because the forward-security of the shred operation relies entirely on the PKW scheme and the secret key, which is local, the ciphertext and header located on the cloud do not have to be immediately deleted. They actually do not need to be deleted at all, although the accumulation of shredded (and therefore irrecoverable) files may become an issue in terms of the storage space they occupy. We will present one way to remove shredded files from the cloud somewhat asynchronously in Section 3.1.4, where files are queued for cloud deletion until a given threshold, after which they are all deleted. Another approach could be a periodic clean-up job, which handles actual cloud deletion at some specified interval and works its way through a queue of files marked to be deleted on the cloud.

Local file names and the corresponding assigned identifiers used to refer to the file when stored on the cloud have to be stored somehow as a lookup table, since the PFS scheme does not keep track of this metadata. Without a lookup functionality, there is no way to quickly access a stored file. During the operation of the client, the information can be held in memory, however, between sessions, this lookup table has to be persisted.

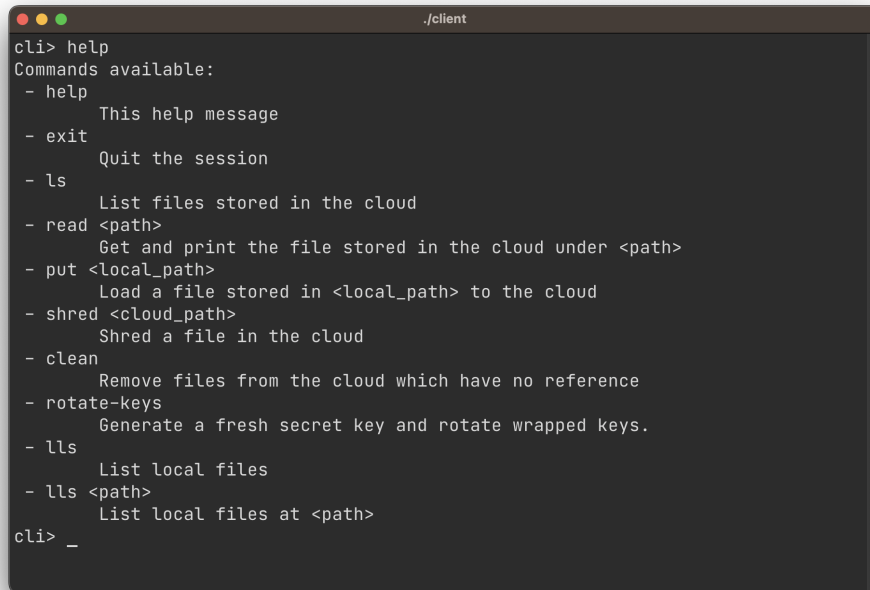
One option is to store it on the client's local storage; however, since it could grow to large sizes, it may be preferable to offload storage. So, instead of storing the lookup table locally, the cloud service can be used to store it (encrypted). To mitigate forward security pitfalls (when using a static key), key ratcheting is used. The lookup table is encrypted under a key  $K_i$  on completion of a session (an initial key  $K_0$  is randomly sampled for the first session). At the start of the next session, the encrypted lookup table is downloaded and decrypted under  $K_i$  and  $K_i$  is ratcheted forward to  $K_{i+1}$ . This process is described in [30] and appeared as part of the "double ratchet" of the Signal protocol [31]. We use the term "ratchet" here only in the symmetric sense, implemented as a key derivation function (HKDF.DeriveKey from CryptoPP [28]).

These aspects (encryption of lookup-table, use of file identifiers as identifiers for cloud storage) are not modelled in a security game, but rather are practical choices. Their formal security analysis is out of scope for this thesis but merits further investigation.

To access the system to inspect stored files, an interactive command line client was implemented, with commands inspired by the file transfer protocol (FTP) (Figure 3.5).

### 3.1.4 Evaluation

To evaluate the performance of the PFS implementation, benchmarks were performed on a laptop computer with an 8-core 2.3 GHz Intel i9 processor, 16 GB of RAM and a wired 1 Gbps (up / down) Internet connection. A first set of benchmarks was run with file storage on GCS, to assess the overhead of the local cryptographic operations in relation to the time used for network operations. In a second step, the implementation was tested by replaying file access patterns extracted from GitHub repository histories. Previous work [26] found that punctures on random PPRF elements induce a large secret key size. Since the PFS secret key is actually a PPRF secret key (Figure 3.4), a focus is placed on investigating the size of the key resulting from file access patterns and drawing lessons from the benchmarks to improve the scheme.



```
cli> help
Commands available:
- help      This help message
- exit      Quit the session
- ls        List files stored in the cloud
- read <path>
            Get and print the file stored in the cloud under <path>
- put <local_path>
            Load a file stored in <local_path> to the cloud
- shred <cloud_path>
            Shred a file in the cloud
- clean     Remove files from the cloud which have no reference
- rotate-keys
            Generate a fresh secret key and rotate wrapped keys.
- lls      List local files
- lls <path>
            List local files at <path>
cli> _
```

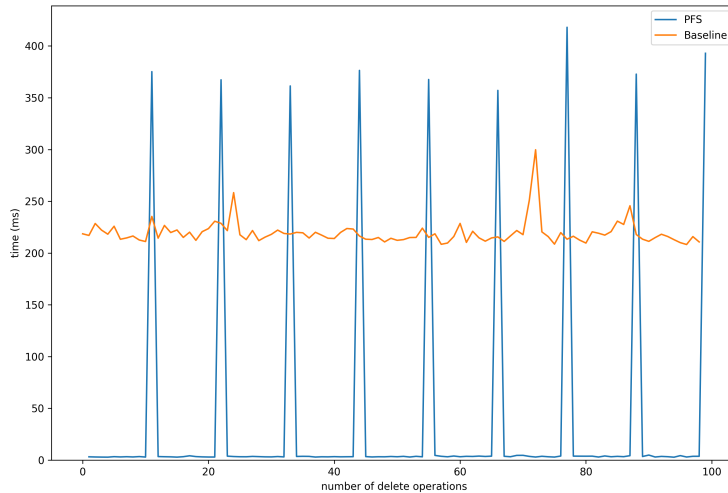
**Figure 3.5:** *The interactive client, with commands inspired by FTP. Here, the help menu is shown.*

#### Encrypting files

The time to encrypt a file and then upload it to the cloud was measured to be about 80 milliseconds, uploading the same file in plaintext took on average about 70 milliseconds. The time to evaluate the PPRF to obtain the correct key-wrapping key was about 2 milliseconds, so the largest overhead is the actual (AEAD) encryption of the file to produce the ciphertext and the data encryption key to produce the header.

#### Shredding files

The setup for this benchmark is simple: files are put in the system and then the time to delete them is measured. As stated earlier, since the PFS scheme provides forward security for shredded files and the cryptographic operations providing the forward secrecy are handled locally, there is no need to physically delete a shredded file on the cloud storage. Here, we handle the deletion after a given number of files are queued for deletion. In Figure 3.6, the spikes in time consumption that this approach produces are clearly visible. We stress that the PFS system can be much more efficient in the time usage of the deletion (shred) operation. Indeed, the machine running the system need not even be online to perform the operation, whereas, in a regular cloud storage setting, the client must wait for the cloud to confirm the



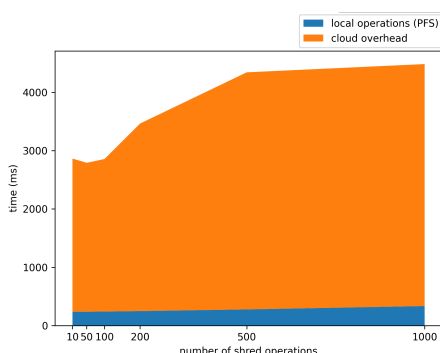
**Figure 3.6:** Time used to shred (PFS, blue) or delete (regular cloud storage, orange) files. The spikes in the PFS system stem from a set of files being deleted from the cloud storage at a specified threshold (in this instance, when the deletion queue contains more than 10 files).

deletion of a specified file.

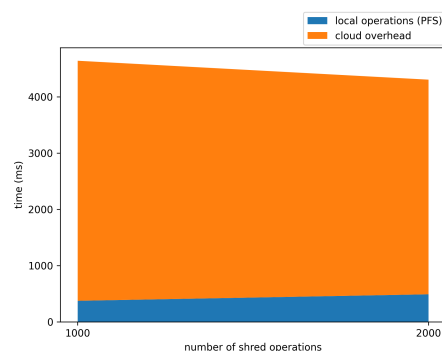
### Rotating keys

Clearly, the time consumption of key rotation is in a linear relationship with the number of stored files. Since all the headers must be downloaded, unwrapped, wrapped under a new key and reuploaded, the goal was to verify that the system behaves as expected, i.e. the time consumption for a fixed number of operations stays reasonably constant. To do so, a baseline of 100 files was uploaded. The key rotation would then affect the headers corresponding to these 100 files. To measure the overhead of having punctures present, batches of files were repeatedly added and shredded in increasing numbers (to cause a change in the secret key). The time was measured only over the key rotation for the 100 base files (which were not touched by the put and shred operations). While the measurements are rather noisy, there is a trend: key rotation after many put/shred operations takes longer than key rotation before. Figures 3.7 and 3.8 show the results of the evaluation. The time used for local operations is shown in blue and the cloud communication in orange. Time consumption by the PKW scheme to unwrap and wrap the headers does not increase much for an increasing number of shredded files (the blue part stays flat in Figure 3.7). So, it is puzzling to see this clear increase in overall time usage. The observed increase is introduced by GCS, possibly because of indexing issues. Because many files are added

### 3. PROTECTED FILE STORAGE



**Figure 3.7:** Time used for key rotation with 100 files present in the system, time measured after putting and then shredding files in batches of 10, 50, 100, 200, 500 and 1000. The cryptographic operations of PFS are shown in blue, network time is shown in orange.



**Figure 3.8:** Time used for key rotation with 100 files present in the system, time measured after putting and then shredding files in batches of 1000, 2000. The cryptographic operations of PFS are shown in blue, network time is shown in orange.

and deleted at a time, the lookup of the file headers (which are retrieved for key rotation) may take an increasingly long time, until better indexes are built. Some indication of this is given by looking at the same benchmark, but where the first batch deletion size is already large (Figure 3.8): the time for key rotation for the 100 files is greater after 1000 files are shredded than after 2000 files are shredded. The random identifiers used to store the files and headers possibly contribute to inconsistent lookup times. However, what this benchmark clearly shows is that network effects dominate key rotation performance.

#### GitHub commit histories

To get an impression of realistic file access patterns in a file storage system, some public GitHub [32] repositories [33, 34, 35, 36] are used. Git records the history of a directory through a chain of commits, each of which documents the files which were added, modified, or deleted. This history is extracted and the actions are sequentially replayed in the PFS system. For these evaluations, no actual data is stored in the cloud, since the purpose is to gather information about local performance. For each file that was added, the file contents are defined as empty. Therefore, no time is spent encrypting files, and the benchmark focusses on the time consumption of PFS-specific operations: generating data encryption keys, wrapping them, and shredding files (i.e. performing PKW punctures, since no files are stored). The four repositories (Linux kernel, Guava, React, and FreeCodeCamp) are selected for aspects of their history (number of files that were added and deleted). No specific metric is used for the selection; rather, it is based on manual inspection of the characteristics of the history of the repository. Of particular



interest are file deletions, since they affect system performance. A deletion operation in the dataset is mapped to a shred operation in PFS, and therefore deletions lead to growth of the secret key.

To extract the history from a repository, the command `git log --name-status --reverse -M100% | grep -E "(M|A|D|R)[0-9]*\s"` is used and the file-names are cleaned up (whitespaces and commas are replaced by dashes). The first part of the command shows all commits and the involved file names and actions (modify, add, delete, rename). The `--reverse` option reverses the history so that the oldest commit is shown first. The option `-M100%` ensures that only “clean” rename operations are considered a rename (if contents of the file changed, too, it is a deletion and addition). Finally, the `grep` command cleans the output from non-relevant information. Only the git history of the default branch is considered.

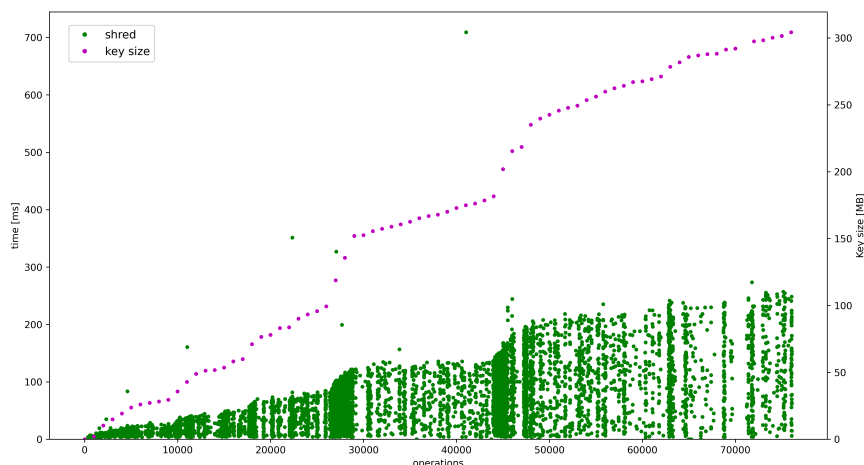
**Linux kernel GitHub history [33]** Initially, the Linux kernel was considered an interesting repository. The first commit is from April 2005, the last from June 2023. However, with 273300 additions and 165300 deletions, it seems too large for practical benchmarks. For its historical significance and to measure the best-performing system more in-depth, the dataset was still created.

**Guava GitHub history [35]** The smallest data-set that was used. The first commit is from June 2009 and until May 2023, about 4500 files were added and 1200 were deleted.

**React GitHub history [34]** A slightly larger dataset with a span from May 2013 to May 2023 and in that time frame about 7000 additions and 4250 deletions.

**FreeCodeCamp GitHub history [36]** The second-to-largest dataset, with its first commit from November 2013 and the last from June 2023. During this time, about 98000 files were added and 55000 files were deleted.

We present the commit history of the React repository run on the implementation described previously in Figure 3.9. It shows an increase in the time used for shred operations, which follows the increasing key size. The longest time used for a shred operation is approximately 700 ms, which is too large for a usable system. The time to delete a file from cloud storage is around 220 ms (“Baseline” in Figure 3.6), so the overhead of using PFS would be noticeable after not too many punctures.



**Figure 3.9:** Key size (purple) and timings for shred operations (green) of the React GitHub history. File identifiers are randomly sampled when files are added (encrypt operation not shown).

### PPRF secret key data structure

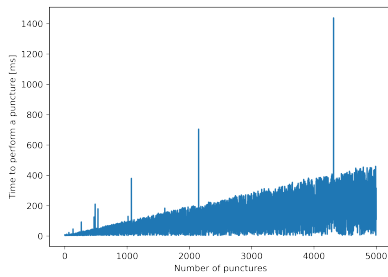
In prior work [26] and also during evaluation, it became apparent that repeated shredding of (random) file identifiers leads to a considerably large secret key. As explained in Chapter 2.2.3, the puncturing algorithm for a PPRF (which is used in the implementation) replaces a single node in the key with a set of nodes. The use of random file identifiers leads to many nodes being added for every puncture. Because a list (C++ `std::vector`) was used to store the GGM nodes of the key (ordered by their label), puncturing became increasingly inefficient. The insertion of the new nodes into the list meant that the list elements had to be moved around, so the ordering remained intact for efficient lookup. A switch to a hashmap to store the nodes (under their label) leads to a significant increase in time performance (see Figure 3.10).

The effect of the change in the key data structure is also very apparent when the React dataset is run with this change (Figure 3.11). Shredding times are much more consistent, with some large outliers that are attributed to the large key size (as in Figure 3.10).

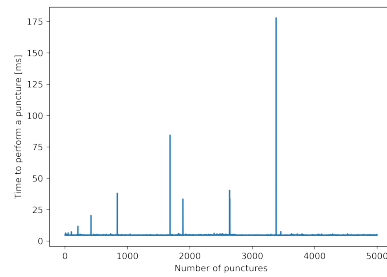
### Large secret key

Although the change in secret key data structure positively impacts execution times, the key size seen in Figure 3.9 and Figure 3.11 remains considerable: after 4250 shred operations, the key size has increased to 300 MB from an initial key size of 72 Bytes. Considering the tag sizes of 256 bits, this is not surprising: since the leaves in the GGM tree (used to wrap the key encryption keys) are very sparse, puncturing them adds a large number

### 3.1. PFS (original definition)

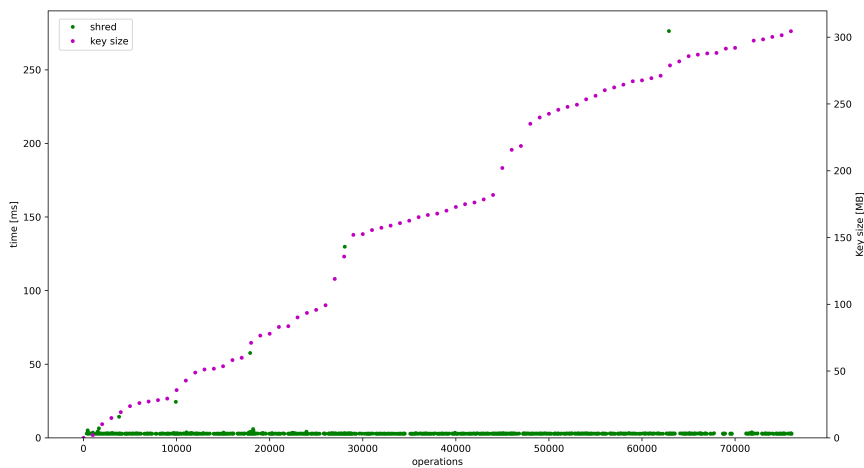


*Time used for puncture operations in PKW, when GGM nodes are stored in a list. The maximum value is approximately 1400ms.*



*Time used for puncture operations in PKW when the GGM nodes are stored in a hashmap. The maximum value is approximately 175ms.*

**Figure 3.10:** Comparison of time usage of punctures on random tags for different PKW secret key data structures. Large spikes are attributed to memory allocation.



**Figure 3.11:** Key size (purple) and timings for shred operations (green) of React GitHub history, with the PKW key data structure changed to a hashmap. File identifiers are sampled randomly when files are added (encrypt operation not shown).

of nodes to the key for every puncture. Using tags of shorter length would somewhat alleviate the problem; however, even halving the tag length does not seem satisfactory, since it would lead to a key of about half the size, which is still very large.

From this result, it seems that using random tags comes at a cost: first, tags must have double the length of what would be required to support a given number of files (e.g., 256 bits to support in the order of  $2^{128}$  files, to avoid collisions of randomly sampled tags (birthday bound)). Second, and even more worryingly, the key size resulting from few punctures quickly becomes untenable.

Led by the intuition that grouping the file identifiers more closely in the GGM tree would be beneficial, we hypothesize that using an increasing counter for file identification, rather than using random values, would lead to more optimal key size.

### Counters as file identifiers

For sufficiently large PKW tag spaces, sampling random tags (as done in the construction of PFS (Figure 3.3, [16])) is not a problem, since tag collisions are a rare event (the birthday bound applies). However, if punctures happen randomly, large tag spaces are not ideal because they lead to a very large key size. Fortunately, nothing speaks against using a counter as a global file identifier: it is not random, but a malicious cloud service will not gain any insight from seeing how many files have been uploaded (which is precisely the information the counter encodes). Moreover, the tag space can be a lot smaller since there is no need to account for collisions in the random sampling of tags (birthday bound). Furthermore, by using a counter, the punctures will result in a (hopefully) small PKW secret key, as the punctured leaves in the PPRF tree will be located more closely. The reason for this assumption is that for  $n_f$  files stored in the system (with sequential identifiers) using PKW space  $\{0,1\}^t$ , the common ancestor in the GGM tree, from which all the leaves corresponding to the files can be derived, is at depth  $\approx t - \log_2(n_f)$ . Due to the way the puncturing algorithm works in the GGM construction, all punctures (except the first) will add at most  $\mathcal{O}(\log_2(n_f))$  (most likely less) nodes to the key, instead of the  $\mathcal{O}(d)$  in the case of random punctures. The mapping from the counter value  $n \in \mathbb{N}_{<2^t}$  to a bitstring in the tag space  $s \in \{0,1\}^t$  is trivial: the number can just be converted to its binary representation.

### Security notions and the cloud

The file identifiers assigned by PFS are given with the intention of them being used to refer to files when they are stored on the cloud [16]. However, there is no “binding” of the file identifier to such usage (the identifier is generated by the scheme, but the security notions do not model the storage service). So, the assignment of file identifiers for cloud storage lies outside the scope of the PFS system and should therefore be modelled separately, since nothing obliges a user of the PFS scheme to use the provided identifiers for cloud storage. The formal security definition of cloud storage is outside the scope of this thesis. In the next section, we define an alternative PFS syntax, which does not use file identifiers.

## 3.2 PFS without file identifiers (PFS<sup>+</sup>)

As stated previously, requiring the file identifiers output by the encryption operation to be random is a little restrictive, and generating file identifiers in the first place is not within the scope of the scheme.

In the following updated definition, file identifiers are therefore removed. Additionally, the confidentiality notion is relaxed to simulatable indistinguishability instead of indistinguishability from random bits. Simulatable indistinguishability is modelled with a simulator, which is allowed to keep state, but is only given a minimum of information (less than the adversary) to simulate output that seems real. To allow the scheme more flexibility with the secret key, the key can be updated during the encryption process to maintain state. For ease of reference, we refer to this updated PFS scheme as PFS<sup>+</sup>.

**Definition 3.4** (PFS<sup>+</sup> scheme). *A protected file storage scheme PFS<sup>+</sup> = (Setup, EncFile, DecFile, ShredFile, RotKey) is a 5-tuple of algorithms with three associated sets; the secret key space SK, the file space F, and the header space H. Associated with the PFS<sup>+</sup> scheme is a ciphertext-length function cl : N → N.*

- Via  $sk \leftarrow \$ \text{Setup}()$ , the probabilistic setup algorithm Setup, taking no input, produces a secret key  $sk \in SK$ .
- Via  $(sk', h, C) / \perp \leftarrow \$ \text{EncFile}(sk, F)$ , the randomized file encryption algorithm EncFile on input the secret key  $sk \in SK$  and a plaintext file  $F \in \mathcal{F}$  produces a potentially updated secret key  $sk'$ , a header  $h \in \mathcal{H}$  and a ciphertext  $C \in \{0, 1\}^{\text{cl}(|F|)}$  or, to indicate failure,  $\perp$ .
- Via  $F / \perp \leftarrow \text{DecFile}(sk, h, C)$ , the deterministic file decryption algorithm DecFile on input the key  $sk \in SK$ , a file header  $h \in \mathcal{H}$ , and a ciphertext  $C \in \{0, 1\}^*$  returns a file plaintext  $F \in \mathcal{F}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{ShredFile}(sk, h)$ , the deterministic shredding algorithm ShredFile on input the secret key  $sk \in SK$  and a header  $h \in \mathcal{H}$  returns the updated secret key  $sk' \in SK$ .
- Via  $(sk', (h'_1, \dots, h'_l)) / (sk', \perp) \leftarrow \$ \text{RotKey}(sk, (h_1, \dots, h_l))$ , the randomized key-rotation algorithm RotKey on input the secret key  $sk \in SK$  and a list of headers  $(h_1, \dots, h_l) \in \mathcal{H}^*$  returns the potentially updated secret key  $sk' \in SK$  and a list of updated headers  $(h'_1, \dots, h'_l) \in \mathcal{H}^*$  or, to indicate failure,  $\perp$ .

For correctness of a PFS<sup>+</sup> scheme, we require:

- Encrypting a new file leaves previously decryptable ciphertext decryptable (the update of the secret key does not impact old ciphertexts).

- A file can always be decrypted after encryption, provided that the secret key  $sk'$  used to decrypt the ciphertext has not experienced a shred operation on the associated header of the file.
- The key rotation algorithm, `RotKey`, leaves all previously decryptable files decryptable (provided that the corresponding headers are passed as arguments).

Formally, we say that for all files  $F \in \mathcal{F}$ , all headers  $(h, h_1, \dots, h_m, h_{m+1}, \dots, h_n) \in \mathcal{H}^n, \forall i \in \{1 \dots, n\} : h \neq h_i$ , and secret key  $sk \leftarrow \$ \text{Setup}()$ , if  $(sk', h, C) \leftarrow \$ \text{EncFile}(sk_{\setminus\{h_1, \dots, h_m\}}, F)$ , then

$$\text{DecFile}(sk'_{\setminus\{h_{m+1}, \dots, h_n\}}, h, C) = F.$$

We use the shorthand  $sk_{\setminus\{h_1, \dots, h_n\}}$  to denote a series of `ShredFile` operations on  $sk$ :  $sk_{\setminus\{h_1, \dots, h_n\}} = \text{ShredFile}(\text{ShredFile}(\dots \text{ShredFile}(sk, h_1) \dots), h_n)$ .

Like in PFS, omitting headers when calling `RotKey` deletes the associated files. However, the intended way to delete a file is `ShredFile`.

### Confidentiality

The security notion for confidentiality is a combination of forward security and indistinguishability from simulation, which we call *forward indistinguishability from simulation* under *real and chosen-plaintext attack* (finds-rcpa). It is essentially the security notion of the original PFS definition, with indistinguishability from random bits replaced by indistinguishability from simulation. This change was initially made to allow more flexibility with file identifiers. With the removal of file identifiers from the syntax, the change however remains an improvement, as headers may wish to encode some information about system state (and must be able to uniquely identify a file), making them nonrandom. For example, in a construction using PKW (as previously presented), a value may be prepended to the header, to encode the PKW tag used to wrap the DEK. A construction may want to use an increasing counter to generate this value; with simulatable indistinguishability, this is allowed (because such headers are simulatable).

In the finds-rcpa game, the adversary is challenged to distinguish between real and simulated outputs of a real or simulated encryption oracle `ROS-ENC`. Part of this output is the header, which is also simulated, thereby encoding a strong form of privacy (the adversary should gain no information from calling the challenge oracle, since it can simulate the output itself). The adversary has access to the `SHRED` oracle and forward security is modelled with the `CORR` oracle, returning the secret key if all challenged files have been shredded. After corruption, challenge queries cannot be made until the `ROTKEY` oracle has been called, which captures a form of post-compromise

security [16, 37]. The adversary also has access to a real encryption oracle,  $\text{ENC}$ , capturing the leakage of encrypted files still in the system after a compromise. The simulation of headers and ciphertexts is produced by a simulator  $\mathbf{S}$ . It has a setup algorithm,  $\mathbf{S}.\text{Setup}()$ , which returns some state  $st$ , and an encryption simulation algorithm,  $\mathbf{S}.\text{SimEncFile}(st, l)$ , taking as input the state  $st$  and a ciphertext length  $l$  to simulate (header lengths are assumed to be independent of ciphertext length). The  $\text{SimEncFile}$  algorithm produces a tuple consisting of a potentially updated state, a header, and a ciphertext  $(st, h, C)$ . A third simulation algorithm, for key-rotation, is  $\mathbf{S}.\text{SimRotKey}(st, n)$ , taking as input the state  $st$  and the number of headers  $n$ , produces a potentially updated state and the specified number of headers. The simulator produces “realistic-looking” output (compared to the PFS<sup>+</sup> algorithms), but can only keep some state that is independent of any secret information.

We provide the  $\text{finds-rcpa}$  game in Figure 3.12.

**Definition 3.5** (PFS<sup>+</sup> confidentiality). *Let PFS<sup>+</sup> be a protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the forward indistinguishability  $\text{finds-rcpa}$  of PFS<sup>+</sup> as*

$$\text{Adv}_{\text{PFS}^+, \mathbf{S}}^{\text{finds-rcpa}}(\mathcal{A}) = 2 \left| \Pr[\mathbf{G}_{\text{PFS}^+, \mathbf{S}}^{\text{finds-rcpa}}(\mathcal{A}) \Rightarrow \text{true}] - \frac{1}{2} \right|.$$

### Integrity

The *ciphertext integrity* ( $\text{int-ctxt}$ ) for the ciphertexts and their associated metadata (headers) is modelled as a game with an adversary given access to an encryption oracle ( $\text{ENC}$ ) and a shredding ( $\text{SHRED}$ ) oracle and a decryption oracle ( $\text{DEC}$ ). The adversary’s goal is to have a ciphertext and header pair that was not produced by the encryption oracle decrypt correctly.

**Definition 3.6** (PFS<sup>+</sup> integrity). *Let PFS<sup>+</sup> be a protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the  $\text{int-ctxt}$  security of PFS<sup>+</sup> as*

$$\text{Adv}_{\text{PFS}^+}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathbf{G}_{\text{PFS}^+}^{\text{int-ctxt}}(\mathcal{A}) \Rightarrow \text{true}].$$

The PFS<sup>+</sup>  $\text{int-ctxt}$  security game is presented in Figure 3.13.

#### 3.2.1 PFS<sup>+</sup> from PKW and AEAD

We present a construction of PFS<sup>+</sup> from PKW and AEAD, a modified version of the construction of PFS (Figure 3.3) to follow the updated syntax. We will refer to this construction by the name  $\text{ctrPFS}$  (Figure 3.14).

In essence, similar to the previous construction, the PKW scheme is used to protect the data encryption keys (DEKs) which are used in the AEAD

<p><b>Game <math>G_{\text{PFS}^+, \mathcal{S}}^{\text{finds-rcpa}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}</math>; <math>sk \leftarrow_{\\$} \text{Setup}()</math></li> <li>2 <math>L_{\text{RoS}} \leftarrow ()</math>; <math>L_{\text{Enc}} \leftarrow ()</math></li> <li>3 <math>\mathcal{S}_h \leftarrow \emptyset</math>; <math>\text{corr} \leftarrow \text{false}</math></li> <li>4 <math>st \leftarrow_{\\$} \mathbf{S.Setup}()</math></li> <li>5 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{RoS-ENC, ENC, SHRED, CORR, ROTKEY}}()</math></li> <li>6 Return <math>b^* = b</math></li> </ol> <p><b>RoS-ENC(<math>F</math>):</b></p> <ol style="list-style-type: none"> <li>7 If <math>\text{corr} = \text{true}</math> then return <math>\perp</math></li> <li>8 <math>(sk, h_1, C_1) \leftarrow_{\\$} \text{EncFile}(sk, F)</math></li> <li>9 If <math>(sk, h_1, C_1) = \perp</math>:</li> <li>10 Return <math>\perp</math></li> <li>11 <math>L_{\text{RoS}} += h_1</math></li> <li>12 <math>(st, h_0, C_0) \leftarrow_{\\$} \mathbf{S.SimEncFile}(st,  C_1 )</math></li> <li>13 <math>\mathcal{S}_h \stackrel{\cup}{\leftarrow} \{h_b\}</math></li> <li>14 Return <math>(h_b, C_b)</math></li> </ol> <p><b>ENC(<math>F</math>):</b></p> <ol style="list-style-type: none"> <li>15 <math>(sk, h, C) \leftarrow_{\\$} \text{EncFile}(sk, F)</math></li> <li>16 <math>(st, -, -) \leftarrow_{\\$} \mathbf{S.SimEncFile}(st,  C )</math></li> <li>17 <math>L_{\text{Enc}} += h</math></li> <li>18 Return <math>(h, C)</math></li> </ol>	<p><b>SHRED(<math>h</math>):</b></p> <ol style="list-style-type: none"> <li>19 <math>sk \leftarrow \text{ShredFile}(sk, h)</math></li> <li>20 <math>L_{\text{RoS}} -= h</math></li> <li>21 <math>L_{\text{Enc}} -= h</math></li> <li>22 <math>\mathcal{S}_h \leftarrow \mathcal{S}_h \setminus \{h\}</math></li> </ol> <p><b>ROTKEY():</b></p> <ol style="list-style-type: none"> <li>23 <math>L \leftarrow L_{\text{RoS}}    L_{\text{Enc}}</math></li> <li>24 <math>(sk, R_1    L_{\text{Enc}}) \leftarrow_{\\$} \text{RotKey}(sk, L)</math></li> <li>25 If <math>R_1    L_{\text{Enc}} = \perp</math>:</li> <li>26 Return <math>\perp</math></li> <li>27 <math>(st, R_0    Q) \leftarrow_{\\$} \mathbf{S.SimRotKey}(st,  L )</math></li> <li>28 <math>\text{corr} \leftarrow \text{false}</math></li> <li>29 Return <math>R_b    L_{\text{Enc}}</math></li> </ol> <p><b>CORR():</b></p> <ol style="list-style-type: none"> <li>30 If <math>\mathcal{S}_h \neq \emptyset</math>:</li> <li>31 Return <math>\perp</math></li> <li>32 <math>\text{corr} \leftarrow \text{true}</math></li> <li>33 Return <math>sk</math></li> </ol>
---	--

**Figure 3.12:** Confidentiality and forward security (finds-rcpa) game for protected file storage scheme  $\text{PFS}^+$ . Lists  $L_{\text{RoS}}$  and  $L_{\text{Enc}}$  keep track of headers currently in the system to facilitate key rotation. We write  $M -= h$  to denote removing a header  $h$  from a list  $M$ , if present.

scheme to encrypt the files. A counter is used for PKW tags instead of randomly chosen tags. This means that when the PKW tag space capacity is reached (the tag counter reaches value  $2^t - 1$ ), no more files can be added.

### 3.2.2 Implementation

Like the implementation of the original PFS scheme,  $\text{ctrPFS}$  is implemented using the PKW library [26], as before, alongside cryptographic primitives from  $\text{CryptoPP}$  [28]. While for the previous construction (Figure 3.3), the (PKW) tag space for random file identifiers was selected to be  $\{0, 1\}^{256}$  to support  $2^{128}$  files, with the removal of file identifiers, the choice of tag space is made for  $\{0, 1\}^{128}$ . The first PKW tag that is assigned is the all-zero bitstring  $0^{128}$ , the second is  $0^{127} || 1$ , etc. This supports a comparable number of files, since the tag space for random tags must be large enough to account for collisions (which by the birthday bound become likely after  $\approx \sqrt{2^{256}}$



<p><b>Game <math>\mathbf{G}_{\text{PFS}^+}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>\text{win} \leftarrow \text{false}; sk \leftarrow \\$ \text{Setup}()</math></li> <li>2 <math>\mathcal{S} \leftarrow \emptyset</math></li> <li>3 <math>\mathcal{A}^{\text{ENC,DEC,SHRED,ROTKEY}}()</math></li> <li>4 <b>Return</b> win</li> </ol> <p><b>ENC(<math>F</math>):</b></p> <ol style="list-style-type: none"> <li>5 <math>(sk, h, C) \leftarrow \\$ \text{EncFile}(sk, F)</math></li> <li>6 <math>\mathcal{S} \stackrel{\cup}{\leftarrow} \{h, C\}</math></li> <li>7 <b>Return</b> <math>(h, C)</math></li> </ol> <p><b>SHRED(<math>h</math>):</b></p> <ol style="list-style-type: none"> <li>8 <math>sk \leftarrow \text{ShredFile}(sk, h)</math></li> <li>9 <math>\mathcal{S} \leftarrow \mathcal{S} \setminus \{(h, *)\}</math></li> </ol>	<p><b>DEC(<math>h, C</math>):</b></p> <ol style="list-style-type: none"> <li>10 <math>F \leftarrow \text{DecFile}(sk, h, C)</math></li> <li>11 <b>If</b> <math>(h, C) \notin \mathcal{S}</math> <b>and</b> <math>F \neq \perp</math>:</li> <li>12     <math>\text{win} \leftarrow \text{true}</math></li> <li>13 <b>Return</b> <math>F</math></li> </ol> <p><b>ROTKEY(<math>(h_1, \dots, h_l)</math>):</b></p> <ol style="list-style-type: none"> <li>14 <math>L \leftarrow (h_1, \dots, h_l)</math></li> <li>15 <math>(sk, (h'_1, \dots, h'_l)) \leftarrow \\$ \text{RotKey}(sk, L)</math></li> <li>16 <b>If</b> <math>(h'_1, \dots, h'_l) = \perp</math></li> <li>17     <b>Return</b> <math>\perp</math></li> <li>18 <math>\mathcal{S}_{\text{new}} \leftarrow \emptyset</math></li> <li>19 <b>For</b> <math>(h, C) \in \mathcal{S}</math> <b>do</b>:</li> <li>20     <b>If</b> <math>\exists i \in \{1, \dots, l\}</math>, s.t. <math>h = h_i</math></li> <li>21         <math>\mathcal{S}_{\text{new}} \stackrel{\cup}{\leftarrow} \{(h'_i, C)\}</math></li> <li>22 <math>\mathcal{S} \leftarrow \mathcal{S}_{\text{new}}</math></li> <li>23 <b>Return</b> <math>(h'_1, \dots, h'_l)</math></li> </ol>
---	---

**Figure 3.13:** *Ciphertext integrity (int-ctxt) game for protected file storage scheme PFS<sup>+</sup>.*

tags are sampled). The PKW tags (assigned in increasing order) are used as names to store the header and ciphertext on GCS as before.

### 3.2.3 Evaluation

For the evaluation of the updated PFS scheme, we focus solely on the GitHub histories and the resulting key sizes and execution times for the ctrPFS construction. As a first comparison, Figure 3.15 shows a reduction of the key size by three orders of magnitude and much more consistent shredding times compared to Figure 3.11. The more consistent shredding times are likely due to the smaller key size.

Apart from the first shred operation, most shred operations take less than 1 millisecond to complete. The first operation takes longer (about 1.6 milliseconds), the reason being that the very first puncture on the PKW key adds 127 new nodes to the key (the co-path from the root node to the first punctured node). Subsequent punctures will add at most  $\log_2(7000) < 13$  (the React dataset stores about 7000 files overall) new nodes per puncture to the key.

An interesting detail is that the key size can decrease, especially if many files are deleted at once. This usually happens when directories are deleted, and it is probably because the files inside the directories were added sequentially, i.e. they have contiguous PKW tags. When they are all deleted, a whole

<p><u>Setup()</u> :</p> <ol style="list-style-type: none"> <li>1 <math>T[\cdot] \leftarrow \perp</math></li> <li>2 <math>sk_{PKW} \leftarrow \\$ PKW.KeyGen()</math></li> <li>3 <math>sk \leftarrow (sk_{PKW}, 0, T)</math></li> <li>4 Return <math>sk</math></li> </ol> <p><u>DecFile(<math>sk, h, N    C</math>):</u></p> <ol style="list-style-type: none"> <li>5 <math>(sk_{PKW}, \cdot, T) \leftarrow sk</math></li> <li>6 <math>id \leftarrow T[h]</math></li> <li>7 If <math>id = \perp</math> then return <math>\perp</math></li> <li>8 <math>K \leftarrow PKW.Unwrap(sk_{PKW}, id, \epsilon, h)</math></li> <li>9 If <math>K = \perp</math> then return <math>\perp</math></li> <li>10 <math>F \leftarrow AEAD.Dec(K, N, \epsilon, C)</math></li> <li>11 Return <math>F</math></li> </ol> <p><u>ShredFile(<math>sk, h</math>):</u></p> <ol style="list-style-type: none"> <li>12 <math>(sk_{PKW}, ctr, T) \leftarrow sk</math></li> <li>13 <math>id \leftarrow T[h]</math></li> <li>14 If <math>id = \perp</math> then return <math>sk</math></li> <li>15 <math>sk'_{PKW} \leftarrow PKW.Punc(sk_{PKW}, id)</math></li> <li>16 Return <math>(sk'_{PKW}, ctr, T)</math></li> </ol>	<p><u>EncFile(<math>sk, F</math>):</u></p> <ol style="list-style-type: none"> <li>17 <math>(sk_{PKW}, ctr, T) \leftarrow sk</math></li> <li>18 <math>K \leftarrow \\$ \{0, 1\}^k</math>;</li> <li>19 <math>h \leftarrow PKW.Wrap(sk, [ctr]_t, \epsilon, K)</math></li> <li>20 If <math>h = \perp</math> :</li> <li>21 Return <math>\perp</math></li> <li>22 <math>T[h] \leftarrow [ctr]_t</math></li> <li>23 <math>N \leftarrow \\$ \{0, 1\}^n</math></li> <li>24 <math>C \leftarrow AEAD.Enc(K, N, \epsilon, F)</math></li> <li>25 <math>sk \leftarrow (sk_{PKW}, ctr + 1, T)</math></li> <li>26 Return <math>(sk, h, N    C)</math></li> </ol> <p><u>RotKey(<math>sk_{old}, (h_1, \dots, h_l)</math>):</u></p> <ol style="list-style-type: none"> <li>27 <math>(sk_{PKW}, ctr, T) \leftarrow sk_{old}</math></li> <li>28 <math>sk'_{PKW} \leftarrow \\$ PKW.KeyGen()</math></li> <li>29 For <math>i \leftarrow 0</math> to <math>l - 1</math> do:</li> <li>30 <math>id_i \leftarrow T[h_i]</math></li> <li>31 If <math>id_i = \perp</math> then return <math>(sk_{old}, \perp)</math></li> <li>32 <math>K_i \leftarrow PKW.Unwrap(sk_{PKW}, id_i, \epsilon, h_i)</math></li> <li>33 <math>h'_i \leftarrow PKW.Wrap(sk'_{PKW}, [i]_t, \epsilon, K_i)</math></li> <li>34 If <math>h'_i = \perp</math> then return <math>(sk_{old}, \perp)</math></li> <li>35 <math>T[h_i] \leftarrow \perp</math>; <math>T[h'_i] \leftarrow i</math></li> <li>36 <math>sk_{new} \leftarrow (sk'_{PKW}, l, T)</math></li> <li>37 Return <math>(sk_{new}, (h'_1, \dots, h'_l))</math></li> </ol>
--	--

**Figure 3.14:**  $PFS^+$  construction from composition of PKW scheme PKW and AEAD scheme AEAD, using counters. The key-wrap space of the PKW scheme is  $\{0, 1\}^k$ , the tag space is  $\{0, 1\}^t$ . The AEAD scheme has nonce space  $\{0, 1\}^n$  and key space  $\{0, 1\}^k$ . The instantiated  $PFS^+$  scheme therefore has header space  $\mathcal{H} = \{0, 1\}^{PKW.cl(k)}$  and an associated ciphertext length of  $PFS.cl(|F|) = AEAD.cl(|F|) + n$ .

subtree of GGM nodes is removed, leading to the reduced key size. For the larger dataset FreeCodeCamp (Figure 3.16), the key size remains reasonably small, and the shred execution times remain consistent throughout the test. Note also the previously described behaviour for the first shred operation, which takes longer than later shreds.

Therefore, an intriguing amendment to the scheme is to try to encode the file and directory hierarchy into the scheme, so that the effect of directory deletions can be better utilized.

### 3.2. PFS without file identifiers (PFS<sup>+</sup>)

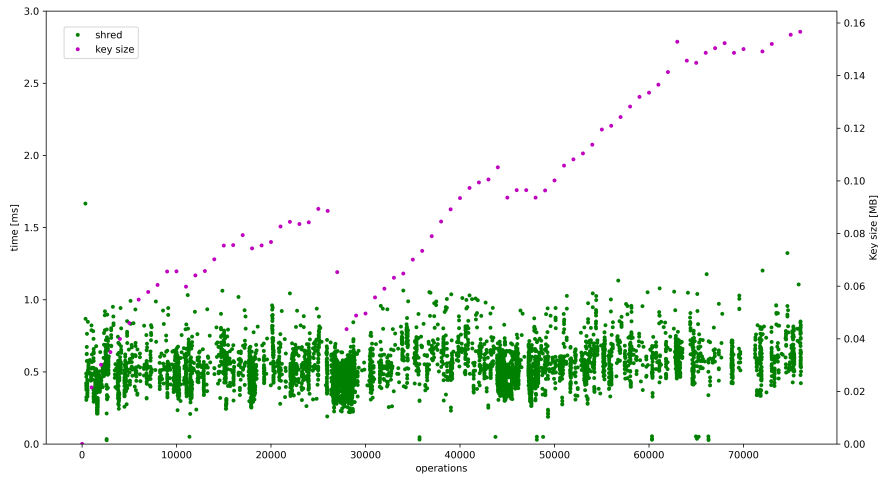


Figure 3.15: React GitHub history run on ctrPFS with a PKW tag space of  $\{0,1\}^{128}$ .

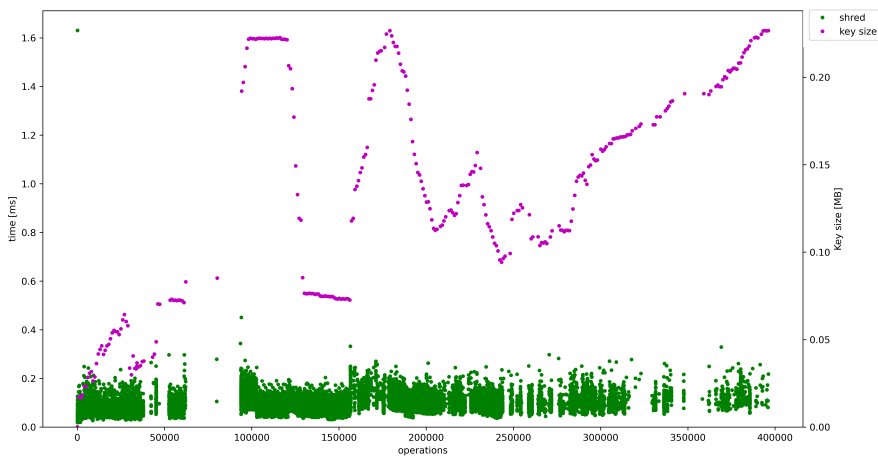


Figure 3.16: FreeCodeCamp GitHub history on ctrPFS with a PKW tag space of  $\{0,1\}^{128}$ .



## Chapter 4

---

# PFS with file hierarchy

---

The PFS scheme presented in Chapter 3 has a shortcoming: it is unaware of the hierarchies in which the files are stored locally. Typically, files are stored in some form of hierarchy, consisting of directories, which in turn contain other directories or files. When a directory is deleted, all the objects it contains (transitively, i.e. all the objects that lie below it hierarchically) are also deleted. To obtain an equivalent functionality in PFS, the shredding operation needs to be adapted, to allow the shredding of entire directories.

We will explore how this change could be achieved by adjusting the primitives used to instantiate PFS: PPRF and PKW. Forward security in this approach is provided by the key of the adapted PKW scheme. An alternative construction, using PKW, is also presented. Here, each directory has a distinct PKW key, which is used to wrap keys for files stored within that directory. We hypothesize that the first construction will be more space- and time-efficient, since only a single key must be stored.

We will first define the syntax and security notions for PFS with this extended functionality. Then, we will present modifications to the primitives, PPRF and PKW, and the constructions. Finally, we will evaluate the performance of the two constructions.

We provide an adapted syntax for this *hierarchical* PFS (HPFS).

### 4.1 Hierarchical protected file storage (HPFS)

While the definition of PFS is not aware of the local storage location of files, in practical scenarios, files are often organized in a hierarchical structure. To harness the benefits of this hierarchy, it is possible to extend the PFS definition by introducing a path space. This path space can be visualized as a tree, with each path corresponding to a sequence of connected nodes and having a distinct parent path. A node has a name and is uniquely identified

by the concatenation of the names of the nodes leading to it, from the root. This defines the path name. By incorporating the concept of a path space into the PFS scheme, the shredding operation can take advantage of the hierarchical structure, facilitating the shredding of entire subtrees.

We begin by providing a definition of a path space as is used in this work.

**Definition 4.1** (Path space). *A path space  $\mathcal{P}$  is a rooted tree in which vertices (or nodes) have names and the edges connect a parent to their child nodes. The special root node has the name  $\epsilon$ . A path is a connected sequence of these nodes, the first of which is the root node, with the root path being the sequence containing only the root node. Associated to the path space is the function  $\text{parent} : \mathcal{P} \rightarrow \mathcal{P}$ , which returns the parent of a path. The parent is defined to be the same sequence of nodes, without the last node, or, for the root path, the root path itself. The relation  $\preceq$  denotes whether a path  $p_1$  is a prefix (“transitive parent”) to another path  $p_2$ , or  $p_1 = p_2$ .*

Using the definition of a path space, we present the syntax of hierarchical protected file storage (HPFS).

**Definition 4.2** (Hierarchical PFS scheme). *A hierarchical protected file storage scheme  $\text{HPFS} = (\text{Setup}, \text{EncFile}, \text{DecFile}, \text{Shred}, \text{RotKey})$  is a 5-tuple of algorithms with four associated sets; the secret key space  $\mathcal{SK}$ , the file space  $\mathcal{F}$ , the path space  $\mathcal{P}$  (as defined in Definition 4.1), and the header space  $\mathcal{H}$ . Associated to HPFS is also a ciphertext-length function,  $\text{cl} : \mathbb{N} \rightarrow \mathbb{N}$ .*

- Via  $sk \leftarrow \$ \text{Setup}()$ , the probabilistic setup algorithm  $\text{Setup}$ , taking no input, produces a secret key  $sk \in \mathcal{SK}$ .
- Via  $(sk', h, C) / \perp \leftarrow \$ \text{EncFile}(sk, F, P)$ , the randomized file encryption algorithm  $\text{EncFile}$  on input the secret key  $sk \in \mathcal{SK}$  and a plaintext file  $F \in \mathcal{F}$  and the path  $P \in \mathcal{P}$  produces a potentially updated secret key  $sk' \in \mathcal{SK}$ , a header  $h \in \mathcal{H}$ , and a ciphertext  $C \in \{0, 1\}^{\text{cl}(|F|)}$ , or, to indicate failure,  $\perp$ .
- Via  $F / \perp \leftarrow \text{DecFile}(sk, h, C)$ , the deterministic file decryption algorithm  $\text{DecFile}$  on input the key  $sk \in \mathcal{SK}$ , a file header  $h \in \mathcal{H}$ , and a ciphertext  $C \in \{0, 1\}^*$  returns a file plaintext  $F \in \mathcal{F}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{Shred}(sk, P)$ , the deterministic file shredding algorithm  $\text{Shred}$  on input the secret key  $sk \in \mathcal{SK}$  and a path  $P \in \mathcal{P}$  returns the updated secret key  $sk' \in \mathcal{SK}$ .
- Via  $(sk', (h'_1, \dots, h'_j)) / (sk', \perp) \leftarrow \text{RotKey}(sk, ((h_1, \dots, h_j)))$ , the randomized key-rotation algorithm  $\text{RotKey}$  on input the secret key  $sk \in \mathcal{SK}$  and a list of headers  $(h_1, \dots, h_j) \in \mathcal{H}^*$  returns the potentially updated secret key  $sk' \in \mathcal{SK}$  and a sequence of updated headers  $(h'_1, \dots, h'_j) \in \mathcal{H}^*$  or, to indicate failure,  $\perp$ .

*Correctness* of the HPFS scheme can be described as follows:

- We require that encrypting a file leaves previously encrypted files decryptable.
- Key rotation leaves all previously decryptable files decryptable, provided that the corresponding headers are passed to the key rotation algorithm.
- We require that a file can always be decrypted after encryption, provided that the secret key  $sk$  used to encrypt the file has not undergone a shred operation on a prefix of the file path.

Formally, we say that for all files  $F \in \mathcal{F}$ , all paths  $P \in \mathcal{P}$ ,  $(P_1, \dots, P_m, P_{m+1}, \dots, P_n) \in \mathcal{P}^n$  s.t.  $\forall i \in \{1, \dots, n\} : P_i \not\preceq P$  and secret key  $sk \xleftarrow{\$} \text{Setup}()$ , if  $(sk', h, C) \xleftarrow{\$} \text{EncFile}(sk_{\setminus\{P_1, \dots, P_m\}}, F, P)$ , then

$$\text{DecFile}(sk'_{\setminus\{P_{m+1}, \dots, P_n\}}, h, C) = F.$$

We use the shorthand  $sk_{\setminus\{P_1, \dots, P_n\}}$  to denote a series of Shred operations on  $sk$ :  $sk_{\setminus\{P_1, \dots, P_n\}} = \text{Shred}(\text{Shred}(\dots \text{Shred}(sk, P_1) \dots), P_n)$ .

The most impactful change in the syntax is the addition of paths. They provide information on the location of the file within the file storage system and allow hierarchical shredding: calling  $\text{Shred}(sk, P)$  shreds everything stored in the scheme located at  $P$  and paths that “lie under”  $P$  (other paths  $P'$ , s.t.  $P \preceq P'$ ). Because not all paths can be known to the scheme in advance ( $\mathcal{P}$  is possibly infinite) and the secret key is somewhat dependent on the paths of the encrypted files, the secret key may change when  $\text{EncFile}$  is called. In the key, information about previously seen paths has to be kept. To enable hierarchical shredding, the scheme needs to map the paths passed to it during encryption into the keyspace. Letting the  $\text{EncFile}$  algorithm update the secret key provides the necessary flexibility for this.

Files are referred to by their path (which is user-specified). This differs from PFS, where files are referred to with the file identifier (or header, with the updated syntax) assigned by the scheme. The paths, therefore, act as user-chosen file identifiers of which the scheme keeps track.

We provide the security game for the finds-rcpa security notion that we have already presented for PFS<sup>+</sup> (Figure 3.12) in Figure 4.1. It follows the new syntax (addition of paths) and has some changes related to the use of paths: the checks for trivial attacks are updated to capture the hierarchical shred operation, and the lists for key rotation are updated to include the path.

**Definition 4.3** (HPFS confidentiality). *Let HPFS be a hierarchical protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the forward*

indistinguishability finds-rcpa of HPFS as

$$\text{Adv}_{\text{HPFS}, \mathcal{S}}^{\text{finds-rcpa}}(\mathcal{A}) = 2 \left| \Pr[\mathbf{G}_{\text{HPFS}, \mathcal{S}}^{\text{finds-rcpa}}(\mathcal{A}) \Rightarrow \text{true}] - \frac{1}{2} \right|.$$

<p><b>Game <math>\mathbf{G}_{\text{HPFS}, \mathcal{S}}^{\text{finds-rcpa}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}</math>; <math>sk \leftarrow_{\\$} \text{Setup}()</math></li> <li>2 <math>L_{\text{RoS}} \leftarrow ()</math>; <math>L_{\text{Enc}} \leftarrow ()</math></li> <li>3 <math>\mathcal{C} \leftarrow \emptyset</math>; <math>\mathcal{P} \leftarrow \emptyset</math>; <math>\text{corr} \leftarrow \text{false}</math></li> <li>4 <math>st \leftarrow_{\\$} \mathbf{S}.\text{Setup}()</math></li> <li>5 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{RoS-ENC, ENC, SHRED, CORR, ROTKEY}}()</math></li> <li>6 Return <math>b^* = b</math></li> </ol> <p><b>RoS-ENC(<math>F, P</math>):</b></p> <ol style="list-style-type: none"> <li>7 If <math>\text{corr} = \text{true}</math> then return <math>\perp</math></li> <li>8 <math>(sk, h_1, C_1) \leftarrow_{\\$} \text{EncFile}(sk, F, P)</math></li> <li>9 If <math>(sk, h_1, C_1) = \perp</math>:</li> <li>10 Return <math>\perp</math></li> <li>11 <math>(st, h_0, C_0) \leftarrow_{\\$} \mathbf{S}.\text{SimEncFile}(st,  C_1 )</math></li> <li>12 <math>L_{\text{RoS}} += (P, h_1)</math></li> <li>13 <math>\mathcal{C} \leftarrow \{P\}</math></li> <li>14 Return <math>(h_b, C_b)</math></li> </ol> <p><b>ENC(<math>F, P</math>):</b></p> <ol style="list-style-type: none"> <li>15 <math>(sk, h, C) \leftarrow_{\\$} \text{EncFile}(sk, F, P)</math></li> <li>16 <math>(st, -, -) \leftarrow_{\\$} \mathbf{S}.\text{SimEncFile}(st,  C )</math></li> <li>17 <math>L_{\text{Enc}} += (P, h)</math></li> <li>18 Return <math>(h, C)</math></li> </ol>	<p><b>SHRED(<math>P</math>):</b></p> <ol style="list-style-type: none"> <li>19 <math>sk \leftarrow \text{Shred}(sk, P)</math></li> <li>20 <math>L_{\text{RoS}} -= (P^*, *)</math>; <math>L_{\text{Enc}} -= (P^*, *)</math></li> <li>21 <math>\mathcal{P} \leftarrow \{P\}</math></li> </ol> <p><b>ROTKEY():</b></p> <ol style="list-style-type: none"> <li>22 <math>P_{\text{Enc}}, H_{\text{Enc}} \leftarrow \text{unzip}(L_{\text{Enc}})</math></li> <li>23 <math>P_{\text{RoS}}, H_{\text{RoS}} \leftarrow \text{unzip}(L_{\text{RoS}})</math></li> <li>24 <math>L \leftarrow H_{\text{RoS}}    H_{\text{Enc}}</math></li> <li>25 <math>(sk, R_1    H_{\text{Enc}}) \leftarrow_{\\$} \text{RotKey}(sk, L)</math></li> <li>26 If <math>R_1    H_{\text{Enc}} = \perp</math> then return <math>\perp</math></li> <li>27 <math>(st, R_0    Q) \leftarrow_{\\$} \mathbf{S}.\text{SimRotKey}(st,  L )</math></li> <li>28 <math>L_{\text{Enc}} \leftarrow \text{zip}(P_{\text{Enc}}, H_{\text{Enc}})</math></li> <li>29 <math>L_{\text{RoS}} \leftarrow \text{zip}(P_{\text{RoS}}, R_1)</math></li> <li>30 <math>\text{corr} \leftarrow \text{false}</math></li> <li>31 Return <math>R_b    H_{\text{Enc}}</math></li> </ol> <p><b>CORR():</b></p> <ol style="list-style-type: none"> <li>32 If <math>\exists P \in \mathcal{C}</math> s.t. <math>\forall P' \in \mathcal{P}, P' \not\prec P</math>:</li> <li>33 Return <math>\perp</math> // all challenged paths must be punctured</li> <li>34 <math>\text{corr} \leftarrow \text{true}</math></li> <li>35 Return <math>sk</math></li> </ol>
--	---

**Figure 4.1:** Security games for confidentiality and forward-security of HPFS. The syntax  $R -= (P^*, *)$  denotes that all pairs in  $R$  in which the first entry is  $P$  or has  $P$  as a prefix are removed. The functions  $\text{zip}()$  and  $\text{unzip}()$ , combine, or split up lists, respectively.  $\text{unzip}(L)$  takes a list consisting of pairs and returns two lists, the first containing all the right entries of the pairs in the list  $L$ , the second all the left entries. The function  $\text{zip}()$  does the opposite. Differences to the  $\text{PFS}^+$  game are *highlighted*.

The ciphertext integrity game remains largely the same as before (for  $\text{PFS}^+$ ), and the differences are highlighted in the game definition (Figure 4.2).

**Definition 4.4** (HPFS integrity). *Let HPFS be a hierarchical protected file storage scheme. We define the advantage of an adversary  $\mathcal{A}$  against the int-ctxt security of HPFS as*

$$\text{Adv}_{\text{HPFS}}^{\text{int-ctxt}}(\mathcal{A}) = \Pr[\mathbf{G}_{\text{HPFS}}^{\text{int-ctxt}}(\mathcal{A}) \Rightarrow \text{true}].$$



<p><b>Game <math>G_{\text{HPFS}}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>\text{win} \leftarrow \text{false}; sk \leftarrow \\$ \text{Setup}()</math></li> <li>2 <math>\mathcal{S} \leftarrow \emptyset</math></li> <li>3 <math>\mathcal{A}^{\text{ENC,DEC,SHRED,ROTKEY}}()</math></li> <li>4 Return win</li> </ol> <p><b>ENC(<math>F, P</math>):</b></p> <ol style="list-style-type: none"> <li>5 <math>(sk, h, C) \leftarrow \\$ \text{EncFile}(sk, F, P)</math></li> <li>6 <math>\mathcal{S} \stackrel{\text{u}}{\leftarrow} \{P, h, C\}</math></li> <li>7 Return <math>(h, C)</math></li> </ol> <p><b>SHRED(<math>P</math>):</b></p> <ol style="list-style-type: none"> <li>8 <math>sk \leftarrow \text{Shred}(sk, P)</math></li> <li>9 <math>\mathcal{S} \leftarrow \{(P', h, C) \in \mathcal{S} \mid P \not\preceq P'\}</math></li> </ol>	<p><b>DEC(<math>h, C</math>):</b></p> <ol style="list-style-type: none"> <li>10 <math>F \leftarrow \text{DecFile}(sk, h, C)</math></li> <li>11 If <math>(*, h, C) \notin \mathcal{S}</math> and <math>F \neq \perp</math>:</li> <li>12     <math>\text{win} \leftarrow \text{true}</math></li> <li>13 Return <math>F</math></li> </ol> <p><b>ROTKEY(<math>(h_1, \dots, h_l)</math>):</b></p> <ol style="list-style-type: none"> <li>14 <math>L \leftarrow (h_1, \dots, h_l)</math></li> <li>15 <math>(sk, (h'_1, \dots, h'_l)) \leftarrow \\$ \text{RotKey}(sk, L)</math></li> <li>16 If <math>(h'_1, \dots, h'_l) = \perp</math></li> <li>17     Return <math>\perp</math></li> <li>18 <math>\mathcal{S}_{\text{new}} \leftarrow \emptyset</math></li> <li>19 For <math>(P, h, C) \in \mathcal{S}</math> do:</li> <li>20     If <math>\exists i \in \{1, \dots, l\}</math>, s.t. <math>h = h_i</math>:</li> <li>21         <math>\mathcal{S}_{\text{new}} \stackrel{\text{u}}{\leftarrow} \{(P, h'_i, C)\}</math></li> <li>22 <math>\mathcal{S} \leftarrow \mathcal{S}_{\text{new}}</math></li> <li>23 Return <math>(h'_1, \dots, h'_l)</math></li> </ol>
--	--

**Figure 4.2:** Ciphertext integrity (int-ctxt) game for hierarchical protected file storage scheme HPFS. The main differences to the int-ctxt game for PFS+ are *highlighted*.

## 4.2 Constructing HPFS using hierarchically puncturable pseudo-random function (hPPRF)

In this section, we explore a modification of the PPRF scheme to support puncturing on elements that are related to instantiate HPFS. The modified PPRF scheme (hierarchically puncturable pseudo-random function (hPPRF)) can then be used to build a hierarchically puncturable key-wrapping (hPKW) scheme, which in turn permits the construction of HPFS.

We use  $a \preceq b$  to denote the relation of  $a$  to  $b$  and impose the relation on the domain  $\mathcal{X}$  of the PPRF. In essence, for  $a, b \in \mathcal{X}$ , with  $a \preceq b$ , a puncture of  $a$  should also puncture  $b$ .

### 4.2.1 Hierarchically puncturable PRF (hPPRF)

Building upon the definition of a PPRF, we present the definition of hierarchically puncturable pseudo-random functions (hPPRFs). It is almost identical to the definition of PPRF, with the puncturing operation additionally allowing one to perform “set”-punctures: elements that are organized in a hierarchical set can be punctured by only puncturing on an element representing that set; an element of the domain which is related to the other elements.

This also naturally introduces the idea that evaluations of such “representative” elements of the domain should allow the derivation of the “represented” values. This is reminiscent of *delegatable* PRFs [24], where a Delegate functionality allows the sharing (*delegating*) of parts of the secret key. For now, however, we focus only on making the punctures hierarchical and leave the syntax the same as for PPRF. The result of evaluating  $a$  with  $a \preceq b$  accordingly does not “leak” any information about the result of evaluating  $b$ .

**Definition 4.5** (hPPRF). *A hierarchically puncturable pseudo-random function hPPRF = (KeyGen, Eval, Punc) is a 3-tuple of algorithms with three associated sets; the secret key space  $\mathcal{SK}$ , the domain  $\mathcal{X} \subseteq \{0,1\}^*$  s.t.  $\mathcal{X}$  is length-closed<sup>1</sup>, and the range  $\mathcal{Y}$ .*

- Via  $sk \leftarrow \$ \text{KeyGen}()$ , the probabilistic key generation algorithm KeyGen, taking no input, outputs the secret key  $sk \in \mathcal{SK}$ .
- Via  $y / \perp \leftarrow \text{Eval}(sk, x)$ , the function evaluation algorithm Eval, taking as input the secret key  $sk \in \mathcal{SK}$  and an element  $x \in \mathcal{X}$ , outputs  $y \in \mathcal{Y}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{Punc}(sk, x)$ , the deterministic puncturing algorithm Punc taking as input the secret key  $sk \in \mathcal{SK}$  and an element  $x \in \mathcal{X}$ , outputs an updated secret key  $sk' \in \mathcal{SK}$ .

For the correctness of a *hierarchically puncturable* PRF, we require that for all  $sk \in \mathcal{SK}$  and all  $x, y \in \mathcal{X}$ :

- (1)  $\Pr[\text{Eval}(sk_0, x) \neq \perp \mid sk_0 \leftarrow \$ \text{KeyGen}()] = 1$ .
- (2) If  $sk' \leftarrow \text{Punc}(sk, x)$  and  $x \not\preceq y$ , then  $\text{Eval}(sk', y) = \text{Eval}(sk, y)$ .
- (3) If  $sk' \leftarrow \text{Punc}(sk, x)$  then for all  $z \in \mathcal{X}$  s.t.  $x \preceq z$ ,  $\text{Eval}(sk', z) = \perp$ .

Requirement (1) ensures that for any freshly generated key  $sk_0$ , and any element  $x \in \mathcal{X}$ ,  $\text{Eval}(sk_0, x) \neq \perp$ . Requirement (2) states that puncturing any secret key  $sk$  on  $x$  will affect only  $x$  and other elements to which  $x$  is related. Requirement (3) stipulates that the evaluation of a punctured element is always  $\perp$ . The puncture of an element can be explicit (i.e. Punc called on the element itself) or implicit (i.e. the element has a punctured element that is related to it).

We present the hPPRF security games, adapted from the PPRF security games shown in [16] (Figure 2.3). The security notion they capture is described as *forward pseudorandomness*, which is a combination of pseudorandomness and forward security. The changes introduced to enable hierar-

<sup>1</sup> $\forall l \in \mathbb{N}$ , either  $\{0,1\}^l \subseteq \mathcal{X}$  or  $\{0,1\}^l \cap \mathcal{X} = \emptyset$

chical puncturing do not affect the games much, and we present two security notions: *real-or-random* (fpr-ro\$), and the stronger *real-and-real-or-random* (fpr-rro\$) which also allows real evaluations. These are almost identical to the PPRF security notions, with only the sets preventing trivial attacks changing slightly (Figure 4.3).

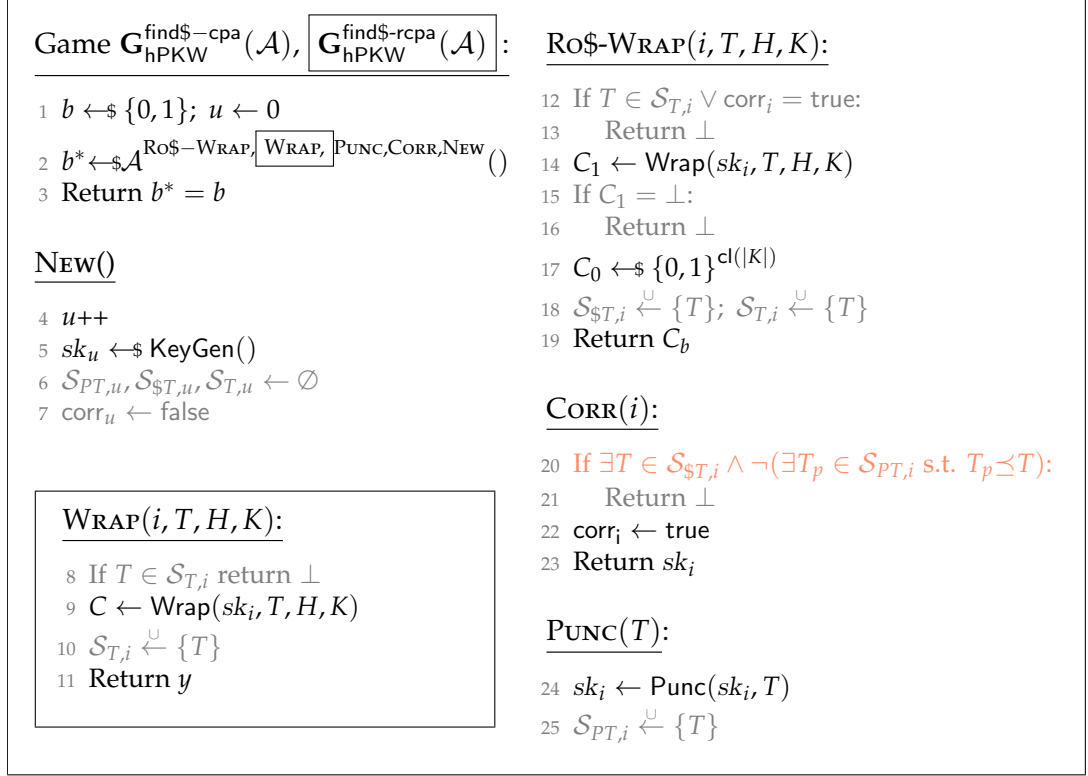
The advantage terms stay the same as for PPRF (see Chapter 2.2.3) and are not repeated here.

<p><b>Game <math>G_{\text{hPPRF}}^{\text{fpr-ro\\$}}(\mathcal{A}), G_{\text{hPPRF}}^{\text{fpr-rro\\$}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>b \leftarrow_{\\$} \{0, 1\}; u \leftarrow 0; \mathbf{T}[\cdot, \cdot] \leftarrow \perp</math></li> <li>2 <math>b^* \leftarrow_{\\$} \mathcal{A}^{\text{NEW, Ro\\$-EVAL, EVAL, CORR, PUNC}}()</math></li> <li>3 <b>Return</b> <math>b^* = b</math></li> </ol> <p><b>NEW()</b></p> <ol style="list-style-type: none"> <li>4 <math>u++; sk_u \leftarrow_{\\$} \text{KeyGen}()</math></li> <li>5 <math>\mathcal{C}_u, \mathcal{E}_u, \mathcal{P}_u \leftarrow \emptyset</math></li> </ol> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><b>EVAL(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>6 <b>If</b> <math>x \in \mathcal{C}_i</math> <b>then return</b> <math>\perp</math></li> <li>7 <math>y \leftarrow \text{Eval}(sk_i, x)</math></li> <li>8 <math>\mathcal{E}_i \leftarrow \cup \{x\}</math></li> <li>9 <b>Return</b> <math>y</math></li> </ol> </div>	<p><b>Ro\\$-EVAL(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>10 <b>If</b> <math>x \in \mathcal{E}_i</math> <b>or</b> <math>\text{corr}_i</math> <b>then return</b> <math>\perp</math></li> <li>11 <math>y_1 \leftarrow \text{Eval}(sk_i, x)</math></li> <li>12 <b>If</b> <math>y_1 = \perp</math> <b>then return</b> <math>\perp</math></li> <li>13 <b>If</b> <math>\mathbf{T}[i, x] = \perp</math>:</li> <li>14     <math>\mathbf{T}[i, x] \leftarrow_{\\$} \mathcal{Y}</math></li> <li>15 <math>y_0 \leftarrow \mathbf{T}[i, x]</math></li> <li>16 <math>\mathcal{C}_i \leftarrow \cup \{x\}</math></li> <li>17 <b>Return</b> <math>y_b</math></li> </ol> <p><b>PUNC(<math>i, x</math>):</b></p> <ol style="list-style-type: none"> <li>18 <math>sk_i \leftarrow \text{Punc}(sk_i, x)</math></li> <li>19 <math>\mathcal{P}_i \leftarrow \cup \{x\}</math></li> </ol> <p><b>CORR(<math>i</math>):</b></p> <ol style="list-style-type: none"> <li>20 <b>If</b> <math>\exists x \in \mathcal{C}_i \wedge \neg(\exists x_p \in \mathcal{P}_i \text{ s.t. } x_p \preceq x)</math>:</li> <li>21     <b>Return</b> <math>\perp</math></li> <li>22 <math>\text{corr}_i \leftarrow \text{true}</math></li> <li>23 <b>Return</b> <math>sk_i</math></li> </ol>
---	--

**Figure 4.3:** Game definitions for the forward pseudo-randomness security notions *fpr-ro\$* (real-or-random) and *fpr-rro\$* (real-and-real-or-random, with additional access to a real Eval oracle). Note that the security game remains almost the same as for PPRF — the checks on the sets used to prevent trivial attacks are highlighted to show the difference to the PPRF game.

## 4.2.2 An instantiation of hPPRF

As mentioned by [20, 19] and others, GGM trees [23] can be used to instantiate a PPRF. This approach can be modified to allow for punctures on “inner” tree nodes, removing whole subtrees at a time. The relation  $\preceq$ , in this case, is the prefix relation of bitstrings:  $a \preceq b$  means that  $a$  is a prefix to  $b$  (note that  $a \preceq a$ ). For example,  $100 \preceq 10010$ , and  $010 \not\preceq 1100$ . An interesting detail is that since elements of any length can be evaluated or punctured,



**Figure 4.4:** Confidentiality and forward security (find\\$-cpa, find\\$-rcpa) games for a hierarchically puncturable key-wrapping scheme hPKW. The difference to the PKW games is *highlighted*.

the GGM tree does not require a fixed (maximum) depth, leading to a more versatile and dynamic primitive. Because Eval now allows the evaluation of elements that were previously considered labels of “inner” nodes, an additional derivation step is necessary before the value can be returned to prevent trivial attacks. The construction would then use a length-tripling PRG  $G(s) = G_1(s) || G_2(s) || G_3(s)$ , the last third of which is only used as the final stage in the derivation:

$$F(sk, x) = G_3(G_{x_l}(G_{x_{l-1}}(\dots G_{x_2}(G_{x_1}(sk))))).$$

In practice, a key derivation step with an appropriate context string can be used for  $G_3$ .

### 4.2.3 Hierarchically puncturable key-wrapping (hPKW)

Based on the definition of PKW, we supply a definition for hierarchically puncturable key-wrapping (hPKW):

<p><b>Game <math>G_{\text{hPKW}}^{\text{int-ctxt}}(\mathcal{A})</math>:</b></p> <ol style="list-style-type: none"> <li>1 <math>\text{win} \leftarrow \text{false}; u \leftarrow 0</math></li> <li>2 <math>\mathcal{A}^{\text{NEW, WRAP, UNWRAP, PUNC}}()</math></li> <li>3 Return win</li> </ol> <p><b>NEW():</b></p> <ol style="list-style-type: none"> <li>4 <math>u++; sk_u \leftarrow \\$ \text{KeyGen}()</math></li> <li>5 <math>\mathcal{S}_{\text{THC},u}, \mathcal{S}_{T,u}, \mathcal{S}_{PT,u} \leftarrow \emptyset</math></li> </ol> <p><b>PUNC(<math>T</math>):</b></p> <ol style="list-style-type: none"> <li>6 <math>sk_i \leftarrow \text{Punc}(sk_i, T)</math></li> <li>7 <math>\mathcal{S}_{PT,i} \leftarrow \cup \{T\}</math></li> </ol>	<p><b>WRAP(<math>i, T, H, K</math>):</b></p> <ol style="list-style-type: none"> <li>8 If <math>T \in \mathcal{S}_{T,i}</math> then return <math>\perp</math></li> <li>9 <math>C \leftarrow \text{Wrap}(sk_i, T, H, K)</math></li> <li>10 If <math>C = \perp</math> then return <math>\perp</math></li> <li>11 <math>\mathcal{S}_{\text{THC},i} \leftarrow \cup \{(T, H, C)\}; \mathcal{S}_{T,i} \leftarrow \cup \{T\}</math></li> <li>12 Return <math>C</math></li> </ol> <p><b>UNWRAP(<math>i, T, H, C</math>):</b></p> <ol style="list-style-type: none"> <li>13 <math>K \leftarrow \text{Unwrap}(sk_i, T, H, C)</math></li> <li>14 If <math>K \neq \perp</math> and <math>((T, H, C) \notin \mathcal{S}_{\text{THC},i}</math> or <math>\exists T' \in \mathcal{S}_{PT,i}</math> s.t. <math>T' \preceq T</math>):</li> <li>15 <math>\text{win} \leftarrow \text{true}</math></li> <li>16 Return <math>K</math></li> </ol>
---	---

**Figure 4.5:** The ciphertext integrity game for hPKW. Differences to the game for PKW are highlighted.

**Definition 4.6** (hPKW scheme). *A hierarchically puncturable key-wrapping scheme consists of four algorithms  $\text{hPKW} = (\text{KeyGen}, \text{Wrap}, \text{Unwrap}, \text{Punc})$  with four associated sets: the secret-key space  $\mathcal{SK}$ , the tag space  $\mathcal{T}$  with relation  $\preceq$ , the header space  $\mathcal{H}$  and the wrap-key space  $\mathcal{K}$ .*

- Via  $sk \leftarrow \$ \text{KeyGen}()$ , the probabilistic key generation algorithm  $\text{KeyGen}$ , taking no input, outputs a secret key  $sk \in \mathcal{SK}$ .
- Via  $C/\perp \leftarrow \text{Wrap}(sk, T, H, K)$ , the deterministic wrapping algorithm  $\text{Wrap}$  on input a secret key  $sk \in \mathcal{SK}$ , a tag  $T \in \mathcal{T}$ , a header  $H \in \mathcal{H}$  and a key  $K \in \mathcal{K}$  outputs a ciphertext  $C \in \{0, 1\}^{\text{cl}(|K|)}$  or, to indicate failure,  $\perp$ .
- Via  $K/\perp \leftarrow \text{Unwrap}(sk, T, H, C)$ , the deterministic unwrapping algorithm  $\text{Unwrap}$  on input a secret key  $sk \in \mathcal{SK}$ , a tag  $T \in \mathcal{T}$ , a header  $H \in \mathcal{H}$  and a ciphertext  $C \in \{0, 1\}^*$  returns a key  $K \in \mathcal{K}$  or, to indicate failure,  $\perp$ .
- Via  $sk' \leftarrow \text{Punc}(sk, T)$ , the deterministic puncturing algorithm  $\text{Punc}$  on input of a secret key  $sk \in \mathcal{SK}$  and a tag  $T \in \mathcal{T}$  returns a potentially updated secret key  $sk' \in \mathcal{SK}$ .

For correctness of an hPKW, we require that intuitively, a wrapped key can always be unwrapped, unless the tag under which it was wrapped, or a tag which is a prefix to the tag under which it was wrapped has been punctured. Formally, we require that for all  $T \in \mathcal{T}$ ,  $H \in \mathcal{H}$  and all tuples  $\bar{T}_1, \bar{T}_2 \in \mathcal{T}^*$  where  $T' \not\preceq T$  for any  $T' \in \bar{T}_1 \cup \bar{T}_2$ ,

$$\Pr \left[ \text{Unwrap}(sk_{\setminus \bar{T}_1}, T, H, \text{Wrap}(sk_{\setminus \bar{T}_2}, T, H, K)) = K \mid sk \leftarrow \$ \text{KeyGen}() \right] = 1.$$

We present the adapted security games in Figure 4.4 (forward indistinguishability) and in Figure 4.5 (ciphertext integrity). Like hPPRF, the advantage terms stay the same for hPKW as for PKW (Chapter 2.2.4), so they will not be repeated here.

#### 4.2.4 hPKW from AEAD and hPPRF

We can construct an hPKW in the same manner as the construction of PKW from a PPRF and AEAD in [16], by replacing the PPRF with a hPPRF (see Figure 4.6): the hPPRF enables the hierarchical punctures.

<u>hPKW[hPPRF, AEAD] :</u>	<u>hPKW.Unwrap(<math>sk_p, T, H, C</math>):</u>
<u>hPKW.KeyGen():</u>	5 $sk_a \leftarrow \text{hPPRF.Eval}(sk_p, T)$
1 Return hPPRF.KeyGen()	6 $K \leftarrow \text{AEAD.Dec}(sk_a, N_0, H, C)$
<u>hPKW.Wrap(<math>sk_p, T, H, K</math>):</u>	7 Return $K$
2 $sk_a \leftarrow \text{hPPRF.Eval}(sk_p, T)$	<u>hPKW.Punc(<math>sk_p, T</math>):</u>
3 $C \leftarrow \text{AEAD.Enc}(sk_a, N_0, H, K)$	8 $sk'_p \leftarrow \text{hPPRF.Punc}(sk_p, T)$
4 Return $C$	9 Return $sk'_p$

**Figure 4.6:** Construction of hPKW from hPPRF and AEAD. The nonce  $N_0$  is a constant of value 0.

#### 4.2.5 HPFS construction

The hPKW that is defined above can be used to instantiate HPFS. To build HPFS, the tag space with relation  $\preceq$  is mapped to the path space of the HPFS. In practice, the tag space of the hPKW could be bitstrings, with the relation  $\preceq$  being the prefix-relation: for files  $F_1, F_2$  stored in the system using  $\text{EncFile}(sk, F_i, P_i)$  with the paths  $P_1, P_2$  having a common parent path  $P_p$  there is a PKW tag  $t_{P_p}$  that can be punctured, resulting in both files being shredded. To correctly map paths to hPKW tags, a mapping is made, which allows looking up the tags for previously seen paths.

The construction (see Figure 4.7), remains very similar to the PFS<sup>+</sup> construction from PKW and AEAD (Figure 3.14), with hierarchical punctures relying on the modified hPKW scheme. The mapping from paths to hPKW tags is made dynamically and depends on what other files have been introduced to the system. A set of associative maps record which tags have been assigned to which paths, so that when a path is shredded, the correct hPKW tag can be retrieved to be punctured.

<p><u>Setup()</u> :</p> <ol style="list-style-type: none"> <li>1 <math>T[\cdot] \leftarrow \perp</math></li> <li>2 <math>T[\epsilon] \leftarrow \epsilon</math></li> <li>3 <math>C[\cdot] \leftarrow 0</math></li> <li>4 <math>H[\cdot] \leftarrow \perp</math></li> <li>5 <math>sk_{hPKW} \leftarrow \\$ hPKW.KeyGen()</math></li> <li>6 Return <math>(sk_{hPKW}, T, C, H)</math></li> </ol> <p><u>EncFile(<math>sk, F, P</math>):</u></p> <ol style="list-style-type: none"> <li>7 <math>K \leftarrow \\$ \{0, 1\}^k</math>;</li> <li>8 <math>(sk_{hPKW}, T, C, H) \leftarrow sk</math></li> <li>9 <math>(t, T, C) \leftarrow getOrGenTag(P, T, C)</math></li> <li>10 <math>h \leftarrow hPKW.Wrap(sk_{hPKW}, t, \epsilon, K)</math></li> <li>11 If <math>h = \perp</math> :</li> <li>12     Return <math>\perp</math></li> <li>13 <math>H[h] \leftarrow P</math></li> <li>14 <math>N \leftarrow \\$ \{0, 1\}^n</math></li> <li>15 <math>C \leftarrow AEAD.Enc(K, N, \epsilon, F)</math></li> <li>16 <math>sk' \leftarrow (sk_{hPKW}, T, C, H)</math></li> <li>17 Return <math>(sk', h_f, N    C)</math></li> </ol> <p><u>getOrGenTag(<math>P, T, C</math>):</u></p> <ol style="list-style-type: none"> <li>18 <math>p_0    \dots    p_d \leftarrow P</math></li> <li>19 For <math>i \leftarrow 0 \dots d</math> :</li> <li>20     <math>P_i \leftarrow p_0    \dots    p_i</math></li> <li>21     If <math>T[P_i] = \perp</math> :</li> <li>22         <math>t \leftarrow T[P_{i-1}]</math> // <math>P_{-1} = \epsilon</math></li> <li>23         <math>c \leftarrow C[P_{i-1}]</math></li> <li>24         <math>T[P_i] \leftarrow t    [c]_{l_t}</math> // <math>[c]_{l_t}</math> makes fixed-length encoding</li> <li>25         <math>C[P_{i-1}] \leftarrow c + 1</math></li> <li>26 <math>t \leftarrow T[P]</math></li> <li>27 Return <math>(t, T, C)</math></li> </ol>	<p><u>DecFile(<math>sk, h, N    C</math>):</u></p> <ol style="list-style-type: none"> <li>28 <math>(sk_{hPKW}, T, C, H) \leftarrow sk</math></li> <li>29 <math>P \leftarrow H[h]</math></li> <li>30 <math>t \leftarrow T[P]</math></li> <li>31 <math>K \leftarrow hPKW.Unwrap(sk_{hPKW}, t, \epsilon, h)</math></li> <li>32 If <math>K = \perp</math> then return <math>\perp</math></li> <li>33 <math>F \leftarrow AEAD.Dec(K, N, \epsilon, C)</math></li> <li>34 Return <math>F</math></li> </ol> <p><u>Shred(<math>sk, P</math>):</u></p> <ol style="list-style-type: none"> <li>35 <math>(sk_{hPKW}, T, C, H) \leftarrow sk</math></li> <li>36 <math>t \leftarrow T[P]</math></li> <li>37 If <math>t \neq \perp</math> :</li> <li>38     <math>sk_{hPKW} \leftarrow PKW.Punc(sk_{hPKW}, t)</math></li> <li>39 <math>T[P^*] \leftarrow \perp</math></li> <li>40 <math>C[P^*] \leftarrow 0</math></li> <li>41 <math>H[., P^*] \leftarrow \perp</math></li> <li>42 Return <math>(sk_{hPKW}, T, C, H)</math></li> </ol> <p><u>RotKey(<math>sk_{old}, (h_1, \dots, h_l)</math>):</u></p> <ol style="list-style-type: none"> <li>43 <math>(sk_{hPKW}, T, C, H) \leftarrow sk</math></li> <li>44 <math>sk'_{hPKW} \leftarrow \\$ hPKW.KeyGen()</math></li> <li>45 <math>T'[\cdot] \leftarrow \perp</math>; <math>T'[\epsilon] \leftarrow \epsilon</math></li> <li>46 <math>C'[\cdot] \leftarrow 0</math>; <math>H'[\cdot] \leftarrow \perp</math></li> <li>47 For <math>i \leftarrow 0</math> to <math>l</math> do:</li> <li>48     <math>P \leftarrow H[h_i]</math></li> <li>49     <math>t \leftarrow T[P]</math></li> <li>50     <math>K_i \leftarrow PKW.Unwrap(sk_{hPKW}, t, \epsilon, h_i)</math></li> <li>51     If <math>K_i = \perp</math> then return <math>(sk, \perp)</math></li> <li>52     <math>(t', T', C') \leftarrow getOrGenTag(P, T', C')</math></li> <li>53     <math>h'_i \leftarrow PKW.Wrap(sk'_{hPKW}, t', \epsilon, K_i)</math></li> <li>54     If <math>h'_i = \perp</math> then return <math>(sk, \perp)</math></li> <li>55     <math>H'[h'_i] \leftarrow P</math></li> <li>56 Return <math>(sk'_{hPKW}, T', C', H')</math></li> </ol>
--	---

**Figure 4.7:** HPFS construction from composition of hPKW scheme hPKW and AEAD scheme AEAD, using counters as tags (the map  $C$  tracks the current value of the counter for each path). Integer counters are mapped to fixed-length bitstrings of length  $l_t$ . The key-wrap space of the hPKW scheme is  $\{0, 1\}^k$ . The AEAD scheme has nonce space  $\{0, 1\}^n$  and key space  $\{0, 1\}^k$ . The instantiated HPFS scheme therefore has header space  $\mathcal{H} = \{0, 1\}^{PKW.cl(k)}$  and an associated ciphertext length of  $PFS.cl(|F|) = AEAD.cl(|F|) + n$ .  $T$ ,  $C$  and  $H$  are associative maps which map paths of files to their hPKW tags, directory paths to their counter value, and headers with their path, respectively. The notation  $T[P^*] \leftarrow \perp$  means that entries in  $T$  of which the key is prefixed by  $P$  are removed.  $H[., P^*] \leftarrow \perp$  denotes that entries of which the value is prefixed by  $P$  are removed.

Due to the complexity introduced by HPFS keeping internal state about the path space, we explain the function of the construction in detail. The internal state consists of three associative maps. The map  $T$  stores which paths are associated to which tags. The map  $C$  is used to store the counters for directory paths. In directories, files are assigned consecutive tags, like in ctrPFS. The counters are used to assign these tags. The third associative map,  $H$ , associates headers and paths (so that the path corresponding to a header can be looked up).

The setup algorithm `Setup` initializes the associative maps. The counter map  $C$  is initialized with all-zero values. The header map is initialized empty, except for the root path, which is associated with an empty tag. The headers map  $H$  is initialized empty. The algorithm also generates a fresh hPKW key. The tuple  $(sk_{\text{hPKW}}, T, C, H)$  makes up the secret key.

We explain the algorithm `getOrGenTag`, which is responsible for generating or retrieving the mapping of a path to an hPKW tag. On input the path  $P$ , the tag lookup table  $T$ , and the counter table  $C$ , it splits the path into the names of the path space nodes. For each path  $P_i$  (line 21) leading to  $P$  (starting at the root), it checks if there is an entry in  $T$  for it. If not, the tag  $t$  of the parent path and the counter value  $c$  of the parent path are retrieved from  $T$  and  $C$ , respectively. To produce the tag for the path  $P_i$ , the counter  $c$  is encoded into a fixed-length bitstring and appended to the tag of the parent path. The tag is then stored in  $T$  under  $P_i$ . After completion of the algorithm, the path  $P$  and all its ancestors have tags assigned to them stored in  $T$ .

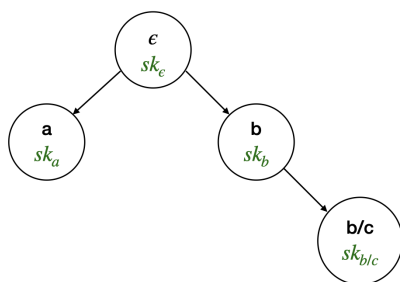
The encryption algorithm `EncFile` functions virtually the same as the algorithm of the same name in PFS<sup>+</sup>, except for the tag, which is generated using the `getOrGenTag` algorithm explained above. A DEK  $K$  is generated and wrapped under hPKW secret key, producing the header  $h$ , which is stored with the path in the headers map  $H$ . The DEK is used to AEAD-encrypt the file, and the secret key is updated with the modified maps.

For decryption, the path corresponding to the header must be retrieved from  $H$ . It is then used to obtain the hPKW tag  $t$ , which in turn is used to unwrap the header. The recovered DEK  $K$  can then decrypt the ciphertext, revealing the file.

Shredding a path  $P$  is done by puncturing the hPKW key on the tag corresponding to the given path. The tag is obtained from the tag map  $T$ . After the puncture, the associative maps  $T$ ,  $C$ , and  $H$  are updated by removing any paths to which the path  $P$  is a prefix. These updates to the associative maps are a clean-up operation and are not needed for security.

The key rotation algorithm `RotKey` generates a fresh hPKW key and initializes a new set of associative maps (in the same way as in `Setup`). For all the





**Figure 4.8:** Example of directory structure and associated PKW keys for the PKW-per-directory approach.

given headers, it then retrieves the corresponding path and tag from the current maps, and unwraps the header using the current hPKW key to reveal the DEK. Then a new tag is generated (using the new associative maps), and the DEK is wrapped using the fresh hPKW key and the new tag. The map that stores the associations of headers and paths,  $H$ , is also updated.

**Security of the construction** As is in the PFS construction, forward security of shredded files is provided by the forward security of hPKW, whereas ciphertext integrity and indistinguishability are provided by the AEAD scheme.

### 4.3 Constructing HPFS from PKW

As an alternative construction of HPFS, we provide an instantiation from a PKW scheme and an AEAD scheme in Figure 4.9, which we call the PKW-per-directory approach. The underlying idea is to dynamically create a new PKW secret key for each new directory introduced to the system (see the example in Figure 4.8). The PKW scheme PKW and AEAD scheme AEAD used in the construction have the following parameters: The key-wrap space of the PKW scheme is  $\{0, 1\}^k$ , the tag space is  $\{0, 1\}^t$ . The AEAD scheme has nonce space  $\{0, 1\}^n$  and key space  $\{0, 1\}^k$ . The instantiated HPFS scheme has header space  $\mathcal{H} = \{0, 1\}^{\text{PKW.cl}(k)}$  and an associated ciphertext length of  $\text{HPFS.cl}(|F|) = \text{AEAD.cl}(|F|) + n$ . The construction maintains the associative map  $T$ , to map file paths to PKW tags, and  $H$ , to map headers to the paths for which they were created.

For key rotation in PFS, all headers must be supplied, or else the files associated with the omitted headers would effectively be deleted.

### Duplicate PKW tags

Since in all directories the PKW tags belong to the same space, there will be duplicates at the HPFS level. The associative map  $T$  records a mapping from paths (which uniquely identify a file) to the PKW tag used for wrapping. Duplicate tags are not problematic for security, since each directory has a distinct key.

#### 4.3.1 Detailed explanation of the construction

The construction is provided in Figure 4.9. Here, we elaborate on the function of the construction in detail:

To start with, consider the Setup algorithm. It initializes the associative map  $T$ , which maps local file paths to their PKW tag. The header map  $H$  (mapping headers to paths) is also initialized. Its purpose is to make it possible to look up a path from a header, which is needed for key rotation. The setup algorithm also sets up the secret key collection  $SK$ , which is also an associative map, associating (directory) paths to PKW keys, and initializes a PKW key for the root path. The tuple  $(SK, T, H)$  makes up the secret key  $sk$ .

To encrypt a file, the secret key (PKW) of the directory containing the file is retrieved. If none exists, a new one is created and stored in the PFS secret key collection  $SK$ , alongside the directory tag counter  $ctr$ , initialized at 0. The secret key is then used to wrap a newly generated data encryption key ( $K_f$ ), with the tag given by the tag counter value  $ctr$  (embedded in a bitstring of length  $t$ ), to produce the header  $h_f$ . The data encryption key is used to encrypt the file using the AEAD scheme.

The decryption of a file involves first obtaining the PKW tag of the file from the map  $T$  and the secret directory key from the secret key collection  $SK$ . The directory key is then used to unwrap the data encryption key, which is finally used to decrypt the AEAD ciphertext.

Key rotation has to rotate each PKW secret key in  $SK$ . So, a new PKW secret key is generated for each directory, and for all headers passed as arguments, the corresponding PKW directory key is retrieved to unwrap the old header and then rewrap it with the fresh directory key. The PKW tag may change in the process: the current number of files in a directory may be smaller than the directory counter value (because of previous shred operations). Therefore, during key rotation, new tags are given, to use all possible non-negative (consecutive) integers. In fact, the reason for the key rotation may be that the tag counter has reached its maximum value. By rotating keys, previously punctured tags can be reused, freeing up space in the “upper end” of the PKW tag space.

<p><u>Setup()</u>:</p> <pre> 1 T[.] ← ⊥; H[.] ← ⊥; SK[.] ← ⊥ 2 sk_ε ← PKW.KeyGen() 3 SK[ε] ← (sk_ε, 0) 4 Return (SK, T, H)  <u>EncFile(sk, F, P_f)</u>: 5 K_f ← \$ {0,1}^k; P_p ← parent(P_f) 6 (SK, T, H) ← sk 7 If SK[P_p] = ⊥: 8   sk_dir ← PKW.KeyGen() 9   SK[P_p] ← (sk_dir, 0) 10 (sk_dir, ctr) ← SK[P_p] 11 id_f ← [ctr]_t 12 h_f ← PKW.Wrap(sk_dir, id_f, ε, K_f) 13 If h_f = ⊥: 14   Return ⊥ 15 SK[P_p] ← (sk_dir, ctr + 1) 16 N ← \$ {0,1}^n 17 C ← AEAD.Enc(K_f, N, ε, F) 18 T[P_f] ← id_f 19 H[h_f] ← P_f 20 sk' ← (SK, T, H) 21 Return (sk', h_f, N    C)  <u>DecFile(sk, h_f, N    C)</u>: 22 (SK, T, H) ← sk 23 P_f ← H[h_f] 24 If h_f = ⊥ then return ⊥ 25 id_f ← T[P_f] 26 If id_f = ⊥ then return ⊥ 27 (sk_dir, -) ← SK[parent(P)] 28 K ← PKW.Unwrap(sk_dir, id_f, ε, h_f) 29 If K = ⊥ then return ⊥ 30 F ← AEAD.Dec(K, N, ε, C) 31 Return F </pre>	<p><u>RotKey(sk, ((h_0, ..., h_l))):</u></p> <pre> 32 (SK, T, H) ← sk 33 SK'[.] ← ⊥ 34 T'[.] ← ⊥ 35 H'[.] ← ⊥ 36 For i ← 0 to l do: 37   P ← H[h_i] 38   id_f ← T[P] 39   If SK'[parent(P)] = ⊥: 40     sk_dir ← PKW.KeyGen() 41     SK'[parent(P)] ← (sk_dir, 0) 42   (sk_dir, -) ← SK[parent(P)] 43   (sk'_dir, ctr') ← SK'[parent(P)] 44   K_i ← PKW.Unwrap(sk_dir, id_f, ε, h_i) 45   If K_i = ⊥ then return (sk, ⊥) 46   id'_f ← [ctr']_t 47   h'_i ← PKW.Wrap(sk'_dir, id'_f, ε, K_i) 48   If h'_i = ⊥ then return (sk, ⊥) 49   SK'[parent(P_i)] ← (sk'_dir, ctr' + 1) 50   T'[P] ← id'_f 51   H'[h'_i] ← P 52 sk' ← (SK', T', H') 53 Return (sk', (h'_0, ..., h'_l))  <u>Shred(sk, P)</u>: 54 (SK, T, H) ← sk 55 id_f ← T[P] 56 If id_f ≠ ⊥: 57   (sk_dir, ctr) ← SK[P] 58   sk_dir ← PKW.Punc(sk_dir, id_f) 59   SK[P] ← (sk_dir, ctr) 60 SK[P*] ← ⊥ delete directory keys 61 T[P*] ← ⊥ 62 H[., P*] ← ⊥ 63 Return sk </pre>
---	---

**Figure 4.9:** HPFS construction from composition of PKW scheme PKW and AEAD scheme AEAD. The key-wrap space of the PKW scheme is  $\{0,1\}^k$ , the tag space is  $\{0,1\}^t$ . The AEAD scheme has nonce space  $\{0,1\}^n$  and key space  $\{0,1\}^k$ . The notation  $T[P*] \leftarrow \perp$  denotes that all entries in the associative map  $sk$  which are indexed by  $P$  or whose index is a path that starts with  $P$  are set to  $\perp$ . The slight abuse of notation  $H[., P*] \leftarrow \perp$  denotes that entries in  $H$  of which the value is prefixed by  $P$  are removed.

### 4.3.2 Security of the construction

As in the previous construction, the confidentiality notion that is achieved is `finds-rcpa`, the integrity notion is `int-ctxt`. We give a brief informal argument for the realization of these notions. Consider the point at which the adversary decides to corrupt. Imagine that it has stored some ciphertexts and headers of files which at this point have already been shredded. Since in `Shred`, the keying material that allows file recovery is always deleted, the adversary cannot decrypt the shredded files: if the parent directory was shredded, the PKW key has been removed; if a single file was shredded, the PKW key of the directory has been punctured and so the PKW scheme provides forward secrecy. Confidentiality of the headers and ciphertexts is provided by PKW and AEAD, respectively. The integrity of ciphertexts and headers is again provided by AEAD and PKW, respectively.

### 4.3.3 Single-directory key rotation

In this construction, every directory has an associated PKW secret key. Key rotation is defined as an operation that rotates all the keying material, but, in this construction, a finer-grained, directory-based approach may be taken: if key rotation is necessary for a given directory (because many punctures have been performed), it could be performed for only that directory. This can greatly reduce network traffic (fewer headers must be down-/uploaded) and therefore also the time for the operation to finish. This optimization, of course, does not fulfil what is needed for post-compromise security. To recover from a corruption, a full key rotation is needed. However, it may merit consideration in practical settings.

## 4.4 Implementation

The two constructions are implemented based on the PKW library [26] mentioned before. In the implementations, directories can contain up to  $2^{16}$  files and subdirectories (in the approach with a PKW key per directory, subdirectories are not counted). The number  $2^{16} = 65536$  is one more than the maximum number of files supported per directory in FAT32. Although modern file systems usually support more files per directory, it is a usable limit for the purposes of this project.

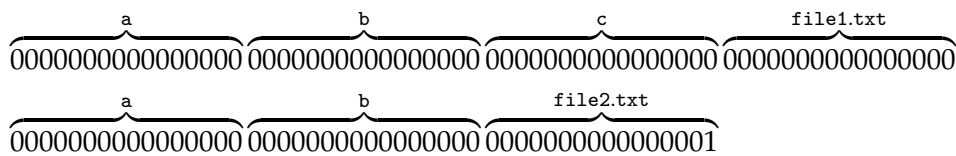
### 4.4.1 Extension of the PKW library

The PKW library [26] required some changes to allow the evaluation and puncturing of tags of any length, or, in other words, to implement `hPKW`. These changes were in the data structure used for tags (a fixed-length array

was replaced with a variable-length array) and in the puncture algorithm, which now allows punctures on tags of any length.

#### 4.4.2 hPKW

With the hPKW approach, little changes in the implementation. Only the assignment of hPKW tags had to be implemented, such that parts of tags would correspond to directories. Concretely, 16 bits are used per directory level and all known directories are put into a lookup table. If there is a file with path `a/b/c/file1.txt` in the system, a new file with path `a/b/file2.txt` will share the first 32 bits of the tag with the first one. Specifically, the 32-bit tag corresponding to the path `a/b/`. In the following, we show a possible assignment of hPKW tags for these two paths.



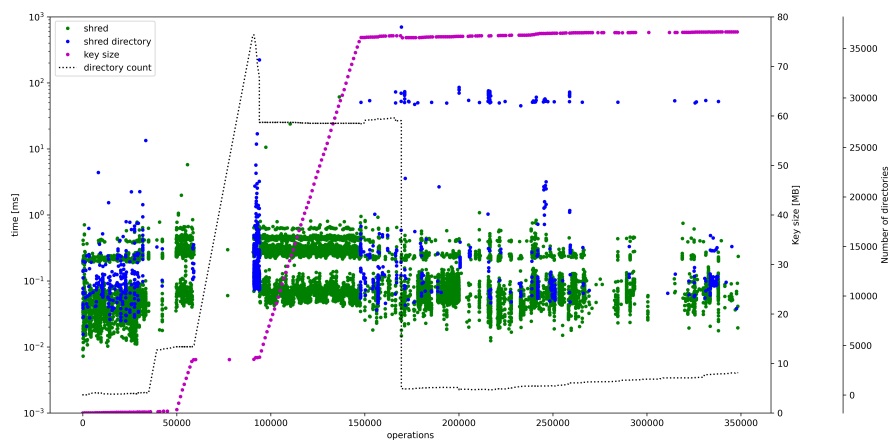
#### 4.4.3 PKW-per-directory

In this approach, for each new directory encountered (during encryption), a new PKW key is added to a lookup table, which maps known directories to PKW keys. On deletion of a directory, the corresponding entry in the secret key lookup table SK is simply deleted. The entries to which the directory is a prefix are also deleted. The PKW schemes use tag sizes of 16 bits, so that the two approaches are comparable.

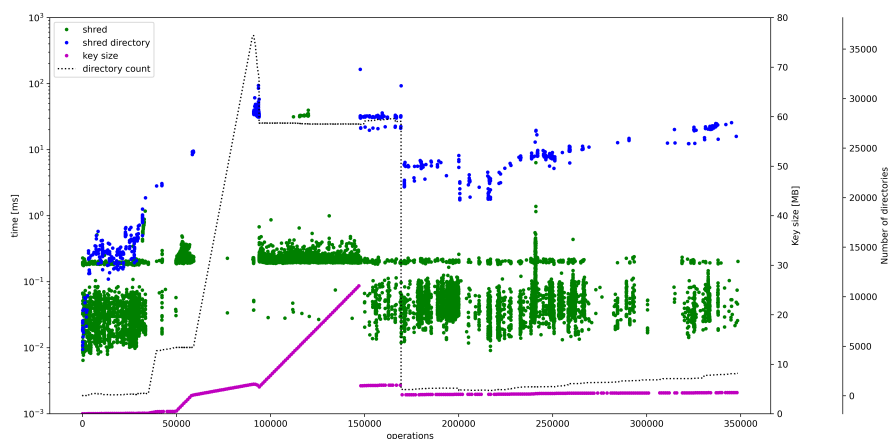
## 4.5 Evaluation

We wanted to test the effect that the inclusion of paths has on key size and time efficiency, as it seemed probable that shredding whole directories instead of individual files would lead to improvements. Again, we focus on the GitHub datasets (Section 3.1.4). Initially, the datasets only included individual file deletions, even when an entire directory was deleted. To identify sequences of file deletions that represent a directory deletion, we simulated the history in a virtual file system. We detected the point at which a directory becomes empty after a series of deletions and then replaced the sequence of file deletions with a directory deletion in a rewritten version of the dataset. Below, we present the GitHub histories for the FreeCodeCamp repository, run on both implementations.

#### 4. PFS WITH FILE HIERARCHY



**Figure 4.10:** FreeCodeCamp GitHub history using the hPKW approach. File deletions are shown in green, directory deletions in blue. For the deletions, the y-position indicates the time used per operation. The key size is shown in purple, and the black dotted line shows the number of directories in the system.



**Figure 4.11:** FreeCodeCamp GitHub history using the PKW-per-directory approach. File deletions are shown in green, directory deletions in blue. For the deletions, the y-position indicates the time used per operation. The key size is shown in purple, and the black dotted line shows the number of directories in the system.

What becomes clear in the direct comparison of the two figures (4.10, 4.11), is that they perform worse than ctrPFS (Figure 3.16) in both the key size and the time it takes to perform a shred operation. In both approaches, directory deletions can take a very long time (note that the y-axis for time is in a logarithmic scale). This may be due to the many lookup tables that have to be kept up-to-date.

This is indicative of a drawback of HPFS: the system has a lot of state it needs to track because the relationship between the local file paths is mapped in the scheme, allowing hierarchical shredding operations.

It would seem that the mappings made by these constructions are not very efficient in terms of the resulting secret key size, although the per-directory PKW approach performs better than the hPKW approach. Against expectations, the key size rarely decreases for hPKW, but rather increases, even as the number of directories shrinks due to deletions. The reason for this is not obvious, but it may be due to a characteristic of these benchmarks. When directory deletions occur, often there were no previous deletions (shreds) in that directory. For the hPPRF key, this means that the subtree that is punctured because of the directory removal was not present (that is, there are no nodes in the key that belong to the subtree) in the key beforehand. And so, no reduction in key size can be achieved with the directory deletion; rather, the operations add to the key size. However, we contend that the file access patterns extracted from the GitHub repositories are a use case that is not unrealistic for HPFS, and so the underwhelming performance of the two implementations cannot be ignored.





## Chapter 5

---

# Discussion

---

As seen in the evaluation sections of Chapters 3 and 4, performance both in time per shred operation and key size was best for the PFS construction in which PKW tags were increasing (ctrPFS). For HPFS, even though intuitively it seemed as though utilizing the hierarchical information would perform well, performance was actually worse, both for key size and time efficiency. Below, we show a summary of the performance on the React dataset.

	original PFS	ctrPFS	hPKW	PKW-per-directory
final key size	300 MB	150 KB	630 KB	208 KB
total time usage	102 s	4.3 s	5.9 s	4.3 s

**Table 5.1:** *GitHub history metrics for the React history, which contains 4250 file deletions.*

In the larger FreeCodeCamp dataset, which contains more deletions, the difference in performance becomes more apparent. The original PFS construction is not shown because it was not evaluated for the FreeCodeCamp dataset.

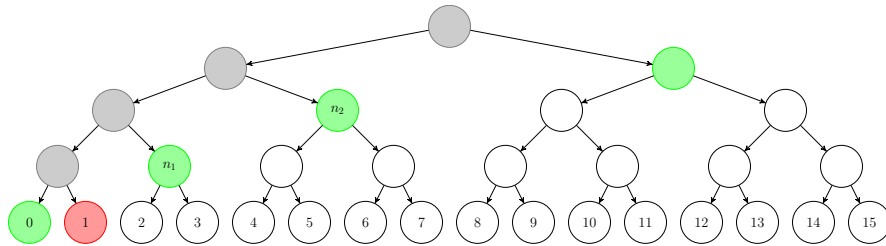
	ctrPFS	hPKW	PKW-per-directory
final key size	230 KB	77 MB	4.5 MB
total time usage	47 s	90 s	415 s

**Table 5.2:** *GitHub history metrics for the FreeCodeCamp history, which contains 55000 file deletions.*

Clearly, ctrPFS is the most efficient of the presented implementations, especially as the size of the datasets increases.

For the key size, this is because ctrPFS is already quite optimal for shred

operations. There is an upper bound for the increase in size per shred operation (in terms of the number of files in the system), and deletions of sequentially added files lead to a reduction in key size. This is not necessarily the case for HPFS: in the PKW-per-directory approach, having many directories in which only a single shred operation occurred is costly (in size), since for each directory 15 key-nodes must be stored. This is because a single puncture in a GGM PPRF results in a key size equal to the tag length minus 1. For the hPKW approach, there is a similar issue for chains of empty directories, of which the last one has had some files shredded in it. This leads to a key with “sparse punctures”, meaning long chains of nodes. Figure 5.2 shows a part of a hPPRF key, with these characteristic long chains of nodes. As a comparison, a part of the PPRF key resulting from using PFS with increasing PKW tags (ctrPFS) for the same dataset is shown in Figure 5.3. Here, the tree is much more balanced because the tags are used in sequential order and without gaps. These figures show the key nodes (in red) and the nodes leading to them (in grey, present to help visualize the key structure).

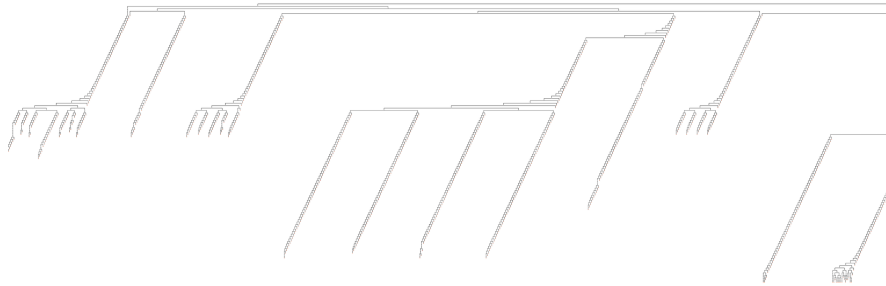


**Figure 5.1:** PPRF key with one punctured element (labelled 1). If nodes 2 and 3 are removed as well, the key size is reduced by one node. Further removal of nodes 4 to 7 reduces the key size by another node.

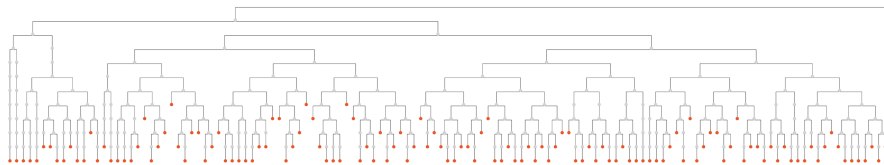
To visualize this point, Figure 5.1 shows the PPRF key for the PFS<sup>+</sup> construction with PKW tag length 4. Imagine that it was used to store 9 files (nodes 0 to 8), of which one (corresponding to the red node) was shredded. Shredding files corresponding to nodes 2 and 3 would lead to the removal of the green node  $n_1$ , and therefore to a reduction in key size.

By introducing the hierarchy of the file system into PFS, we also introduced inefficiencies which distanced it from the optimization we initially made in Chapter 3 when increasing PKW tags were introduced (ctrPFS). The HPFS constructions, then, while better than the original PFS construction, cannot come close to ctrPFS in key size because:

- For the hPKW approach, the tags which are assigned to files are only consecutive within a directory. Shredding files across different directories is unlikely to lead to a decrease in key size (but it does for ctrPFS, as seen in the evaluations). Furthermore, shredding a directory usu-



**Figure 5.2:** Left-most part of the hPPRF secret key structure produced by the Guava GitHub dataset run on HPFS[hPKW, AEAD]. The HPFS scheme employs 10 bits of hPKW tag per hierarchy level for improved visualization. Note the long chains of nodes which do not flare out towards the bottom, indicative of directories in which a single file was shredded.



**Figure 5.3:** Left-most part of the PPRF secret key structure produced by the Guava GitHub dataset run on PFS[PKW, AEAD]. Key nodes are shown in red. The PKW scheme uses tag space  $\{0, 1\}^{128}$ .

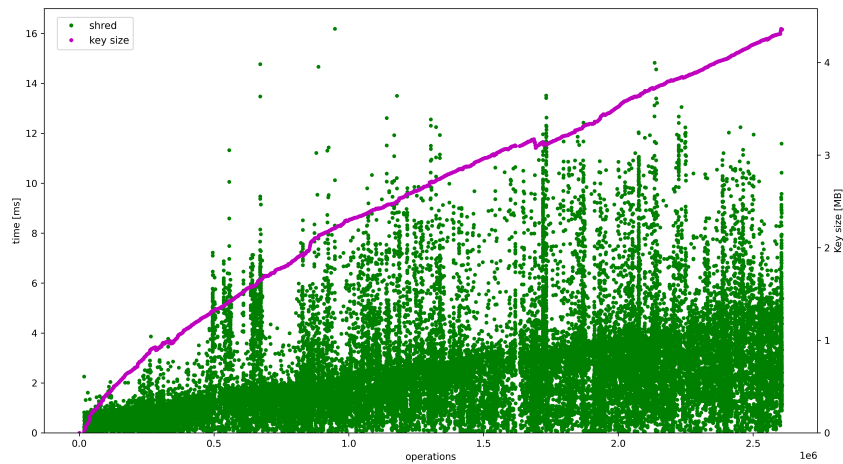
ally leads to an increase in key size, since the underlying operation is a puncture on a hPKW tag.

- For the PKW-per-directory approach, essentially, multiple ctrPFS instances exist in parallel (one per directory). Therefore, when no files are shredded, the key size is a multiple of the key size in ctrPFS. The first shred operation in ctrPFS incurs a higher cost than later shreds (a chain of nodes is generated by the puncture algorithm in the underlying PPRF (see the evaluation section for PFS<sup>+</sup>)). In the PKW-per-directory approach, this cost is incurred multiple times, once for each directory in which at least one file is shredded.

But the hypothesis for better efficiency was not only about the key size, but also about time: are hierarchical shreds more time-efficient than shredding all the contained files? By closer inspection of the mean execution times for directory delete operations, we conclude that this is not the case. In the FreeCodeCamp GitHub dataset there are 99033 delete operations, which, after extracting the directory delete operations, correspond to 47232 file deletions and 3768 directory deletions. Therefore, on average, close to 14 files are deleted per directory deletion. By comparing the average execution times (2 milliseconds for a directory shred in (the faster hPKW-based) HPFS vs.

## 5. DISCUSSION

---



**Figure 5.4:** Full Linux kernel GitHub commit history (18 years) in ctrPFS.

70 microseconds for a file shred in PFS), there appears to be no upside to using HPFS because on average performing the 14 deletions is faster than performing the hierarchical shredding.

To show the efficiency of ctrPFS, we provide the GitHub history of the Linux kernel dataset in Figure 5.4. The key size after replaying commits of 18 years only grows to about 4.2 MB, and the execution times for shred operations remain usable. However, with monthly or even yearly key rotation, the key size could be kept much smaller and the execution times would stay more consistent.

# Conclusion

---

We presented forward-secure cloud storage in the form of PFS, as described by [16], and showed evaluations for our implementation of it. Furthermore, we presented an optimization of the syntax (PFS<sup>+</sup>) and its construction (ctrPFS) that greatly improved the efficiency of the implementation. We extended the syntax of PFS to also include paths (HPFS), and presented two possible constructions. Moreover, we laid out the limitations of the PFS model, and explained that the remote storage of encrypted data is not captured by the security notions. We provided constructions and evaluations for PFS and HPFS and showed that the implementations of HPFS are less efficient. For the two constructions of HPFS, we explained that they are less efficient than the optimized construction of PFS, ctrPFS, because ctrPFS already uses the tree structure of the underlying PPRF optimally. Our implementation ctrPFS shows that forward-secure cloud storage is possible with an acceptable overhead in time and memory consumption.

### 6.1 Future work

In this thesis, we only used the cloud for the storage of encrypted files and headers. Recently, work has been done on searchable encryption [13, 14], which would allow outsourcing the lookup tables to the cloud as well, further reducing the memory footprint of a local client. Work has also been done which aims at hiding metadata of encrypted files, specifically also the file size [11]. The composition of forward-secure cloud storage with searchable encryption for file lookup and more privacy-preserving file storage could be an interesting avenue of exploration. It would also provide the opportunity to investigate a unified security model in which the cloud is the adversary under consideration, as in this work the PFS adversary has much control over the actions (it can add and shred files, in addition to corrupting the key). In this work, key delegation [24] was not considered. However, file

## 6. CONCLUSION

---

sharing, which is a functionality users typically expect from cloud storage, is only possible by sharing the entire secret key. Exploring the intricacies of key sharing (delegation) while maintaining the forward-security properties of PFS is another promising area of study.

---

## Bibliography

---

- [1] *Xplain Hack: Federal Council Commissions a Policy Strategy Crisis Team on Data Leaks*. [https://www.ncsc.admin.ch/ncsc/en/home/aktuell/im-fokus/2023/xplain\\_3.html](https://www.ncsc.admin.ch/ncsc/en/home/aktuell/im-fokus/2023/xplain_3.html). (Visited on July 26, 2023).
- [2] Johana Bhuiyan. “How Can US Law Enforcement Agencies Access Your Data? Let’s Count the Ways”. In: *The Guardian. Technology* (Apr. 4, 2022). ISSN: 0261-3077. <https://www.theguardian.com/technology/2022/apr/04/us-law-enforcement-agencies-access-your-data-apple-meta>. (Visited on July 26, 2023).
- [3] *Boxcryptor Security for Your Cloud*. <https://www.boxcryptor.com/en/>. (Visited on Jan. 24, 2023).
- [4] MEGA. MEGA. <https://mega.io/storage>. (Visited on June 5, 2023).
- [5] *Nextcloud Files - Open Source File Sync & Share*. Nextcloud. <https://nextcloud.com/files/>. (Visited on June 5, 2023).
- [6] *iCloud Data Security Overview*. Apple Support. <https://support.apple.com/en-us/HT202303>. (Visited on July 30, 2023).
- [7] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. “MEGA: Malleable Encryption Goes Awry”. In: *IEEE S&P 2023*. 959. San Francisco, CA, 2022.
- [8] Anders P. K. Dalskov and Claudio Orlandi. “Can You Trust Your Encrypted Cloud? An Assessment of SpiderOakONE’s Security”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security. ASIACCS '18*. New York, NY, USA: Association for Computing Machinery, May 29, 2018, pp. 343–355. ISBN: 978-1-4503-5576-6. DOI: [10.1145/3196494.3196547](https://doi.org/10.1145/3196494.3196547).

- [9] Seny Kamara and Kristin Lauter. “Cryptographic Cloud Storage”. In: *Financial Cryptography and Data Security*. Ed. by Radu Sion et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 136–149. ISBN: 978-3-642-14992-4. DOI: [10.1007/978-3-642-14992-4\\_13](https://doi.org/10.1007/978-3-642-14992-4_13).
- [10] Jan Stanek et al. “A Secure Data Deduplication Scheme for Cloud Storage”. In: *Financial Cryptography and Data Security*. Ed. by Nicolas Christin and Reihaneh Safavi-Naini. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 99–118. ISBN: 978-3-662-45472-5. DOI: [10.1007/978-3-662-45472-5\\_8](https://doi.org/10.1007/978-3-662-45472-5_8).
- [11] Sebastian Messmer et al. “A Novel Cryptographic Framework for Cloud File Systems and CryFS, a Provably-Secure Construction”. In: *Data and Applications Security and Privacy XXXI*. Ed. by Giovanni Livraga and Sencun Zhu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 409–429. ISBN: 978-3-319-61176-1. DOI: [10.1007/978-3-319-61176-1\\_23](https://doi.org/10.1007/978-3-319-61176-1_23).
- [12] Haifeng Li et al. “Lattice-Based Privacy-Preserving and Forward-Secure Cloud Storage Public Auditing Scheme”. In: *IEEE Access* 8 (2020), pp. 86797–86809. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2020.2991579](https://doi.org/10.1109/ACCESS.2020.2991579).
- [13] Kee Sung Kim et al. “Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2017, pp. 1449–1463. ISBN: 978-1-4503-4946-8. DOI: [10.1145/3133956.3133970](https://doi.org/10.1145/3133956.3133970). (Visited on July 19, 2023).
- [14] Ming Zeng et al. “Forward Secure Public Key Encryption with Keyword Search for Outsourced Cloud Storage”. In: *IEEE Transactions on Cloud Computing* 10.1 (Jan. 2022), pp. 426–438. ISSN: 2168-7161. DOI: [10.1109/TCC.2019.2944367](https://doi.org/10.1109/TCC.2019.2944367).
- [15] Nirvan Tyagi et al. “Burnbox: Self-Revocable Encryption in a World of Compelled Access”. In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC’18. USA: USENIX Association, Aug. 15, 2018, pp. 445–461. ISBN: 978-1-931971-46-1.
- [16] Matilda Backendal, Felix Günther, and Kenneth G. Paterson. “Puncturable Key Wrapping and Its Applications”. In: *Advances in Cryptology – ASIACRYPT 2022*. Ed. by Shweta Agrawal and Dongdai Lin. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2022, pp. 651–681. ISBN: 978-3-031-22966-4. DOI: [10.1007/978-3-031-22966-4\\_22](https://doi.org/10.1007/978-3-031-22966-4_22).



- [17] Phillip Rogaway and Thomas Shrimpton. “A Provable-Security Treatment of the Key-Wrap Problem”. In: *Advances in Cryptology - EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 373–390. ISBN: 978-3-540-34546-6 978-3-540-34547-3. DOI: [10.1007/11761679\\_23](https://doi.org/10.1007/11761679_23). (Visited on Mar. 6, 2023).
- [18] Phillip Rogaway. “Authenticated-Encryption with Associated-Data”. In: *Proceedings of the ACM Conference on Computer and Communications Security* (2002), pp. 98–107. ISSN: 15437221. DOI: [10.1145/586110.586125](https://doi.org/10.1145/586110.586125).
- [19] Dan Boneh and Brent Waters. “Constrained Pseudorandom Functions and Their Applications”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8270 LNCS (PART 2 2013), pp. 280–300. ISSN: 03029743. DOI: [10.1007/978-3-642-42045-0\\_15/COVER/](https://doi.org/10.1007/978-3-642-42045-0_15/COVER/).
- [20] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. “Functional Signatures and Pseudorandom Functions”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8383 LNCS (2014), pp. 501–519. ISSN: 16113349. DOI: [10.1007/978-3-642-54631-0\\_29/COVER/](https://doi.org/10.1007/978-3-642-54631-0_29/COVER/).
- [21] Matilda Backendal, Felix Günther, and Kenneth G. Paterson. *Puncturable Key Wrapping and Its Applications*. 2022. <https://eprint.iacr.org/2022/1209>. (Visited on Jan. 23, 2023). preprint.
- [22] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography (Version 0.6)*. Jan. 2023. <https://toc.cryptobook.us/>. (Visited on Apr. 1, 2023).
- [23] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. “How to Construct Random Functions”. In: *Journal of the ACM (JACM)* 33.4 (Aug. 1986), pp. 792–807. ISSN: 1557735X. DOI: [10.1145/6490.6503](https://doi.org/10.1145/6490.6503).
- [24] Aggelos Kiayias et al. “Delegatable Pseudorandom Functions and Applications”. In: *Proceedings of the ACM Conference on Computer and Communications Security* (2013), pp. 669–683. ISSN: 15437221. DOI: [10.1145/2508859.2516668](https://doi.org/10.1145/2508859.2516668).
- [25] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. “Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements”. In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by Bart Preneel. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1807. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 259–274. ISBN: 978-3-540-67517-4 978-3-540-45539-4. DOI: [10.1007/3-540-45539-6\\_18](https://doi.org/10.1007/3-540-45539-6_18). (Visited on Apr. 6, 2023).

- [26] Younis Khalil. *Implementing a Puncturable Key Wrapping Library*. Semester Project. ETH Zurich, 2022. [https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/Implementing\\_a\\_Puncturable\\_Key\\_Wrapping\\_Library\\_pub.pdf](https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/Implementing_a_Puncturable_Key_Wrapping_Library_pub.pdf). (Visited on Apr. 13, 2023).
- [27] Marko Milanovic. *Human Rights Treaties and Foreign Surveillance: Privacy in the Digital Age*. Mar. 31, 2014. <https://papers.ssrn.com/abstract=2418485>. (Visited on July 26, 2023). preprint.
- [28] *Crypto++ Library 8.7 — Free C++ Class Library of Cryptographic Schemes*. <https://www.cryptopp.com/>. (Visited on June 5, 2023).
- [29] *C++ Programming Language — Google Cloud*. <https://cloud.google.com/cpp>. (Visited on July 13, 2023).
- [30] Mihir Bellare and Bennet Yee. “Forward-Security in Private-Key Cryptography”. In: *Topics in Cryptology — CT-RSA 2003*. Ed. by Marc Joye. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 1–18. ISBN: 978-3-540-36563-1. DOI: [10.1007/3-540-36563-X\\_1](https://doi.org/10.1007/3-540-36563-X_1).
- [31] *Advanced Cryptographic Ratcheting*. Signal Messenger. <https://signal.org/blog/advanced-ratcheting/>. (Visited on June 5, 2023).
- [32] *GitHub*. <https://github.com/>. (Visited on June 2, 2023).
- [33] Linus Torvalds. *Torvalds/Linux*. <https://github.com/torvalds/linux>. (Visited on July 15, 2023).
- [34] *React*. <https://github.com/facebook/react>. (Visited on June 2, 2023).
- [35] *Google/Guava: Google Core Libraries for Java*. <https://github.com/google/guava>. (Visited on June 2, 2023).
- [36] *freeCodeCamp/freeCodeCamp*. <https://github.com/freeCodeCamp/freeCodeCamp>. (Visited on June 15, 2023).
- [37] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. “On Post-compromise Security”. In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016 IEEE 29th Computer Security Foundations Symposium (CSF). June 2016, pp. 164–178. DOI: [10.1109/CSF.2016.19](https://doi.org/10.1109/CSF.2016.19).