ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

EPFL

Securing Cloud Storage with OpenPGP: An Analysis of Proton Drive

Master Thesis

L. Micheloud

February 2, 2024

Advisors: Prof. Dr. Kenny Paterson (ETH Zürich), Prof. Dr. Serge Vaudenay (EPFL),
Matilda Backendal (ETH Zürich), Daniel Huigens (Proton AG)

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

Abstract

In the last decades, the use of cloud storage has grown significantly. Simultaneously, awareness around privacy issues related to outsourced data increases among uses, causing several cloud storage services offering end-to-end encryption to emerge. As an extension of its existing privacy-oriented product line, Proton AG launched its own end-to-end encrypted (E2EE) cloud storage service, Proton Drive, in 2022. Among the other existing providers such as MEGA and Nextcloud, Proton Drive stands out due to its use of OpenPGP and the structure of its key hierarchy.

We fully document the cryptographic protocol used in Proton Drive, and analyze the security of the file encryption algorithm. We prove that it achieves OUT-IND-CPA security for confidentiality and OUT-WUF-CMA security for authenticity, and show with an attack that stronger notions of OUT-IND-CCA and OUT-SUF-CMA are not possible. In the threat model implied by E2EE, where the adversary controls ciphertexts, weak unforgeability is sufficient for authenticity. For confidentiality however, security under chosen plaintext attack is not sufficient, and it is advisable to aim for security under chosen ciphertext attack.

We put these results in perspective, and discuss other weaknesses in Proton Drive which could lead to attacks. In particular, we stress that the lack of consequences of failed signature checks can lead to powerful attacks, where an adversary who successfully inserts a folder in the file system of a user gains access to everything which the user puts in that folder. We also point out a weak attack allowing to swap some folder and file names, and warn against the addition of compression before file encryption.

Contents

Chapter 1

Introduction

The amount of data stored on computers has increased exponentially over the past few decades. Businesses and individuals alike are moving from physical to digital storage, as it tends to be less cumbersome and more practical, but even though larger and faster hard drives and SSDs have become more widely available and affordable, keeping everything on one's computer can still prove resource intensive. To solve this problem, services have emerged that offer to take the burden of storage away from consumers: cloud storage.

This solution allows the problem of storage capacity to be delegated to third parties. It also places the guarantee of availability on the shoulders of the company providing the service, and offers additional benefits, such as easier backup and file sharing. Due to their convenience, cloud storage services have been widely adopted, with services such as Google Drive surpassing one billion users in 2018 [1].

However, one concession that comes with using cloud storage is the loss of control over who has access to the data one stores. One way of ensuring access control is to let the storage provider encrypt the data before storing it, with the goal of limiting the access to unencrypted data to authorized entities. This strategy is called encryption at rest, and while this does offer protection against adversaries that compromise the database of the service provider, but not the keys, it does not necessarily provide satisfactory guarantees. For example, several of the biggest cloud storage providers, notably DropBox [23], Google Drive [30], and iCloud [7] (for its default setting) offer encryption with keys that are generated and stored on their servers. Keeping the encryption keys on the server of the service provider means that the client does not need to store the keys, making it harder to lose them. However, this also implies that if the cloud storage provider is malicious or if the

[1] https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week/

servers containing the keys are compromised, all security assurances on the data stored by that company are lost.

Because of privacy and security concern raised by cloud storage providers keeping encryption keys, several cloud services have constructed their brand identity around stronger security for the data they store. More specifically, the gold standard for cloud storage security is to provide end-to-end encryption (E2EE), meaning that the data is encrypted and authenticated with keys that only authorized users have access to. Privacy-minded users now consider it normal for their data to be E2EE, especially in messaging apps, where the expectation is that only the sender and recipient of a message can have access to it. In the context of cloud storage, E2EE prevents any other entity, including the cloud storage provider and its servers, from gaining access to the unencrypted files. Note that this is not a formal criterion, and that while E2EE is widely accepted as being a goal to strive for, there is currently no consensus on a security definition for cloud storage.

An increasing number of cloud storage providers are placing a strong emphasis on using E2EE. Examples include MEGA [39], iCloud (in its advanced data protection setting) [7], NextCloud (from version 3.0, with E2EE enabled) [42], Sync [50], pCloud [44] or iDrive [33]. However, recent work on such services [8, 4, 20, 46] has shown that building a robust protocol with the constraints set by the threat model that comes with E2EE is far from trivial.

In this thesis, we examine the solution offered by a relatively new contender, launched in September 2022 by the company Proton AG: Proton Drive. With a line of E2EE products, which started in 2014 with scientists from CERN creating Proton Mail, an email service, Proton AG positions itself as a bastion for privacy among an online services landscape dominated by big tech. Apart from the original email service and Proton Drive, its range of products includes a calendar, a VPN, and a password manager. Due to its history of starting off as an email company, Proton AG makes heavy use of OpenPGP, an open source message format standard which is mostly used for email encryption. In fact, Proton AG is a maintainer for two OpenPGP libraries, OpenPGP.js and GopenPGP, and all of its products use them for all cryptographic operations.

Proton Drive is no exception to that rule, and the entire protocol is designed around OpenPGP. This comes with its own set of constraints and difficulties, because while OpenPGP is also intended for encryption of data at rest [27], at its core, it is designed with email encryption in mind. Proton Drive therefore deviates from the traditional use of OpenPGP. This is due to its design choices in the treatment of files of large size as well as in the structure of the Proton Drive protocol, which closely follows that of a classical file system. We consider these choices, and evaluate whether the downsides that

come with the restrictions imposed by the use of the OpenPGP standard are outweighed by the security provided by the use of an established library.

Overview. The remainder of this chapter is dedicated to related work and contributions. We then start off by giving some notation, definitions, and explanations of the cryptographic primitives used in Chapter 2. Chapter 3 gives a description of the Proton Drive protocol and its threat model. In Chapter 4, we analyze and prove secure (with some restrictions) the file encryption algorithm used in Proton Drive. Finally, we point out parts of the protocol which might be problematic in Chapter 5, and conclude in Chapter 6.

1.1 Related Work

E2EE. E2EE applications have been gaining more and more terrain in recent years. Most notably perhaps, several messaging applications are using E2EE. Examples include the Signal Protocol, which has gotten a lot of attention from academia [6, 19, 13], but also application which have been found to have vulnerabilities, such as Threema [43] and Telegram [5].

While E2EE for messaging and for cloud storage share the same goal of strictly restricting data access to a determined set of users, they present a few differences. Firstly, data in cloud storage is persistent, whereas messages are ephemeral. This means that contrary to cloud storage keys, messaging keys can be discarded once they have been delivered and used, making it easier to achieve strong security guarantees. Secondly, messaging is centered around exchange between users, whereas this notion only appears when sharing files in cloud storage, with most uses corresponding to a user retrieving the data it previously stored. Finally, messaging needs to preserve the chronological order of messages, whereas chronology only comes in play in cloud storage if it implements some form of versioning of files. In these aspects, E2EE for cloud storage is closer to E2EE for password managers such as 1Password [2], Bitwarden [14] or Dashlane [21], where the server acts as a vault more than as an intermediary between a sender and a receiver.

Note that E2EE does not aim to provide metadata protection. New file sharing applications, such as Metal [18] and Titanium [17] aim to also hide metadata, such as user identities and access rights, from the server. Such systems are called metadata-hiding file-sharing systems.

Cloud Storage Security. Several cloud storage services centered around security have been developed in recent years. Among these, MEGA [39], Nextcloud [1], PreVeil [45], and Tresorit [53] provide a white paper of the design of their protocol. Surveys on the security of such service have also

been conducted by Virvilis et al. [54], and more recently by Haller [31]. The latter revealed major vulnerabilities in the design of MEGA, which were further explored by Backendal et al. [8] and Albrecht et al. [4]. Analyses Nextcloud also pointed out some flaws in their protocol [20].

OpenPGP. OpenPGP is mostly used in email or email-related services, with integration in Mailvelope, Mailfence, Proton Mail (which has been audited several times [35, 48, 47]), and Delta Chat, for example. Other applications are rare, but examples include the password manager Passbolt [32], the entire Proton suite (which includes a calendar, a VPN and a password manager), and, more generally, using OpenPGP to encrypt local files.

We draw attention to some attacks that have been found on OpenPGP. The first one is a key overwriting attack related to the way asymmetric keys are encrypted, and is presented in a paper by Bruseghini et al. [16]. Some attacks using format oracles have also been pointed out by Mister et al. [40] and by Maury et al. [37].

Provable Security. To evaluate the security provided by the Proton Drive protocol, we build on common security notions for different types of cryptographic primitives. We use some notions that have long been well-established in the field [10], such as indistinguishability or unforgeability, and more recent notions such as key robustness [3, 25, 28] or security for signcryption schemes [36]. We use standard game-hopping proof techniques that have been described several times, by Bellare and Rogaway [11, 12], Shoup [49] and Dent [22] among others.

1.2 Contributions

This thesis makes the following contributions:

1. We provide a full description of the Proton Drive protocol from a cryptographic point of view. This description is completed with explanations of the relevant parts of the underlying OpenPGP standard as well as of some design choices in the protocol. We produce a white paper of Proton Drive, which we use for our subsequent analysis.
2. We translate the claims made by Proton AG regarding the security of Proton Drive into a threat model, and use this base to define concrete goals in terms of security properties.
3. We evaluate whether the file encryption algorithm of Proton Drive meets these goals. For confidentiality, we prove that it fails to provide the security goal established by our threat model, and show a weaker property instead. For authenticity, we prove that it meets the security goal from the threat model.

Chapter 2

Background

As we will be discussing the security of a cryptographic protocol, we first need to introduce some notation as well as the cryptographic primitives that Proton Drive uses. We then present OpenPGP, the data formatting standard which underlies operations in the protocol, and present the main building blocks that are used in Proton Drive. This enables us to explain some of the design choices of the protocol. For example, Proton Drives chose to use a key hierarchy which is follows the same tree-like structure as unix file systems, with nodes of the tree being linked to each other by the classical PGP encryption method.

2.1 Notation

The following notation conventions are used in this thesis:

2.1.1 Operations on Strings

We consider characters in strings or bits in bitstrings to be numbered from left to right starting from 0.

a^x The string composed of the concatenation of the character a x times.

$\mathcal{S}^n$ With $\mathcal{S}$ a set of characters, the set of all strings of length n composed of characters in the set $\mathcal{S}$.

$\mathcal{S}^*$ With $\mathcal{S}$ a set of characters, the set of all strings of non-negative integer length composed of characters in the set $\mathcal{S}$.

$x \parallel y$ The concatenation of the strings x and y.

$|x|$ The length of the string x. If $|x| < |y|$, $x \oplus y$ is syntactic sugar for $(x \parallel 0^{|y|-|x|}) \oplus y$, and vice versa.

$s[n:m]$ The substring of s from the n-th character (included) to the m-th character (excluded).

`Bytes` The set of all bytes, i.e $\{$`0x00`, ..., `0xff`$\}$.

`Chars` The set of all Unicode characters.

2.1.2 Pseudo-code

$y \leftarrow x$ The value of x is assigned to the variable y.

$x \xleftarrow{\$} X$ The value x is picked uniformly at random from the set X.

$x = y$ The boolean expression that compares the values of x and y. If they are equal, return true, else return false.

$y \leftarrow f(x)$ The output of the function f on input x is assigned to the variable y.

$y \xleftarrow{\$} f(x)$ The output of the non-deterministic algorithm f on input x is assigned to the variable y.

`Obj`.*attr* The attribute *attr* which belongs to object `Obj`.

`Obj`.`fun` The function `fun` which belongs to object `Obj`.

$f(x_1, ..., x_{ar(f)})$ The function f called on its arguments $x_1, ..., x_{ar(f)}$, where $ar(f)$ denotes the arity of f.

$\mathcal{S} \xleftarrow{\cup} \mathcal{S}'$ We use this as a shorthand notation for $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}'$.

In code, we consider that $\top$ has the boolean value *true* and that $\bot$ has the boolean value *false*.

When writing for loops of the form **For** i **from** 0 **to** n, we consider i to iterate on all values from 0 to n, including n.

2.1.3 Cryptography

$\mathcal{M}$ The message space of a cryptographic primitive.

$\mathcal{C}$ The ciphertext space of a cryptographic primitive.

$\mathcal{K}$ The key space of a cryptographic primitive. For asymmetric primitives, the space of key pairs.

$\mathcal{PuK}$ For asymmetric cryptographic primitives, the space of public keys.[1]

[1] We use *puk* for public keys and *prk* for private keys instead of the conventional *pk* and *sk*. This notation choice allows use to distinguish private keys from session keys, which we write *sk*, and signature keys, which we write *sik*.

$\mathcal{PrK}$ For asymmetric cryptographic primitives, the space of private keys.[2]

$\mathcal{H}$ The output space of a hash function or the Hash-based Message Authentication Code (HMAC) function.

IV An initialization vector. This is a value which is passed to an encryption algorithm in order to give it a starting state. Typically, it is used to allow reusing a key without risking to leak information on the plaintext.

Sign The signature generation function of a signature scheme.

Vfy The verification function of a signature scheme.

When referring to an encryption or decryption algorithm that uses a given key k, we write the key as a subscript, e.g. AES-256.Enc_k.

2.1.4 Game-Based Security

$\mathcal{A}^{\mathcal{O}}$ The adversary $\mathcal{A}$ is given access to an oracle $\mathcal{O}$.

$Game^{\mathcal{A}}_{Scheme}(args)$ The return value of game *Game* defined for the cryptographic scheme *Scheme* with arguments *args* when played by an adversary $\mathcal{A}$.

$\mathcal{A}dv^{Game}_{Scheme}(\mathcal{A})$ The advantage of adversary playing the game $Game_{Scheme}$. For games where the adversary produces a guess b' for a value $b \in \{0,1\}$ chosen by the challenger, we define it as

$$\mathcal{A}dv^{Game}_{Scheme}(\mathcal{A}) = 2\left(Pr[Game^{\mathcal{A}}_{Scheme}()] - 1/2\right),$$

which is equivalent to

$$\mathcal{A}dv^{Game}_{Scheme}(\mathcal{A}) = Pr[b' = 1|b = 1] - Pr[b' = 1|b = 0]$$

by the advantage rewriting lemma.

2.2 Cryptographic Primitives

We start by giving some basic cryptographic building blocks that we use. When relevant, we also give game-based security definitions that can be applied to those primitives. Such games allow to model the capacities of an attacker and to put a bound on its winning probability.

[2] See Footnote 1.

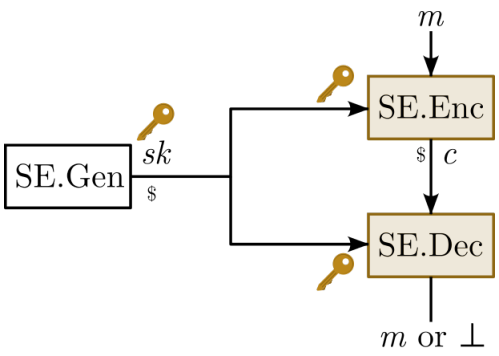

Figure 2.1: The three algorithms composing an SE scheme.

2.2.1 Symmetric Encryption

A symmetric encryption (SE) scheme with key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is a tuple (Gen, Enc, Dec) of functions such that:

- $\mathsf{Gen} : \emptyset \to \mathcal{K}$ generates a key uniformly at random from the keyspace $\mathcal{K}$.
- $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ encrypts a message in the message space $\mathcal{M}$ using a key in the keyspace $\mathcal{K}$.
- $\mathsf{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M} \cup \{\perp\}$ decrypts a message in the ciphertext space $\mathcal{C}$ using a key in the keyspace $\mathcal{K}$. In case of failure (typically due to an invalid ciphertext) Dec returns $\perp$.

An SE scheme is correct if we have

$$Pr[\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m] = 1 \qquad \forall m \in \mathcal{M}$$

with $k \xleftarrow{\$} \mathsf{Gen}()$.

IND-CPA

Indistinguishability under chosen plaintext attack (IND-CPA) is a security notion for encryption schemes that indicates that an adversary which chooses two plaintext and gets back the encryption of one of the two cannot distinguish which plaintext it was derived from. For a given symmetric encryption scheme $SE = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, we define it in Fig. 2.2.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv_{SE}^{\text{IND-CPA}}(\mathcal{A}) = 2\left(Pr[\text{IND-CPA}_{SE}^{\mathcal{A}}()] - 1/2\right).$$

Game $\text{IND-CPA}^{\mathcal{A}}_{SE}()$	Oracle $LoR(m_0, m_1)$
1 $b \xleftarrow{\$} \{0,1\}$	1 **if** $\lvert m_0\rvert \neq \lvert m_1\rvert$
2 $k \xleftarrow{\$} \mathsf{Gen}()$	2 **return** $\perp$
3 $b' \xleftarrow{\$} \mathcal{A}^{LoR}()$	3 **return** $\mathsf{Enc}_k(m_b)$
4 **return** $b = b'$	

Figure 2.2: IND-CPA security game for SE schemes.

IND-CCA

Indistinguishability under chosen ciphertext attack (IND-CCA) follows the same idea as IND-CPA, but the adversary has access to a decryption oracle as well in addition to the encryption oracle. For a given symmetric encryption scheme $SE = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, we define it in Fig. 2.3.

Game $\text{IND-CCA}^{\mathcal{A}}_{SE}()$	Oracle $LoR(m_0, m_1)$
1 $b \xleftarrow{\$} \{0,1\}$	1 **if** $\lvert m_0\rvert \neq \lvert m_1\rvert$
2 $\mathcal{S} \leftarrow \emptyset$	2 **return** $\perp$
3 $k \xleftarrow{\$} \mathsf{Gen}()$	3 $c \leftarrow \mathsf{Enc}_k(m_b)$
4 $b' \xleftarrow{\$} \mathcal{A}^{LoR,dec}()$	4 $\mathcal{S} \xleftarrow{\cup} \{c\}$
5 **return** $b = b'$	5 **return** c
	Oracle $dec(c)$
	1 **if** $c \in \mathcal{S}$
	2 **return** $\perp$
	3 **return** $\mathsf{Dec}_k(c)$

Figure 2.3: IND-CCA security game for SE schemes.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{IND-CCA}}_{SE}(\mathcal{A}) = 2\left(Pr[\text{IND-CCA}^{\mathcal{A}}_{SE}()] - 1/2\right).$$

INT-CTXT

Integrity of ciphertexts (INT-CTXT) is a security notion for SE schemes that indicates that an adversary which has access to an encryption oracle cannot forge a new ciphertext. For a given symmetric encryption scheme $SE = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, we define it in Fig. 2.4.

Game $\text{INT-CTXT}^{\mathcal{A}}_{SE}()$	Oracle $enc(m)$
1 $b \xleftarrow{\$} \{0,1\}$	1 $c \leftarrow \mathsf{Enc}_k(m_b)$
2 $k \xleftarrow{\$} \mathsf{Gen}()$	2 $\mathcal{S} \xleftarrow{\cup} \{c\}$
3 $\mathcal{S} \leftarrow \emptyset$	3 **return** c
4 $c' \xleftarrow{\$} \mathcal{A}^{enc}()$	
5 $m' \leftarrow \mathsf{Dec}_k(c')$	
6 **return** $m' \neq \perp$ and $c' \notin \mathcal{S}$	

Figure 2.4: INT-CTXT security game for SE schemes.

The advantage of an adversary $\mathcal{A}$ playing this game is

$$\mathcal{A}dv^{\text{INT-CTXT}}_{SE}(\mathcal{A}) = Pr[\text{INT-CTXT}^{\mathcal{A}}_{SE}()].$$

AE

Authenticated encryption (AE) is the combination of IND-CPA and INT-CTXT security.

KROB

Key robustness (KROB) is a security notion indicating that for a given ciphertext c encrypted under a secret key k, it is not possible to find another secret key k' under which c decrypts correctly. This notion was first introduced for public-key cryptography by Abdalla et al. [3] and adapted for symmetric encryption by Farshim et al. [26]. We define the game KROB for a given symmetric encryption scheme $SE = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ in Fig. 2.5.

Game $\text{KROB}^{\mathcal{A}}_{SE}()$
1 $(c, sk_0, sk_1) \xleftarrow{\$} \mathcal{A}()$
2 **return** $sk_0 \neq sk_1$ and $\mathsf{Dec}_{sk_0}(c) \neq \perp$ and $\mathsf{Dec}_{sk_1}(c) \neq \perp$

Figure 2.5: KROB security game for SE schemes.

The advantage of an adversary $\mathcal{A}$ playing this game is

$$\mathcal{A}dv^{\text{INT-CTXT}}_{SE}(\mathcal{A}) = Pr[\text{KROB}^{\mathcal{A}}_{SE}()].$$

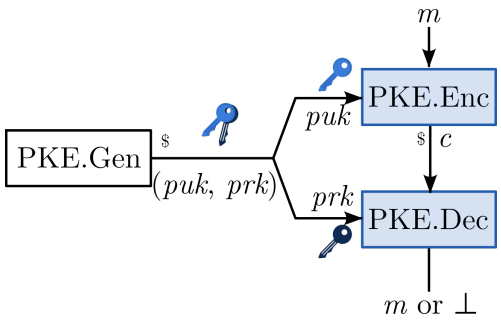

Figure 2.6: The three algorithms composing a PKE scheme.

2.2.2 Public-Key Encryption

A public-key encryption (PKE) scheme with key space $\mathcal{K} = \mathcal{PuK} \times \mathcal{PrK}$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is a tuple (Gen, Enc, Dec) of functions such that:

- Gen : $\emptyset \rightarrow \mathcal{PuK} \times \mathcal{PrK}$ generates a key pair uniformly at random from the keyspace $\mathcal{K}$.
- Enc : $\mathcal{PuK} \times \mathcal{M} \rightarrow \mathcal{C}$ encrypts a message in the message space $\mathcal{M}$ using a key in the public key space $\mathcal{PuK}$.
- Dec : $\mathcal{PrK} \times \mathcal{C} \rightarrow \mathcal{M} \cup \{\perp\}$ decrypts a message in the ciphertext space $\mathcal{C}$ using a key in the private key space $\mathcal{PrK}$. In case of failure (typically due to an invalid ciphertext) Dec returns $\perp$.

A PKE scheme is correct if we have

$$Pr[\mathsf{Dec}_{prk}(\mathsf{Enc}_{puk}(m)) = m] = 1 \qquad \forall m \in \mathcal{M}$$

with $(puk, prk) \xleftarrow{\$} \mathsf{Gen}()$.

IND-CPA

Similarly to the IND-CPA notion for SE schemes, we define the game $\text{IND-CPA}^{\mathcal{A}}_{PKE}$ in Fig. 2.7.

The advantage of an adversary $\mathcal{A}$ playing this game is

$$\mathcal{A}dv^{\text{IND-CPA}}_{PKE}(\mathcal{A}) = 2\left(Pr[\text{IND-CPA}^{\mathcal{A}}_{PKE}()] - 1/2\right).$$

IND-CCA

We define IND-CCA for PKE schemes in Fig. 2.8.

The advantage of an adversary $\mathcal{A}$ playing this game is

$$\mathcal{A}dv^{\text{IND-CCA}}_{PKE}(\mathcal{A}) = 2\left(Pr[\text{IND-CCA}^{\mathcal{A}}_{PKE}()] - 1/2\right).$$

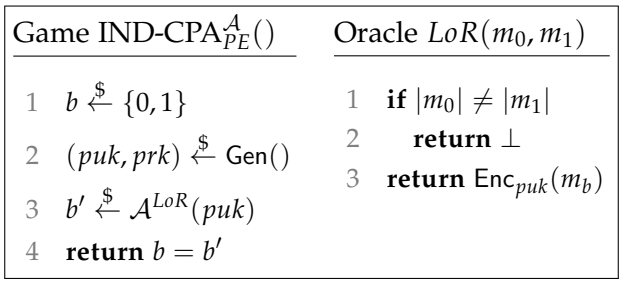

Figure 2.7: IND-CPA security game for PKE schemes.

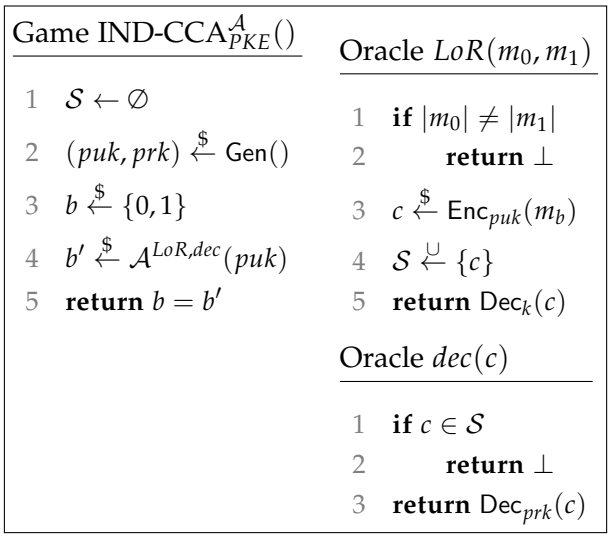

Figure 2.8: IND-CCA security game for PKE schemes.

2.2.3 Message Authentication Code

A message authentication code (MAC) scheme with key space $\mathcal{K}$, message space $\mathcal{M}$ and tag space $\mathcal{T}$ is a tuple (Gen, Tag, Vfy) of functions such that:

- $\mathsf{Gen} : \emptyset \to \mathcal{K}$ generates a key uniformly at random from the keyspace $\mathcal{K}$.
- $\mathsf{Tag} : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ returns a tag on a message in the message space $\mathcal{M}$ using a key in the key space $\mathcal{K}$.
- $\mathsf{Vfy} : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \{\top, \perp\}$ verifies that a tag corresponds to a given message. If this is the case, Vfy returns $\top$, otherwise it returns $\perp$.

A MAC scheme is correct if we have

$$Pr[\mathsf{Vfy}_{tk}(m, \mathsf{Tag}_{tk}(m)) = \top] = 1 \qquad \forall m \in \mathcal{M}$$

with $tk \xleftarrow{\$} \mathsf{Gen}()$.

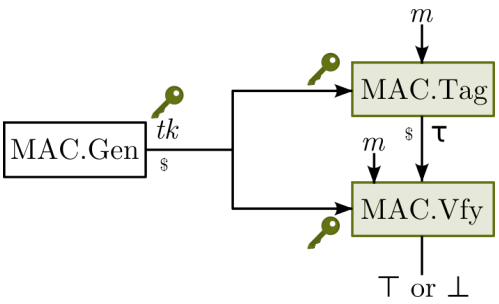

Figure 2.9: The three algorithms composing a MAC scheme.

WUF-CMA

Weak unforgeability under chosen message attack (WUF-CMA) is a security notion for MAC schemes that indicates that an adversary which has access to a tag oracle cannot create a valid tag for a message that it has never queried before. We define the game $\text{WUF-CMA}^{\mathcal{A}}_{MAC}$ as in Fig. 2.10.

Game $\text{WUF-CMA}^{\mathcal{A}}_{MAC}()$	Oracle $tag(m)$
1 $\mathcal{S} \leftarrow \emptyset$	1 $\tau \xleftarrow{\$} \mathsf{Tag}_k(m)$
2 $k \xleftarrow{\$} \mathsf{Gen}()$	2 $\mathcal{S} \xleftarrow{\cup} \{m\}$
3 $(m, \tau) \xleftarrow{\$} \mathcal{A}^{tag}()$	3 **return** τ
4 **return** $m \notin \mathcal{S}$ and	
5 $\mathsf{Vfy}_k(m, \tau) = \top$	

Figure 2.10: WUF-CMA security game for MAC schemes.

The advantage of an adversary $\mathcal{A}$ playing this game is

$$\mathcal{A}dv^{\text{WUF-CMA}}_{MAC}(\mathcal{A}) = Pr[\text{SUF-CMA}^{\mathcal{A}}_{MAC}()].$$

2.2.4 Digital Signature

A digital signature (DS) scheme with key space $\mathcal{K} = \mathcal{P}u\mathcal{K} \times \mathcal{P}r\mathcal{K}$, message space $\mathcal{M}$ and signature space $\mathcal{S}$ is a tuple (Gen, Sign, Vfy) of functions such that:

- $\mathsf{Gen} : \emptyset \rightarrow \mathcal{P}u\mathcal{K} \times \mathcal{P}r\mathcal{K}$ generates a key uniformly at random from the keyspace $\mathcal{K}$.
- $\mathsf{Sign} : \mathcal{P}r\mathcal{K} \times \mathcal{M} \rightarrow \mathcal{S}$ signs a message in the message space $\mathcal{M}$ using a key in the private key space $\mathcal{P}r\mathcal{K}$.

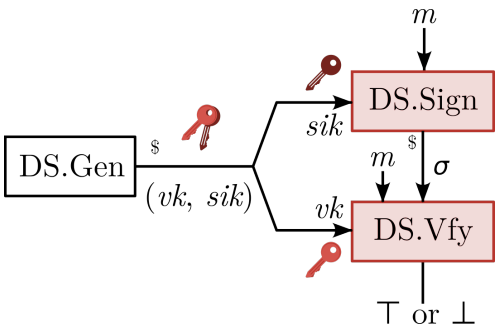

Figure 2.11: The three algorithms composing a DS scheme.

- Vfy : $\mathcal{PuK} \times \mathcal{M} \times \mathcal{S} \rightarrow \{\top, \bot\}$ verifies that a signature corresponds to a given message. If this is the case, Vfy returns $\top$, otherwise it returns $\bot$.

A signature scheme is correct if we have

$$Pr[\mathsf{Vfy}_{vk}(m, \mathsf{Sign}_{sik}(m)) = \top] = 1 \qquad \forall m \in \mathcal{M}$$

with $(vk, sik) \xleftarrow{\$} \mathsf{Gen}()$.

WUF-CMA

The notion of WUF-CMA for DS schemes is similar to WUF-CMA for MAC schemes, with the difference that DS use a key pair, where the public part is used for verification and the private part for signing. The adversary is given the knowledge of the public part of the key.

SUF-CMA

Strong unforgeability under chosen message attack (SUF-CMA) for DS schemes is a security notion indicating that an adversary which has access to a signature oracle cannot create a new valid message-signature pair. We define the game $\text{SUF-CMA}^{\mathcal{A}}_{DS}$ in Fig. 2.12.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{SUF-CMA}}_{DS}(\mathcal{A}) = Pr[\text{SUF-CMA}^{\mathcal{A}}_{DS}()].$$

2.2.5 Signcryption

A signcryption (SC) scheme is a tuple (SGen, RGen, SC, USC) which performs a combination of signature and encryption. Signcryption schemes are designed to provide confidentiality and authentication on a message transmitted from a sender to a receiver, with asymmetric keys rather than

Game $\text{SUF-CMA}^{\mathcal{A}}_{DS}(vk)$	Oracle $sign(m)$
1 $\mathcal{S} \leftarrow \emptyset$	1 $\sigma \xleftarrow{\$} \mathsf{Sign}_{sik}(m)$
2 $(vk, sik) \xleftarrow{\$} \mathsf{Gen}()$	2 $\mathcal{S} \xleftarrow{\cup} \{(m, \sigma)\}$
3 $(m, \sigma) \xleftarrow{\$} \mathcal{A}^{sign}(vk)$	3 **return** σ
4 **return** $(m, \sigma) \notin \mathcal{S}$ and	
5 $\mathsf{Vfy}_{vk}(m, \sigma) = \top$	

Figure 2.12: SUF-CMA security game for DS schemes.

symmetric ones. Note that the sender and receiver can be the same entity, as will typically be the case in our applications.

Signcryption schemes are often built from other cryptographic primitives, for instance a PKE and a DS scheme, in which case its sender key space would be $\mathcal{SK} = DS.\mathcal{K}$ and its receiver key space $\mathcal{RK} = PKE.\mathcal{K}$. Because of this, we borrow the notations for keys from DS and PKE schemes: we refer to the sender key pair as (vk, sik) and to the receiver key pair as (puk, prk).

- $\mathsf{SGen} : \emptyset \to \mathcal{SK}$ generates a sender key pair uniformly at random from the keyspace $\mathcal{SK}$. The sender key is attached to the identity of a given sender, and it is assumed that only that sender knows the private sender key. This allows to authenticate the entity which has performed a given signcryption.
- $\mathsf{RGen} : \emptyset \to \mathcal{RK}$ generates a receiver key pair uniformly at random from the keyspace $\mathcal{RK}$. The receiver key is attached to the identity of a given receiver, and it is assumed that only that receiver knows the private receiver key. This allows to restrict the ability to unsigncrypt a ciphertext to a given receiver.
- $\mathsf{SC} : PKE.\mathcal{PuK} \times DS.\mathcal{PrK} \times \mathcal{M} \to \mathcal{C}$, the signcryption algorithm, signcrypts a message in the message space $\mathcal{M}$ using an encryption key in the keyspace $\mathcal{RK}.\mathcal{PuK}$ and a signature key in the keyspace $\mathcal{SK}.\mathcal{PrK}$. The result can either be a signature-ciphertext pair, in which case we say that the signature is detached, or the signature can be encrypted together with the plaintext, in which case it is attached.
- $\mathsf{USC} : PKE.\mathcal{PrK} \times DS.\mathcal{PuK} \times \mathcal{C} \to \mathcal{M} \cup \{\perp\}$ unsigncrypts a signcryption using a decryption key in the keyspace PKE.$\mathcal{PrK}$ and a verification key in the keyspace DS.$\mathcal{PuK}$. In case of failure (for example if the signature does not correspond to the message) Dec returns $\perp$.

A signcryption scheme is correct if we have

$$Pr[\mathsf{USC}_{vk,prk}(\mathsf{SC}_{sik,puk}(m)) = m] = 1 \qquad \forall m \in \mathcal{M}$$

with $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$ and $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$.

Security notions for signcryption schemes are classified into categories following two axes. The first one is how many users are involved in the signcryption scheme. If there is only one sender and one receiver, we speak of a two-user setting, otherwise we call it a multi-user setting. The second axis depends on the knowledge of the adversary. In a setting where the adversary is internal, we consider that it can have access to some of the private keys, either because the adversary belongs to some entity involved in the signcryption or because some key has been compromised. On the other hand, outsider security implies that the adversary does not have any knowledge of private keys. We give three security notions for outsider security which are adapted from those defined in [36].

OUT-IND-CPA

The OUT-IND-CPA notion for signcryption is similar to that of PKE schemes, with the addition of the sender keys. We define it in Fig. 2.13.

Game $\text{OUT-IND-CPA}^{\mathcal{A}}_{SC}()$	Oracle $LoR(m_0, m_1)$
1 $b \xleftarrow{\$} \{0,1\}$	1 **if** $\lvert m_0\rvert \neq \lvert m_1\rvert$
2 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$	2 **return** $\perp$
3 $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$	3 **return** $\mathsf{SC}_{sik,puk}(m_b)$
4 $b' \xleftarrow{\$} \mathcal{A}^{LoR}(vk, puk)$	
5 **return** $b = b'$	

Figure 2.13: OUT-IND-CPA security game for SC schemes.

The advantage of the adversary $\mathcal{A}$ against this game is

$$\mathcal{A}dv^{\text{OUT-IND-CPA}}_{SC}(\mathcal{A}) = 2\left(Pr[\text{OUT-IND-CPA}^{\mathcal{A}}_{SC}()] - 1/2\right).$$

OUT-IND-CCA

The IND-CCA notion for signcryption schemes is similar to that of PKE schemes. The most notable difference lies in the addition of the sender key, of which the public part is given to the adversary. A signcryption scheme satisfying IND-CCA security notion makes recognizing which of two chosen plaintexts was used to produce the retrieved signcryption. We define the game OUT-IND-CCA in Fig. 2.14.

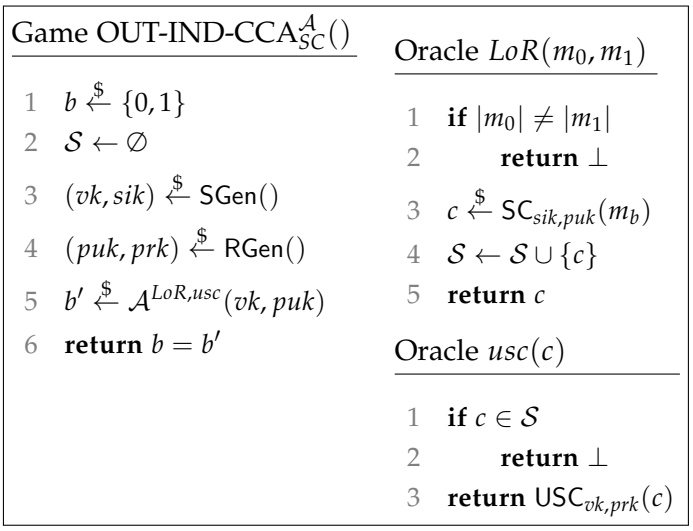

Figure 2.14: OUT-IND-CCA security game for SC schemes.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{OUT-IND-CCA}}_{SC}(\mathcal{A}) = 2\left(Pr[\text{OUT-IND-CCA}^{\mathcal{A}}_{SC}()] - 1/2\right).$$

OUT-WUF-CMA

The OUT-WUF-CMA notion for signcryption indicates that an adversary with access to a signcryption oracle cannot find a valid signcryption for a message that it has never queried before. We define the game OUT-WUF-CMA$^{\mathcal{A}}_{SC}$ in Fig. 2.15.

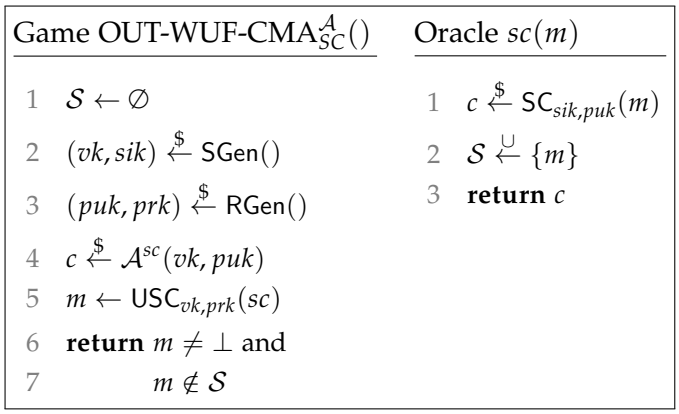

Figure 2.15: OUT-WUF-CMA security game for SC schemes.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{OUT-WUF-CMA}}_{SC}(\mathcal{A}) = Pr[\text{OUT-WUF-CMA}^{\mathcal{A}}_{SC}()].$$

OUT-SUF-CMA

The OUT-SUF-CMA notion for signcryption indicates that an adversary with access to a signcryption oracle cannot find a new message-signcryption pair. We define the game $\text{OUT-SUF-CMA}^{\mathcal{A}}_{SC}$ in Fig. 2.16.

Game $\text{OUT-SUF-CMA}^{\mathcal{A}}_{SC}()$	Oracle $sc(m)$
1 $\mathcal{S} \leftarrow \emptyset$	1 $c \xleftarrow{\$} \mathsf{SC}_{sik,puk}(m)$
2 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$	2 $\mathcal{S} \xleftarrow{\cup} \{(m, c)\}$
3 $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$	3 **return** c
4 $c \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$	
5 $m' \leftarrow \mathsf{USC}_{vk,prk}(sc)$	
6 **return** $(m, c) \notin \mathcal{S}$ and	
7 $m \neq \perp$	

Figure 2.16: OUT-SUF-CMA security game for SC schemes.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{OUT-SUF-CMA}}_{SC}(\mathcal{A}) = Pr[\text{OUT-SUF-CMA}^{\mathcal{A}}_{SC}()].$$

2.2.6 Hash

A cryptographic hash function $\texttt{Hash} : \{0,1\}^* \rightarrow \{0,1\}^\ell$ is a cryptographic primitive that maps any binary string to a binary string of length ℓ.

Collision Resistance

Collision resistance (CR) is a security notion for hash functions stating that it is hard to find two values that produce the same hash. We give its definition in Fig. 2.17.

Game $\text{CR}^{\mathcal{A}}_{\texttt{Hash}}()$
1 $m_0, m_1 \xleftarrow{\$} \mathcal{A}()$
2 **return** $\texttt{Hash}(m_0) = \texttt{Hash}(m_1)$

Figure 2.17: Collision resistance security game for hash functions.

The advantage of an adversary $\mathcal{A}$ playing that game is

$$\mathcal{A}dv^{\text{CR}}_{\texttt{Hash}}(\mathcal{A}) = Pr[\text{CR}^{\mathcal{A}}_{\texttt{Hash}}()].$$

2.2.7 PAKE

A Password Authenticated Key Exchange (PAKE) is a way to derive a key between two parties, typically a client and a server, based on at least one party knowing a password. PAKEs can also be used as a way for a client to prove to the server that it knows a secret without revealing its value. In these cases, since the server itself does not know the password, the protocol is actually called an augmented (or asymmetric) PAKE (aPAKE). Currently, the aPAKE which Proton uses is Secure Remote Password (SRP), but there are discussions about replacing it by OPAQUE.

2.3 Cryptographic Algorithms

2.3.1 AES

OpenPGP uses the Advanced Encryption Standard (`AES`) for all symmetric encryption. `AES` is a block cipher, i.e. a symmetric encryption algorithm that encrypts blocks of 128 bits of data at a time. It exists in three version, namely `AES-128`, `AES-192`, and `AES-256`, with key lengths of 128, 192, and 256 bits respectively. The version used in Proton drive is `AES-256`.

Data of length greater than 128 bits needs to be split into blocks of compliant size. The way a symmetric encryption scheme is built from AES is called the mode of operation. We present two modes of operation, Cipher Feedback (CFB) and Galois/Counter Mode (GCM), as the former one is currently used for symmetric encryption in OpenPGP and the latter is being introduced as part of the revision of the message format, crypto refresh [56].

CFB

AES-CFB is the SE scheme that is currently used by OpenPGP. This mode of operation presents the advantage of not requiring an implementation of AES-256 decryption. We give a graphical representation of the CFB mode encryption in Fig. 2.18, but no complete description as we do not need it in this thesis.

Assuming that AES-256 is a pseudo-random permutation (PRP) and as long as the IV is not predictable, AES-CFB is IND-CPA. A proof of this can be found in [55].

GCM

GCM is an AEAD scheme which is not yet used in OpenPGP, but will be introduced as part of the crypto refresh [56] to replace the current use of AES-CFB for encryption. We do not give a description of the GCM mode of operation as we do not need it in this thesis.

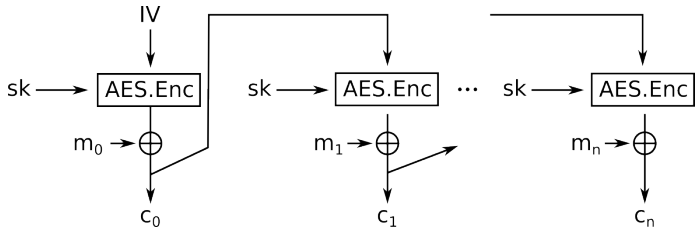

Figure 2.18: Graphical representation of the encryption for the CFB mode of operation for block ciphers.

Assuming that AES-256 is a PRP and that the IV is not repeated, AES-GCM is proven to be AE secure in [38].

2.3.2 PKE on Curve25519

By default, Proton Drive uses a PKE based on Elliptic Curve Diffie Hellman (ECDH) on Curve25519 for public-key encryption, as specified in the crypto refresh [56], because there are no standardized PKEs using solely elliptic curve cryptography (ECC). The algorithm that is used is a hybrid; it utilizes a key agreement algorithm to generate a symmetric key, which serves to initialize a chain of encryption of symmetric keys, thus internally using a SE scheme for the effective encryption of data. The high-level algorithm, which involves elliptic curve Diffie-Hellman (ECDH), a key derivation function (KDF), a key wrapping algorithm, and OpenPGP.SE, is the following.

A key is derived using ECDH, then passed through a KDF to transform it into a key encryption key (KEK). Then a session key is generated, which encrypts the data, and is stored encrypted by the KEK. It is possible for a user to manually add a key for a different algorithm, namely RSA on the condition that the key be at least 2048 bits long, or one of the NIST elliptic curves. In the following, we only consider the use of ECC with Curve25519, and refer to that PKE as OpenPGP.PKE.

2.3.3 HMAC-SHA256

As Message Authentication Code (MAC), HMAC-SHA256 is used, and we refer to it as OpenPGP.HMAC. HMAC-SHA256 has been shown to be a PRF by Bellare [9], and as a MAC built from a pseudo-random-function (PRF) is WUF-CMA secure [10, Proposition 7.3], we have that HMAC-SHA256 is WUF-CMA.

2.3.4 EdDSA

As for PKE, the default digital signature algorithm in Proton Drive, Edwards-Curve Digital Signature Algorithm (EdDSA), uses Curve25519, but it is also

possible to import keys for RSA signature or other curves. Again, we denote the use of Ed25519 (EdDSA on Curve25519) as a DS as OpenPGP.DS and disregard the other options.

Ed25519 has been shown to be SUF-CMA by Brendel et al. [15].

2.3.5 SHA256

As a generic hash function, Proton Drive uses Secure Hash Algorithm (SHA)256, which we denote as OpenPGP.Hash.

2.3.6 bcrypt

bcrypt is a key derivation function (KDF) which is used to derive a passphrase (the name given to a key used for "locking" —that is, encrypting— another key), as well as as part of the user authentication.

2.3.7 SRP

SRP is an interactive protocol between a client and a server and belongs to the family of aPAKEs. The server holds a password-derived value which is used to verify that the client knows the password. This value is created by the client during the password registration as follows: the client picks a salt s, computes $v = g^{H(s,p)} \mod N$ (where p is the password, H is a hash function, N is a Sophie Germain prime, and g is a generator of $\mathbb{Z}_N^*$ are SRP parameters), and sends s and v to the server, which links them to the identity of the client. To verify that the client has knowledge of the password, they both create a value derived from ephemeral asymmetric keys, and the server deduces that the client knows the password if both values are equal.

2.4 OpenPGP

OpenPGP is a standard defining formats for secure exchange and storage of data, and is based on PGP (Pretty Good Privacy), an encryption program released in 1991 by Phil Zimmermann. Its current official version, and that which is used for all Proton applications, is RFC4880 [27], but a revision of the document, crypto refresh [56], is submitted for validation. The use of OpenPGP was motivated by the fact that the company already had an OpenPGP implementation, due to its first product being a mailing service.

2.4.1 Messages and Packets

To achieve confidentiality and integrity for the data it handles, OpenPGP uses symmetric and asymmetric encryption as well as signature algorithms. It therefore needs to manipulate keys, signatures and encrypted data. All of

these are formatted in blocks called messages, which themselves are formed from one or more data units named packets, where each packet has a given type indicating what it contains. The type of a packet is indicated in the first byte of its header.

The rest of the header is used to indicate the length of the packet. This value can either be entirely determined in the header, in which case its maximal possible size is around 4MiB, or, for a data packet whose final length is not known at first, it can give a partial body length, meaning that the packet is actually composed of multiple chunks, with each chunk starting with an indication of its length. Only the former is used in Proton Drive.

In the next subsections, we give an overview of the various parts of OpenPGP that Proton Drive utilizes.

2.4.2 Keys

Asymmetric keys

In OpenPGP, asymmetric keys are stored in key material packets (KM packet), which encapsulate either a public key, that contains only the public key material, or a private key, in which case it comprises both the public and secret key material. An interesting thing to note is that in RFC4880 [27], when encrypting a private key for storage, only the secret key material is encrypted, meaning that there is no integrity protection on the public key material and a verification step should be added after decryption to check that both parts of the key match.

Moreover, a key can be a subkey, meaning it is associated to a primary key, which allows to constitute some form of keyring. A subkey only differs from a normal key by its tag; its bound to a given primary key is indicated by a (separate) subkey binding signature packet. Since the primary key is required to sign such a packet, it is necessary for it to be a signing key. Typically, applications use a primary signing key to identify the user, and associate a subkey to it for encryption.

Every key is identified either by its fingerprint, which consists of a hash digest of the key, or by its key ID, which corresponds to the first sixty-four bits of the fingerprint. This allows to identify which key was used to sign or encrypt a given packet.

Symmetric keys

Symmetric keys are called session keys and are used to encrypt messages. They are randomly generated and have the format of the key for the cipher that is being used, e.g. if the encryption cipher is AES-256, the session key is 32 bytes long.

Symmetric keys need to be encrypted for storage. There are two types of packets for that, namely public-key encrypted session key (PKESK) packets and symmetric-key encrypted session key (SKESK) packets, for asymmetrically and symmetrically encrypted session keys respectively. These can be prepended to a packet encrypted by the session key they contain. It is to be noted that SKESK packets are not encrypted directly by another session key, but rather use passphrases, which we describe next.

Passphrases

Passphrases are used to encrypt keys symmetrically. In contrast to symmetric keys, they do not need to respect the key format of the block cipher, because they are passed through the KDF of OpenPGP, string-to-key (S2K). The way S2K works is by passing the provided passphrase and (in the case of a salted S2K) a salt one or more times through a hash function. The number of hash iterations, the presence and value of a salt as well as the hash functions to use are all indicated in a specific packet type called an S2K packet. In the case of Proton, Iterated and Salted S2K is used, with SHA256 as hash function.

If a passphrase is used, we say that it locks and unlocks a key instead of encrypting and decrypting it.

2.4.3 Signatures

Signatures are used to link the knowledge of a secret, which is generally recognized as a proof of identity, to some data. The main uses of this are to show ownership, give a certification (typically to recognize that some key belongs to a given user) or revoke it, and key binding (as mentioned in Section 2.4.2).

Note that signatures are not made for confidentiality, and may they therefore leak information about the data they sign. In our context, signatures are applied on top of unencrypted data, to give an authenticity guarantee on the content (i.e. the content has not changed since the signature was computed). Therefore, revealing information is problematic. To mitigate this, OpenPGP signatures compute a digest of the message and the signature algorithm is applied to that digest rather than to the message itself. This operation is part of the OpenPGP signature algorithm. We call the cryptographic scheme which combines the hashing with the DS scheme OpenPGP.DS.

The packets holding signatures are called Signature Packets. Their body starts with metadata which includes the signature version, the identifiers of the algorithms used to hash and sign the data, and signature subpackets, which are used to give usage indications about the data being signed. The last element in the Signature Packet is the signature itself, which is com-

puted over the hash of a concatenation of parts of the metadata, and the data to be signed. To facilitate checking that a signature is valid, an OpenPGP message may start with a packet called a One-Pass Signature Packet, which contains the information about the hashing and signing algorithms as well as the signing key. This allows to check the signature by doing one pass (hence the name) on the OpenPGP message.

A signature can be qualified as attached or detached depending on whether it is stored alongside the data it signs or not. If the encryption scheme used provides some authentication, attached signatures benefit from it as the signature is encrypted together with the plaintext, which makes them harder to tamper with. In the rest of this paper, we use OpenPGP.`AttSig` to denote the DS that returns the concatenation of a signature of a message and the message instead of just the signature.

OpenPGP.`AttSig` presents the same properties as OpenPGP.`DS`, as the only difference between the two is whether the signature and message are handled separately or together.

Signing Keys

When creating a new asymmetric key pair, it should immediately be signed in order to bind it to its owner, and specify its usage. This is done through a so-called self-signature, or subkey binding signature. Each user has a signing key which is bound to its identity, and which is what we call the primary key of the keyring of the user. To mark a key (which can be either a signing key or an encryption key) as belonging to that keyring, it needs to be signed by the primary key. Alongside the key itself, we also sign information about its usage, i.e. things like the algorithm for which it is suitable, the creation date and the validity status of the key. This whole thing forms a self-signature. Note that the primary key is also part of the keyring, and that as such, a self-signature also needs to be computed on that key.

It is also possible for a key to be signed by someone who doesn't own it. This is used to indicate that that user recognizes that key as belonging to some given user and establishes the level of trust that is put into that binding. If such a signature already exists, a revocation signature could also be emitted in order to withdraw that certification.

2.4.4 Encrypted Data

In OpenPGP, data is symmetrically encrypted and stored in symmetrically encrypted integrity protected data packets (SEIPD). Symmetrically encrypted data packets also exist for legacy, but they are deprecated.

Figure 2.19: In the modified CFB mode of encryption used by OpenPGP, the input to the regular CFB mode derived from the plaintext. r is a sixteen bytes value generated uniformly at random.

All data is encrypted in CFB mode, and the block cipher that is used depends on the implementation (Proton applications use AES-256). The initial vector (IV) is specified as all zeros. Instead of relying on the IV for randomness, OpenPGP prefixes random bytes to the data before it is encrypted. The number of bytes is equal to the block size of the cipher in bytes, plus two (16 + 2 = 18 bytes for AES-256), and the last two octets are a repetition of the two preceding.

To provide some integrity, the plaintext is appended two constant octets, and twenty octets corresponding to the SHA-1 hash of all of what precedes are appended afterwards. This hash is called Modification Detection Code (MDC) and serves to detect errors in the plaintext when decrypting.

We provide a graphical representation of the input construction in Fig. 2.19.

We call OpenPGP.SE the cryptographic scheme which creates an input as described above and applies AES-CFB on top of it. The key generation algorithm is the same as that of AES-CFB. The encryption algorithm first constructs the input and passes it to AES-CFB for encryption. The decryption algorithm applies AES-CFB decryption, and then checks that the format of the input is correct. If it is, it returns the plaintext, otherwise it returns $\perp$.

2.4.5 Usages

In the following, we define a few blocks that are frequently used in OpenPGP.

We start by describing the classical use of OpenPGP, where a public encryption key is used on a randomly generated session key that encrypts the data rather than on the data itself. This forms a PKE scheme which we call Hyb for hybrid because both symmetric and asymmetric cryptography are used internally. Its key generation algorithm is the key generation algorithm of OpenPGP.PKE and we specify its encryption algorithm in Algorithm 1 and its decryption algorithm in Algorithm 2. The entire scheme is illustrated in Fig. 2.20.

OpenPGP does not provide the possibility of symmetrically encrypting a key with a symmetric key directly. To circumvent this restriction, a locking mechanism is used, which we call PSE, for password symmetric encryption.

Algorithm 1: OpenPGP.Hyb.Enc$_{puk}(m)$

Input: $puk \in$ OpenPGP.PKE.$\mathcal{PUK}$
$m \in$ OpenPGP.SE.$\mathcal{M}$
Output: $c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$c \in$ OpenPGP.SE.$\mathcal{C}$

1 $sk \xleftarrow{\$}$ AES-256.Gen() //Generate a session key
2 $c_{sk} \xleftarrow{\$}$ OpenPGP.PKE.Enc$_{puk}(sk)$
3 $c \xleftarrow{\$}$ OpenPGP.SE.Enc$_{sk}(m)$
4 **return** c_{sk}, c

Algorithm 2: OpenPGP.Hyb.Dec$_{prk}(c_{sk}, c)$

Input: $prk \in$ OpenPGP.PKE.$\mathcal{PRK}$
$c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$c \in$ OpenPGP.SE.$\mathcal{C}$
Output: $sk \in$ OpenPGP.SE.$\mathcal{K}$
$m \in$ OpenPGP.SE.$\mathcal{M}$

1 $sk \leftarrow$ OpenPGP.PKE.Dec$_{prk}(c_{sk})$
2 $m \leftarrow$ OpenPGP.SE.Dec$_{sk}(c)$
3 **return** sk, m

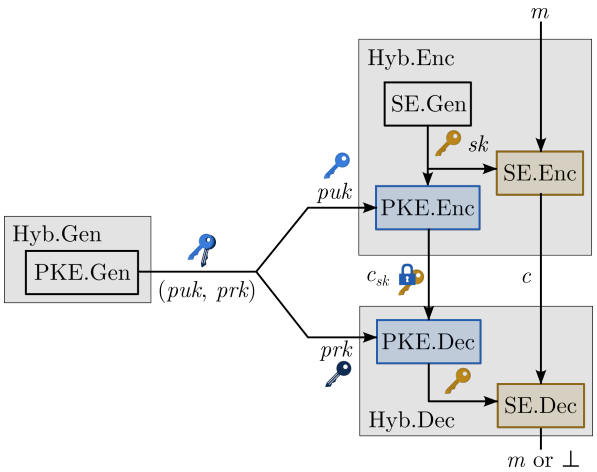

Figure 2.20: The Hyb PKE scheme.

This corresponds to a SE scheme where the key space is the set of possible passwords, $\texttt{Bytes}^*$, and the message space is the key space of either a SE or a PKE scheme. Note that the key generation is not uniformly random in practice because the password is picked by a user. We specify the encryption algorithm in Algorithm 3 and the decryption algorithm Algorithm 4. The entire scheme is illustrated in Fig. 2.21.

Algorithm 3: $\textsc{OpenPGP.PSE.Lock}_{pp}((puk, prk))$

Input: $pp \in \textsc{OpenPGP.PSE}.\mathcal{PP}$ `//The space of passphrases.`
$(puk, prk) \in \textsc{OpenPGP.SE}.\mathcal{K} \cup \textsc{OpenPGP.PKE}.\mathcal{K}$
Output: $l_{(puk,prk)} \in \textsc{OpenPGP.SE}.\mathcal{C}$

$sk \leftarrow \textsc{OpenPGP.S2K}(pp)$
$l_{(puk,prk)} \xleftarrow{\$} \textsc{OpenPGP.SE.Enc}_{sk}((puk, prk))$
return $l_{(puk,prk)}$

Algorithm 4: $\textsc{OpenPGP.PSE.Unlock}_{pp}(l_{(puk,prk)})$

Input: $pp \in \textsc{OpenPGP.PSE}.\mathcal{PP}$ `//The space of passphrases.`
$l_{(puk,prk)} \in \textsc{OpenPGP.SE}.\mathcal{C}$
Output: $(puk, prk) \in \textsc{OpenPGP.SE}.\mathcal{K} \cup \textsc{OpenPGP.PKE}.\mathcal{K}$

$sk \leftarrow \textsc{OpenPGP.S2K}(pp)$
$(puk, prk) \leftarrow \textsc{OpenPGP.SE.Dec}_{sk}(l_{(puk,prk)})$
return (puk, prk)

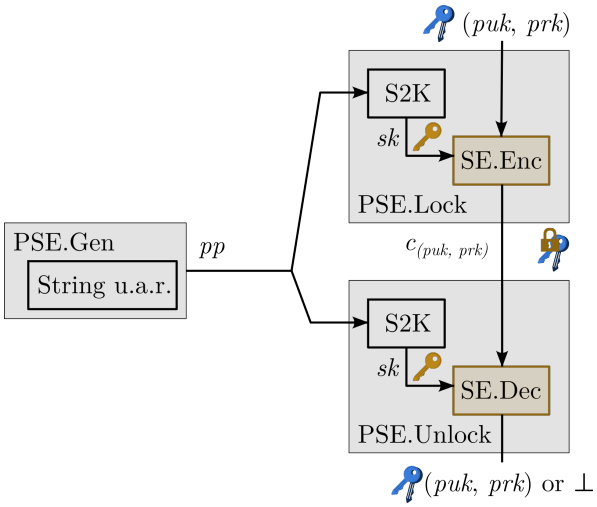

Figure 2.21: The PSE SE scheme. Here, we give an example where an encryption key pair is locked, but it can also be applied on a signature key pair.

Chapter 3

The Proton Drive Protocol

In this chapter, we provide a description of the Proton Drive protocol which serves as a basis for analysis. Note that there are several implementations of Proton Drive, namely one for each platform for which Proton Drive has a client; we aim to describe the underlying protocol rather than its implementations, unless they all noticeably deviate from the planned protocol in the same way.

Proton Drive is the cloud storage service from Proton AG. As such, it is used to store files and folders from users on the infrastructure provided by Proton. Ideally, the experience for a user should be as close as possible to that of accessing local files on their computer. However, as files are stored on a remote machine, it is not possible to replicate it exactly, because there are necessarily intermediates introducing authentication steps and delays which are not present on a local storage. Keeping interface elements such as the tree-like structure of the file system helps keep that familiar feeling.

The user interface is not the only place where Proton Drive retrieves that characteristic of file systems. The key hierarchy follows the same structure, with nodes in the tree corresponding to files, folders and helpers for user authentication or access control. Links between nodes represent the encryption of a child using the keys of its parents, where all cryptographic functions being called from the OpenPGP library.

In more detail, all keys used by Proton Drive are kept in a key hierarchy containing both signature and encryption keys. One signature key is generally used for every signature created by a user, because it is attached to the identity of that user, specifically to an email address belonging to that user. On the other hand, there is one encryption key associated to each node in the key hierarchy, and the encryption key of a node is used to encrypt both the key and data of its children. Altogether, the resulting key hierarchy structure closely follows that of the underlying file system, with only a few

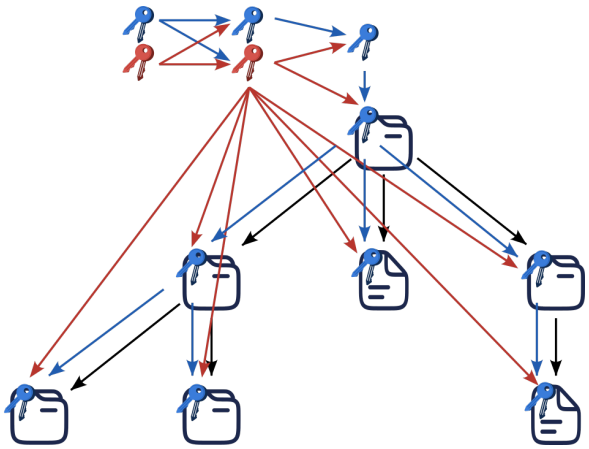

Figure 3.1: An example of a file system and its associated key hierarchy. Blue key pairs represent encryption keys, and red key pairs represent signature keys. Blue arrows represent the key pair at their start being used for encryption on the key pair and/or file or folder they point to. Red arrows are used in the same way but for signatures.

auxiliary nodes and the signature keys as outliers. An example illustration of a file system and its associated key hierarchy is given in Fig. 3.1.

As for a typical file system, once the user is authenticated to the server, every file or folder in the volume can be accessed from an entry point, the root folder, by trickling down the file system tree and decrypting the key and data at every level with the encryption key from the parent object and the right signature key. In each link between a node and its children, a combination of encryption and authentication through signatures is used. This combination is called signcryption, and the inverse operation is called unsigncryption.

Like all Proton applications, the Proton Drive protocol is built using the OpenPGP message formatting standard. Specifically, cryptographic functions are called from GopenPGP and OpenPGP.js, two libraries of which Proton AG is a maintainer. OpenPGP signatures and encryption are used to provide both authentication and confidentiality.

Proton Drive aims to secure online storage using end-to-end encryption (E2EE), endeavoring for the contents and metadata of any file, folder or key to only be accessible with the authorization of their owner. All the keys are generated on the client-side, and the processing of the data, metadata and keys is also done locally. The server only acts as an authenticator and a storage service. For each user, an allotted memory space which we call the volume is kept, and the whole file system of the user is stored on that volume. The user can then send and retrieve the encrypted data to and from the server.

We start by presenting the way Proton Drive uses OpenPGP in Section 3.1 in order to be able to concretely explain the different objects that are used in the protocol and how they combine in Section 3.2, and finish by giving the security goals that Proton Drive claims to achieve in Section 3.3.

3.1 Use of OpenPGP

In the Proton Drive protocol, OpenPGP.Hyb.Enc and OpenPGP.PSE.Lock, two basic OpenPGP building blocks that we presented in Section 2.4.5, are used together in order to chain several OpenPGP encryptions.

These two cryptographic schemes by themselves do not provide any ciphertext integrity guarantees. This is due to the fact that the part of the key which is used for encryption is public, and that public parts of keys are not encrypted. Therefore, anyone who can access the public key can produce a valid ciphertext for any message they want. In the setting of Proton Drive this is not desirable; instead, we expect an encryption to be bound to the identity of the user performing it by the use of a secret key. Because of this, a signature of the passphrase by the signing key linked to the identity of the user is added. Note that the passphrase is not encrypted before signature, because it is assumed that its high entropy suffices to make its recovery difficult using the signature only.

This forms a signcryption scheme, which we call hybrid signed encryption (HybSig). Its key generation functions are respectively that of the underlying PKE and DS schemes, namely

$$\text{OpenPGP.HybSig.SGen} = \text{OpenPGP.DS.Gen}$$

and

$$\text{OpenPGP.HybSig.RGen} = \text{OpenPGP.Hyb.Gen}.$$

We specify the pseudocode of the signcryption algorithm in Algorithm 5 and of the unsigncryption in Algorithm 6, and give an illustration of HybSig in Fig. 3.2.

3.2 Structure

Proton Drive is heavily centered around storage, with the only operation which is not directly storage-related being user authentication, which uses an interactive protocol between the client and server.

The construction that Proton Drive uses for file hierarchy is the same as the one that is used in classical file systems. Semantically, there are two objects that exist, folders and files. A folder is a container, which can hold other folders or files, and a file is an object containing file data. We call the objects

Algorithm 5: ProtonDrive.HybSig.$\mathsf{SC}_{sik,puk}((puk', prk'))$

Input: $sik \in$ OpenPGP.Sign.$\mathcal{PrK}$
$puk \in$ OpenPGP.PKE.$\mathcal{PuK}$
$(puk', prk') \in$ OpenPGP.PKE.$\mathcal{K} \cup$ OpenPGP.DS.$\mathcal{K}$
Output: $c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$\sigma_{\text{pp}} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{pp} \in$ OpenPGP.SE.$\mathcal{C}$
$l_{(puk',prk')} \in$ OpenPGP.SE.$\mathcal{C}$

1 $pp \xleftarrow{\$}$ OpenPGP.PSE.Gen() //Generate a passphrase.
2 $\sigma_{pp} \leftarrow$ OpenPGP.DS.$\mathsf{Sign}_{\text{sik}}(pp)$
3 $c_{sk}, c_{pp} \xleftarrow{\$}$ OpenPGP.Hyb.$\mathsf{Enc}_{\text{puk}}(pp)$
4 $l_{(puk',prk')} \xleftarrow{\$}$ OpenPGP.PSE.$\mathsf{Lock}_{\text{pp}}((puk', prk'))$
5 **return** c_{sk}, σ_{pp}, c_{pp}, $l_{(puk',prk')}$

Algorithm 6: ProtonDrive.HybSig.$\mathsf{USC}_{vk,prk}(c_{sk}, \sigma_{pp}, c_{pp}, l_{(puk',prk')})$

Input: $vk \in$ OpenPGP.Sign.$\mathcal{PuK}$
$prk \in$ OpenPGP.PKE.$\mathcal{PrK}$
$c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$\sigma_{pp} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{pp} \in$ OpenPGP.SE.$\mathcal{C}$
$l_{(puk',prk')} \in$ OpenPGP.SE.$\mathcal{C}$
Output: $(puk', prk') \in$ OpenPGP.PKE.$\mathcal{K} \cup$ OpenPGP.DS.$\mathcal{K} \cup \{\perp\}$

1 $sk, pp \leftarrow$ OpenPGP.Hyb.$\mathsf{Dec}_{prk}(c_{sk}, c_{pp})$
//Verify the passphrase.
2 **if** OpenPGP.DS.$\mathsf{Vfy}_{vk}(\sigma_{pp}, pp) = \perp$ **then**
3 **return** $\perp$
4 $(puk', prk') \leftarrow$ OpenPGP.PSE.$\mathsf{Unlock}_{pp}(l_{(puk',prk')})$
5 **return** (puk', prk')

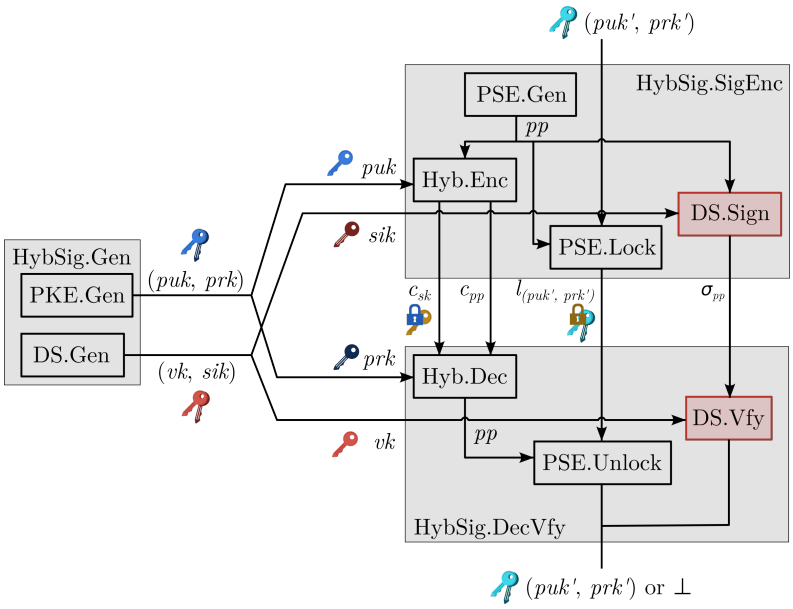

Figure 3.2: The `HybSig` signcryption scheme.

contained in a folder its children, and the folder itself is the parent of these objects. In a non-encrypted file system, the link between a parent folder and its children is done with pointers, namely a list of references to every child. For an encrypted storage, we additionally want a key to be associated to each node, which is used as encryption key for its children if the node is a folder, or for the file data itself if the node is a file.

The aim of encryption is to provide confidentiality, but we also want authentication on the files. Every folder or file belongs to a user, and the user provides a signature in every node of its file system, or of every file it modifies.

Syntactically, Proton Drive uses a several object types, to represent users, to provide and manage access to the files and folders, and to to keep the names and metadata of files and folders. Except for the topmost level, every object has a parent whose key is used to encrypt and decrypt the object, and retrieving the information contained in a specific node can hence be done by decrypting the whole chain from the topmost level down to the node in question (while checking that all signatures are correct). In the following subsection, we present the different object types used in Proton Drive, and give the algorithms used to encrypt and decrypt them.

3.2.1 Users and Addresses

A user is an entity interacting with Proton Drive through an account using an application on their device or browser which we call the client. Upon

registering, a user provides the server with some value derived from its password, which is used for authentication at each login. The server only sends back the data that a user stored if the authentication was successful. We represent this as a new authenticated session being established between the client and server upon calling SRP.*Server*.Vfy$_{\rightleftharpoons}$[1] with the correct password. The user can then query any value in its volume using this session.

Users are identified by their ID, and each have a key which allows them to sign and asymmetrically encrypt data. One or more email addresses, each with their own address key, are linked to a user. If several addresses belong to the same user, one of them is defined as the default address and, unless otherwise specified, that address provides the signing key and encryption address key used in drive. In the rest of this report, we only consider the default address. This could be extended to have an address that is shared between several people in the future (e.g. for company services).

Both the user key and address key are actually keyrings composed of a primary signing key and an encryption key. In general cases, more than one key of each type can be contained in a keyring, in which case one of them needs to be marked as the default, but we ignore this possibility in our analysis, as only one signing key and one encryption key are used. The way keyrings are formed is described in Section 2.4.2.

The keys are locked for storage on the server. We call ProtonDrive.`UserKey` the SE scheme that is used to lock and unlock them. The key generation function corresponds to the user picking a string as a password, and is therefore not uniformly random. We give the pseudocode of the signcryption of a user key in Algorithm 7 and of its unsigncryption in Algorithm 8, and give an illustration in Fig. 3.3. Note that we also include user authentication in the scheme, and that decryption can fail if the user provides a bad password.

We call ProtonDrive.`AddressKey` the signcryption scheme that is used to encrypt and decrypt the share. The sender key generation algorithm returns the user signature key pair as the DS key pair and the receiver key generation algorithm returns the user encryption key as the PKE key pair. We define the pseudocode of the signcryption algorithm in Algorithm 9 and of the unsigncryption algorithm in Algorithm 10, and give an illustration of the entire scheme in Fig. 3.4.

Throughout the rest of the structure, objects use the signing key of their owner address in order to sign their attributes. To this end, they keep track of the identity of the address who created them.

[1] We use $\rightleftharpoons$ as a subscript to emphasize that this is an interactive call to the server.

Algorithm 7: ProtonDrive.UserKey.Lock($password, (puk_{user}, prk_{user})$)

Input: $password \in \texttt{String}$
$(puk_{user}, prk_{user}) \in \text{OpenPGP.PKE}.\mathcal{K} \cup \text{OpenPGP.DS}.\mathcal{K}$
Output: $salt_{\text{SRP}} \in \texttt{Bytes}^{16}$
$srpVerifier \in \mathbb{Z}^*_{\text{SRP}.N}$
$salt_{key} \in \texttt{Bytes}^{16}$
$locked_{(puk_{user}, prk_{user})} \in \text{OpenPGP.SE}.\mathcal{C}$

```
//Create a verifier for SRP authentication.
//The verifier value is sent to the server and kept by it.
```

1 $salt_{\text{SRP}} \xleftarrow{\$} \texttt{Bytes}^{16}$
2 $h_{password} \leftarrow \text{bcrypt}(srpSalt, password)$
3 **for** i from 0 to 3 **do**
4 $\quad h_{password} \leftarrow \text{OpenPGP.Hash.h}(h_{password})$
5 $srpVerifier \leftarrow \text{SRP.GenerateVerifier}(h_{password})$

```
//Lock the user key.
```

6 $salt_{key} \xleftarrow{\$} \texttt{Bytes}^{16}$
7 $pp \leftarrow \text{bcrypt}(keySalt, password)$
8 $l_{(puk_{user}, prk_{user})} \xleftarrow{\$} \text{OpenPGP.PSE.Lock}_{pp}((puk_{user}, prk_{user}))$
9 **return** $salt_{\text{SRP}}, srpVerifier, salt_{key}, l_{(puk_{user}, prk_{user})}$

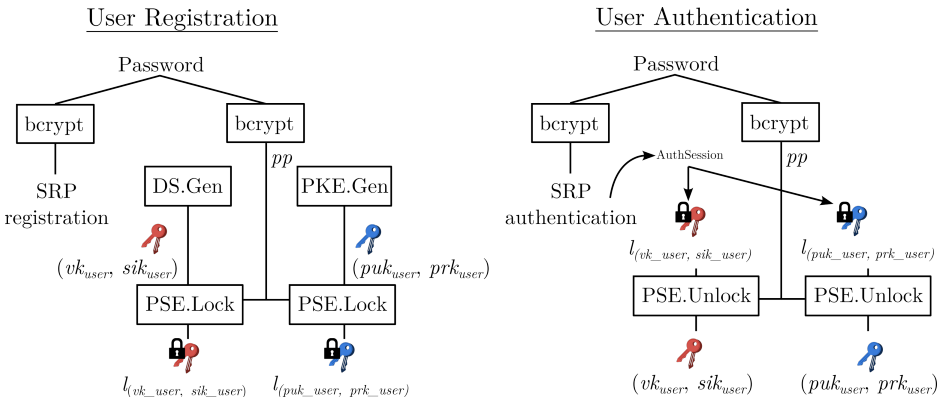

Figure 3.3: User registration and authentication.

Algorithm 8: ProtonDrive.UserKey.Unlock($password, srpSalt, srpVerifier, keySalt$)

Input: $password \in \texttt{String}$
$salt_{\text{SRP}} \in \texttt{Bytes}^{16}$
$srpVerifier \in \mathbb{Z}^*_{\text{SRP}.N}$
$salt_{key} \in \texttt{Bytes}^{16}$
Output: $(puk_{user}, prk_{user}) \in \text{OpenPGP.PKE}.\mathcal{K} \cup \text{OpenPGP.DS}.\mathcal{K}$

```
//Verifying is done by communicating with the server, who establishes an
  authenticated session if the client can prove that it knows the
  password.  The value password itself is never sent to the server.
```

$h_{password} \leftarrow \text{bcrypt}(salt_{\text{SRP}}, password)$
for i from 0 to 3 **do**
  $h_{password} \leftarrow \text{OpenPGP.Hash.h}(h_{password})$
$authSession \leftarrow \text{SRP}.Server.\text{Vfy}_{\rightleftharpoons}(h_{password})$
if $authSession = \perp$ **then**
  return $\perp$

```
//Retrieve the locked key.
```

$l_{(puk_{user}, prk_{user})} \leftarrow Server.getLockedUserSubkey(authSession)$

```
//Unlock the key.
```

$pp \leftarrow \text{bcrypt}(salt_{key}, password)$
$(puk_{user}, prk_{user}) \leftarrow \text{OpenPGP.PSE.Unlock}_{pp}(l_{(puk_{user}, prk_{user})})$
return (puk_{user}, prk_{user})

Algorithm 9: ProtonDrive.AddressKey.$\text{SC}_{puk_{user}, sik_{user}}((puk_{addr}, prk_{addr}))$

Input: $puk_{user} \in \text{OpenPGP.PKE}.\mathcal{PuK}$
$sik_{user} \in \text{OpenPGP.DS}.\mathcal{PrK}$
$(puk_{addr}, prk_{addr}) \in \text{OpenPGP.PKE}.\mathcal{K} \cup \text{OpenPGP.DS}.\mathcal{K}$
Output: $c_{sk} \in \text{OpenPGP.PKE}.\mathcal{C}$
$c_{pp_{addr}} \in \text{OpenPGP.SE}.\mathcal{C}$
$\sigma_{pp_{addr}} \in \text{OpenPGP.DS}.\mathcal{S}$
$l_{(puk_{addr}, prk_{addr})} \in \text{OpenPGP.SE}.\mathcal{C}$

return ProtonDrive.HybSig.$\text{SC}_{puk_{user}, sik_{user}}((puk_{addr}, prk_{addr}))$

Algorithm 10: ProtonDrive.AddressKey.USC$_{prk_{user},vk_{user}}(c_{sk}, c_{pp_{addr}}, \sigma_{pp_{addr}}, l_{(puk_{addr},prk_{addr})})$

Input: $prk_{user} \in$ OpenPGP.PKE.$\mathcal{PrK}$
$vk_{user} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{pp_{addr}} \in$ OpenPGP.SE.$\mathcal{C}$
$\sigma_{pp_{addr}} \in$ OpenPGP.DS.$\mathcal{S}$
$l_{(puk_{addr},prk_{addr})} \in$ OpenPGP.SE.$\mathcal{C}$
Output: $(puk_{addr}, prk_{addr}) \in$ OpenPGP.PKE.$\mathcal{K} \cup$ OpenPGP.DS.$\mathcal{K} \cup \{\perp\}$

1 **return** ProtonDrive.HybSig.USC$_{prk_{user},vk_{user}}(c_{sk}, c_{pp_{addr}}, \sigma_{pp_{addr}}, l_{(puk_{addr},prk_{addr})})$

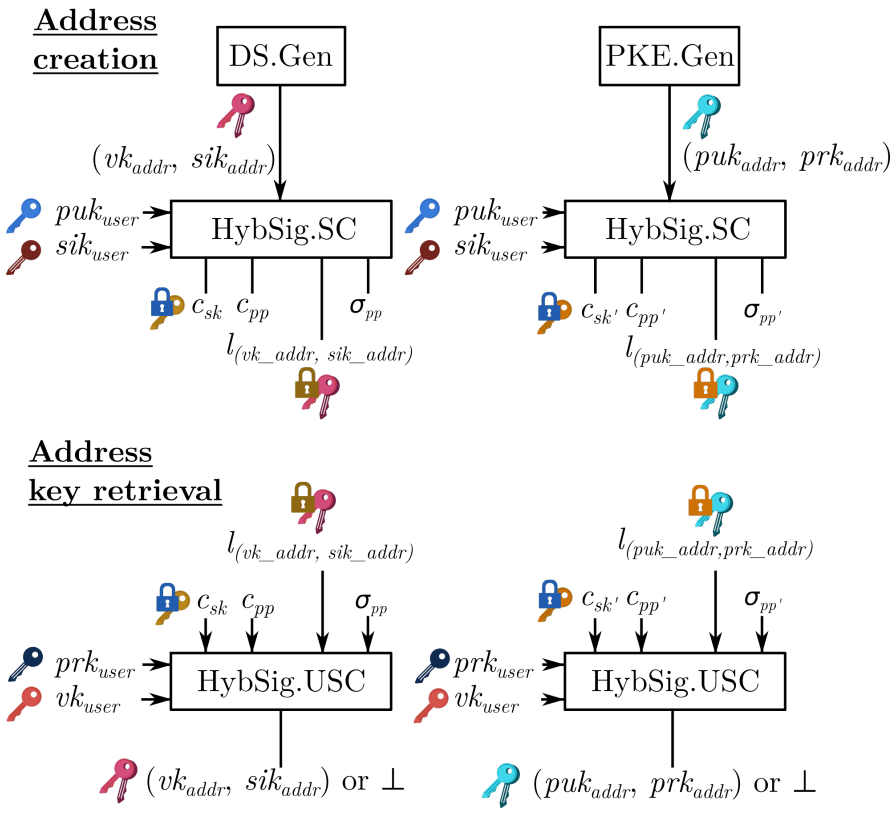

Figure 3.4: Address registration and authentication.

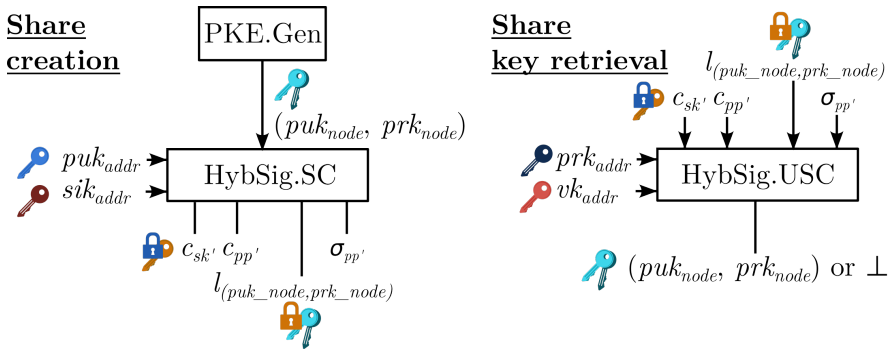

Figure 3.5: Share creation and retrieval.

3.2.2 Shares

Shares are the entry points to the drive file system. They can point to a folder or a file. Each file hierarchy has a root share, which represents the entry point to the basis of a volume, the root folder, but other shares can also be created lower in the hierarchy.

As a share is the first object encountered when entering the file system, it directly uses the encryption key of the address for asymmetric encryption. To encrypt the object that it points to, the share needs to have its own asymmetric key, which we call the Share Key. Additionally, every share has a session key, a share passphrase, and a signature over that passphrase.

We call ProtonDrive.Share the signcryption scheme which is used to encrypt and decrypt the shares. The sender key generation algorithm returns the address signature key pair as the DS key pair and the receiver key generation algorithm returns the address encryption key as the PKE key pair. We give the pseudocode of the signcryption algorithm in Algorithm 11 and of the unsigncryption algorithm in Algorithm 12, and an illustration of the whole scheme in Fig. 3.5.

Algorithm 11: ProtonDrive.Share.$SC_{puk_{owner},sik_{owner}}((puk_{share}, prk_{share}))$

Input: $puk_{owner} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$sik_{owner} \in$ OpenPGP.DS.$\mathcal{PrK}$
$(puk_{share}, prk_{share}) \in$ OpenPGP.PKE.$\mathcal{K}$

Output: $c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{pp_{share}} \in$ OpenPGP.SE.$\mathcal{C}$
$\sigma_{pp_{share}} \in$ OpenPGP.DS.$\mathcal{S}$
$l_{(puk_{share},prk_{share})} \in$ OpenPGP.SE.$\mathcal{C}$

1 **return** ProtonDrive.HybSig.$SC_{puk_{owner},sik_{owner}}((puk_{share}, prk_{share}))$

Algorithm 12: ProtonDrive.Share.USC$_{prk_{owner},vk_{owner}}(encSessionKey, encSharePassphrase, signSharePassphrase, l_{(puk_{share},prk_{share})})$

Input: $prk_{owner} \in$ OpenPGP.PKE.$\mathcal{PrK}$
$vk_{owner} \in$ OpenPGP.DS.$\mathcal{PuK}$
$c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{pp_{share}} \in$ OpenPGP.SE.$\mathcal{C}$
$\sigma_{pp_{share}} \in$ OpenPGP.PKE.$\mathcal{C}$
$l_{(puk_{share},prk_{share})} \in$ OpenPGP.SE.$\mathcal{C}$
Output: $(puk_{share}, prk_{share}) \in$ OpenPGP.PKE.$\mathcal{K}$

1 **return** ProtonDrive.HybSig.USC$_{prk_{user},vk_{user}}(c_{sk}, c_{pp_{share}}, \sigma_{pp_{share}}, l_{(puk_{share},prk_{share})})$

As its name suggests, a share can be used for sharing folders and files with other people or accounts. For now, only sharing using a URL is implemented, but it is also planned to add sharing between Proton accounts in the future. The idea behind both kinds of sharing is to reencrypt the session key of a share to make it accessible to whoever it is to be shared with.

URL Share

A URL share is an object used to give access to a regular share through a URL link, which is generated by the owner of the share. The link is made up of a part that is common to every shared file, `https://drive.proton.me/urls/`, followed by ten characters between A-Z and 0-9 that identify where the share is pointing. To lock the file, a password is created, that has a mandatory random part and an optional part that can be chosen by the owner. The random part consists of nine bytes encoded in URL-safe no-padding base64. The custom part is a Unicode string of at most fifty characters. The entire URL password is then formed by concatenating both parts, with the random part coming first.

To access the URL share, one can either put the URL password in the fragment of the URL, which is the part after the # symbol that the queried server does not receive, or be prompted for it when entering the web page. As for the user password, the URL password is first checked to be correct with SRP before being passed to bcrypt. The resulting hash is used to encrypt a URL passphrase, which locks the session key of the underlying share.

The owner of the URL share also stores the URL password encrypted with its address encryption key. Therefore, the owner does not need to remember or store the password in plaintext in order to be able to access the share through the URL share.

We call the scheme that is used for the creation and decryption of URL

shares ProtonDrive.ShareURL and define it as $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{DecURL}, \mathsf{DecUser})$, where

- $\mathsf{Gen} : \emptyset \to$ OpenPGP.PKE.$\mathcal{K} \times \{a, ..., z, A, ..., Z, 0..., 9\}^{10} \times \mathtt{Chars}^{\leq 50}$, the password generation algorithm. The key pair is the address encryption key pair of the creator of the URL share, the second part is generated uniformly at random upon creation of the URL share, and the last part can be chosen by the user but left blank by default.
- Enc as defined in Algorithm 13,
- DecURL as defined in Algorithm 14,
- DecUser as defined in Algorithm 15.

Note that this scheme differs from a regular SE or PKE scheme in that it provides two ways to retrieve the URL share, namely through the encrypted URL password for the URL share owner, or through the URL link for others.

Proton to Proton Share

While it is not implemented yet, sharing between Proton members can be done by creating a new copy of the session key, encrypted with the address key of the user we want to share it with.

We call ProtonDrive.ShareInternal the PKE scheme that is used to encrypt and decrypt the Proton to Proton share. The key generation algorithm returns the address encryption key of the member who we want to share the share to as the PKE key pair and the address signature key pair of the share owner as the DS key pair. We define the encryption algorithm in Algorithm 16 and the decryption algorithm in Algorithm 17.

3.2.3 Folders

A folder is composed of two main parts, namely a link and a node, a basic structure that it shares with files. We present these two objects in the context of a folder first and then explain how they are also used in files. The link is the entry point to the folder for its parents, and the node connects it to its children.

Link

A link is both responsible for linking a folder with any object pointing to it, be it a parent folder or shares, and for holding data relevant to the folder, namely its name and some extended attributes (xAttr). The name is a string (with no length limit), whereas xAttr come in the form of a JSON document in UTF-8 format storing information such as the last modification time. Both are stored signed and encrypted, with the addition of being compressed

Algorithm 13: ProtonDrive.ShareURL.$\mathsf{Enc}_{puk_{owner}}(customPassword, sk_{share})$

Input: $puk_{owner} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$randomPassword \in \{a, ..., z, A, ..., Z, 0..., 9\}^{10}$
$customPassword \in \texttt{Chars}^{\leq 50}$
$sk_{share} \in$ OpenPGP.SE.$\mathcal{K}$

Output: $c_{urlPassword} \in$ OpenPGP.PKE.$\mathcal{C}$
$salt_{SRP} \in \texttt{Bytes}^{16}$
$srpVerifier \in$ SRP.$\mathcal{V}$
$salt_{pp} \in \texttt{Bytes}^{16}$
$l_{sk_{share}} \in$ OpenPGP.PSE.$\mathcal{C}$

```
//Derive password and encrypt it for storage.
```
1 $urlPassword \leftarrow randomPassword \,||\, customPassword$
2 $c_{urlPassword} \xleftarrow{\$}$ OpenPGP.PKE.$\mathsf{Enc}_{pk_{owner}}(urlPassword)$

```
//Create a verifier for SRP authentication.
```
3 $salt_{SRP} \xleftarrow{\$} \texttt{Bytes}^{16}$
4 $srpVerifier \leftarrow \mathsf{SRP.GenerateVerifier}(bcrypt(salt_{SRP}, urlPassword))$

```
//Generate and encrypt a passphrase.
```
5 $salt_{passphrase} \xleftarrow{\$} \texttt{Bytes}^{16}$
6 $urlPassphrase \leftarrow \mathsf{bcrypt}(salt_{pp}, urlPassword)[29:]$ `//Remove the bcrypt prefix.`

```
//Lock the share session key.
```
7 $l_{sk_{share}} \xleftarrow{\$}$ OpenPGP.PSE.$\mathsf{Lock}_{urlPassphrase}(sk_{share})$
8 **return** $c_{urlPassword}, salt_{SRP}, srpVerifier, salt_{pp}, l_{sk_{share}}$

first for xAttr. Moreover, for non root folders, a HMAC tag of the name is kept, which allows for quick lookup and prevents name duplicates without needing to decrypt the name. Note that this value is not used for verification purposes. The choice of a keyed primitive over a simple hash function here serves to prevent the server from being able to check whether some folder contains a file or folder of a given name, providing some metadata hiding.

We call ProtonDrive.Name the combination of a signcryption and MAC scheme which is used to encrypt and decrypt file or folder names. The sender key generation algorithm of the signcryption returns the address signature key pair as the DS key pair, its receiver key generation algorithm returns the parent node or share encryption key as the PKE key pair. Additionally, the tag key for the MAC scheme is a key belonging to the parent node and is encrypted with the encryption key of the parent. We give the pseudocode of the signcryption and MAC algorithm in Algorithm 18 and of

Algorithm 14: ProtonDrive.ShareURL.DecURL($url, randomPassword, customPassword, salt_{\text{SRP}}, salt_{pp}$)

Input: $url \in \texttt{String}$
$randomPassword \in \{a, ..., z, A, ..., Z, 0..., 9\}^{10}$
$customPassword \in \texttt{Chars}^{\leq 50}$
$salt_{SPR} \in \texttt{Bytes}^{16}$
$salt_{pp} \in \texttt{Bytes}^{16}$
Output: $sk_{share} \in$ OpenPGP.SE.$\mathcal{K}$

```
//Use the URL password to authenticate to the server and retrieve the
  locked session key.
```
$urlPassword \leftarrow randomPassword \parallel customPassword$
$authSession \leftarrow \text{SRP}.Server.\text{Vfy}_{\rightleftharpoons}(url, bcrypt(salt_{SRP}, urlPassword))$
if $authSession = \perp$ **then**
  return $\perp$

```
//Recompute the passphrase.
```
$urlPassphrase \leftarrow \text{bcrypt}(salt_{pp}, urlPassword)[29:]$

```
//Get and unlock the share session key.
```
$l_{sk_{share}} \leftarrow Server.getLockedShareSessionKey(authSession)$
$sk_{share} \leftarrow \text{OpenPGP.PSE.Unlock}_{urlPassphase}(l_{sk_{share}})$
return sk_{share}

Algorithm 15: ProtonDrive.ShareURL.DecUser$_{prk_{owner}}$($c_{urlPassword}, salt_{pp}, l_{sk_{share}}$)

Input: $prk_{owner} \in$ OpenPGP.PKE.$\mathcal{PRK}$
$c_{urlPassword} \in$ OpenPGP.PKE.$\mathcal{C}$
$salt_{pp} \in \texttt{Bytes}^{16}$
$l_{sk_{share}} \in$ OpenPGP.PSE.$\mathcal{C}$
Output: $sk_{share} \in$ OpenPGP.SE.$\mathcal{K}$

```
//Use the URL password to authenticate to the server and retrieve the
  locked session key.
```
$urlPassword \leftarrow \text{OpenPGP.PKE.Dec}_{prk_{owner}}(c_{urlPassword})$
$urlPassphrase \leftarrow \text{bcrypt}(salt_{pp}, urlPassword)[29:]$

```
//Unlock the share session key.
```
$sk_{share} \leftarrow \text{OpenPGP.PSE.Unlock}_{urlPassphrase}(l_{sk_{share}})$
return sk_{share}

Algorithm 16: ProtonDrive.ShareInternal.Enc$_{puk_{other\ user}}(sk_{share})$

Input: $puk_{other\ user} \in$ OpenPGP.PKE.$\mathcal{PUK}$
$sk_{share} \in$ OpenPGP.SE.$\mathcal{K}$
Output: $c_{sk_{share}} \in$ OpenPGP.PKE.$\mathcal{C}$

1 **return** OpenPGP.PKE.Enc$_{puk_{other\ user}}(sk_{share})$

Algorithm 17: ProtonDrive.ShareInternal.Dec$_{prk_{other\ user}}(c_{sk_{share}})$

Input: $prk_{other\ user} \in$ OpenPGP.PKE.$\mathcal{PRK}$
$c_{sk_{share}} \in$ OpenPGP.PKE.$\mathcal{C}$
Output: $sk_{share} \in$ OpenPGP.SE.$\mathcal{K}$

1 **return** OpenPGP.PKE.Dec$_{prk_{other\ user}}(c_{sk_{share}})$

the unsigncryption and verifying algorithm in Algorithm 19. We moreover give a graphical representation of the signcryption algorithm in Fig. 3.6.

Algorithm 18: ProtonDrive.Name.SCTag$_{puk_{parent},sik_{owner},hk_{parent}}(name)$

Input: $puk_{parent} \in$ OpenPGP.PKE.$\mathcal{PUK}$
$sik_{owner} \in$ OpenPGP.DS.$\mathcal{PRK}$
$tk_{parent} \in$ OpenPGP.HMAC.$\mathcal{K}$
$name \in$ String
Output: $c_{sk_{name}} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{\sigma||name} \in$ OpenPGP.SE.$\mathcal{C}$
$\tau_{name} \in$ OpenPGP.HMAC.$\mathcal{T}$

```
//HMAC the name with the HMAC key of the parent.  Use the HMAC to check
  whether the name is a duplicate, if yes break.
```

1 $\tau_{name} \xleftarrow{\$}$ OpenPGP.HMAC.Tag$_{tk_{parent}}(name)$
2 **if** the name is a duplicate **then**
3 | **return** $\perp$

```
//Sign and encrypt the name.
```

4 $\sigma_{name} \xleftarrow{\$}$ OpenPGP.AttSig.Sign$_{sik_{owner}}(name)$
5 $(c_{sk_{name}}, c_{\sigma||name}) \xleftarrow{\$}$ OpenPGP.Hyb.Enc$_{puk_{parent}}(\sigma_{name})$
6 **return** $c_{sk_{name}}, c_{\sigma||name}, \tau_{name}$

We call ProtonDrive.XAttr the signcryption scheme which is used to encrypt and decrypt xAttr. The sender key generation algorithm returns the address signature key pair as the DS key pair and the receiver key genera-

Algorithm 19: ProtonDrive.Name.USCVfy$_{prk_{parent}, vk_{owner}, tk_{parent}}(c_{sk_{name}}, c_{\sigma||name})$

Input: $prk_{parent} \in$ OpenPGP.PKE.$\mathcal{PRK}$
$vk_{owner} \in$ OpenPGP.DS.$\mathcal{PUK}$
$tk_{parent} \in$ OpenPGP.HMAC.$\mathcal{K}$
$c_{sk_{name}} \in$ OpenPGP.SE.$\mathcal{C}$
$c_{\sigma||name} \in$ OpenPGP.SE.$\mathcal{C}$
Output: $name \in$ OpenPGP.SE.$\mathcal{C}$

1 $\sigma_{name} \leftarrow$ OpenPGP.Hyb.Dec$_{prk_{parent}}(c_{sk_{name}}, c_{\sigma||name})$
2 $name \leftarrow$ OpenPGP.AttSig.Vfy$_{vk}(\sigma_{name})$
3 **return** $name$

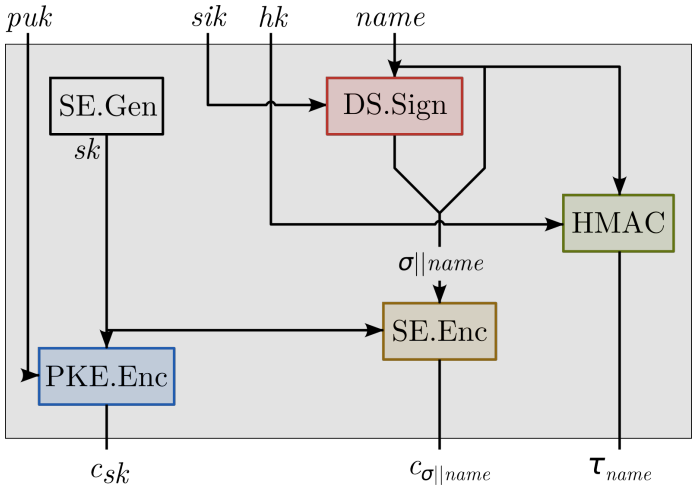

Figure 3.6: The name signcryption algorithm.

tion algorithm returns the parent encryption key as the PKE key pair. We give the pseudocode of the signcryption algorithm in Algorithm 20 and of the unsigncryption algorithm in Algorithm 21. We moreover give a graphical representation of the signcryption algorithm in Fig. 3.7.

Should there be a future version of Proton Drive implementing symbolic links, implying that a folder could be accessed from more than one place under more than one name, it would be possible to keep more than one link for any given folder.

Node

A node is the object representing the point in the file hierarchy where the folder is situated. It contains information that links it to its children, like the hash key that serve for hashing their name, the node key which is used to encrypt their link key packets, as well as the passphrase and passphrase

Algorithm 20: ProtonDrive.XAttr.SC$_{puk_{parent},sik_{owner}}(xAttr)$

Input: $puk_{parent} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$sik_{owner} \in$ OpenPGP.DS.$\mathcal{PrK}$
$xAttr \in$ String
Output: $c_{\sigma_{comprXAttr}} \in$ OpenPGP.PKE.$\mathcal{C}$

$comprXAttr \leftarrow$ OpenPGP.Comp.Compr$(xAttr)$
$\sigma_{comprXAttr} \leftarrow$ OpenPGP.DS.Sign$_{sik_{owner}}(comprXAttr)$
$c_{\sigma_{comprXAttr}} \xleftarrow{\$}$ OpenPGP.PKE.Enc$_{puk_{parent}}(\sigma_{comprXAttr})$
return $c_{\sigma_{comprXAttr}}$

Algorithm 21: ProtonDrive.XAttr.USC$_{prk_{parent},vk_{owner}}(c_{\sigma_{comprXAttr}})$

Input: $prk_{parent} \in$ OpenPGP.PKE.$\mathcal{PrK}$
$vk_{owner} \in$ OpenPGP.DS.$\mathcal{PuK}$
$c_{\sigma_{comprXAttr}} \in$ OpenPGP.PKE.$\mathcal{C}$
Output: $xAttr \in$ String

$\sigma_{comprXAttr} \leftarrow$ OpenPGP.PKE.Dec$_{prk_{parent}}(c_{\sigma_{comprXAttr}})$
$comprXAttr \leftarrow$ OpenPGP.AttSig.Vfy$_{vk_{owner}}(\sigma_{comprXAttr})$
if $comprAttr = \perp$ **then**
  return $\perp$
$xAttr \leftarrow$ OpenPGP.Comp.Decompr$(xAttr)$
return $xAttr$

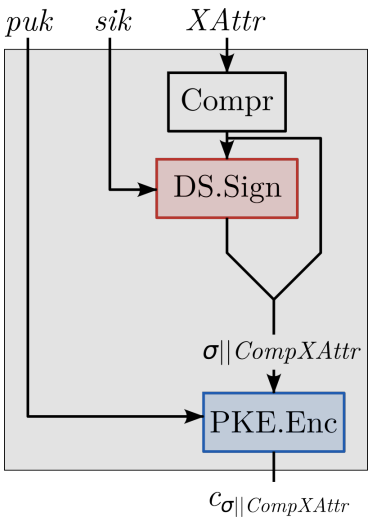

Figure 3.7: The xAttr signcryption algorithm.

signature which are used to decrypt the node key. The list of the children of a folder is stored in clear.

Creating a folder

In Algorithm 22, we describe the creation of a the link and node objects that are the basis for both folders and files. Of the outputs, everything related to the name and XAttr belongs to the Link of the folder, whereas the other outputs, such as the encrypted key pair and the signcrypted HMAC key are attributes of the Node.

We call ProtonDrive.LinkAndNode the combination of the signcryption scheme for the link and node key and the signcryptions of the name and xAttr of a file or folder. The key generation algorithm returns the parent node or share encryption key as the PKE key pair and the address signature key pair of the owner as the DS key pair. We give the pseudocode of the signcryption algorithm in Algorithm 22 and of the unsigncryption algorithm in Algorithm 23.

In the current state of the drive, it is possible to access the same folder (under the same name) from different objects, because apart from its parent (if the current folder is not the root), there could also be shares pointing to it. This is allowed by keeping as many copies of the session keys as there are accesses to the folder.

3.2.4 Files

Files contain the same structure as folders do, with two exceptions. First, the node of a file does not contain any hash key. Second, the node key encrypts a session key instead, which is used to encrypt the file data. Additionally, files contain the data in itself, and a few data structures serving the confidentiality and integrity of the packet.

Data Block

Data blocks correspond to the file chunks encrypted using the node session key. Their size is at most 4MiB, and they are OpenPGP SEIPD packets using AES-256 as block cipher, as described in Section 2.4.4.

The reason for chunking the data before encryption instead of letting OpenPGP do bigger packets using partial length (as we described in Section 2.4.1) is to be able to have MACs on smaller data chunks instead of on the whole file. This could allow re-encrypting only the relevant data when some but not all chunks are modified, thus reducing the amount of volume being used by a revision. That feature is not implemented yet, but is part of the documentation, meaning that there is a strive to integrate it at some point.

Algorithm 22: ProtonDrive.LinkAndNode.$\mathsf{SC}_{puk_{parent},sik_{owner},hk_{parent}}($
$name, xAttr, (puk_{node}, prk_{node}), tk_{node})$

Input: $puk_{parent} \in$ OpenPGP.PKE.$\mathcal{PUK}$
$sik_{owner} \in$ OpenPGP.DS.$\mathcal{PRK}$
$tk_{parent} \in$ OpenPGP.HMAC.$\mathcal{K}$
$name \in$ String
$xAttr \in$ String
$(puk_{node}, prk_{node}) \in$ OpenPGP.PKE.$\mathcal{K}$
$tk_{node} \in$ OpenPGP.HMAC.$\mathcal{K}$

Output: $c_{sk_{name}} \in$ OpenPGP.SE.$\mathcal{C}$
$c_{\sigma \| name} \in$ OpenPGP.SE.$\mathcal{C}$
$\tau_{name} \in$ OpenPGP.HMAC.$\mathcal{H}$
$c_{sk} \in$ OpenPGP.PKE.$\mathcal{C}$
$\sigma_{pp} \in$ OpenPGP.DS.$\mathcal{S}$
$c_{pp} \in$ OpenPGP.SE.$\mathcal{C}$
$l_{(puk_{node},prk_{node})} \in$ OpenPGP.SE.$\mathcal{C}$
$c_{\sigma_{tk_{node}}} \in$ OpenPGP.PKE.$\mathcal{C}$
$c_{\sigma_{comprXAttr}} \in$ OpenPGP.PKE.$\mathcal{C}$

1 $(c_{sk_{name}}, c_{\sigma \| name}, \tau_{name})$
$\xleftarrow{\$}$ ProtonDrive.Name.SC.$\mathsf{Tag}_{puk_{parent},sik_{owner},tk_{parent}}(name)$

2 $(c_{sk}, \sigma_{pp}, c_{pp}, l_{(puk_{node},prk_{node})})$
$\xleftarrow{\$}$ ProtonDrive.HybSig.$\mathsf{SC}_{puk_{parent},sik_{owner}}((puk_{node}, prk_{node}))$

3 $\sigma_{tk_{node}} \xleftarrow{\$}$ OpenPGP.AttSig.$\mathsf{Sign}_{sik_{owner}}(tk_{node})$

4 $c_{\sigma_{tk_{node}}} \xleftarrow{\$}$ OpenPGP.PKE.$\mathsf{Enc}_{puk_{node}}(\sigma_{tk_{node}})$

5 $c_{\sigma_{comprXAttr}} \xleftarrow{\$}$ ProtonDrive.XAttr.$\mathsf{SC}_{puk_{parent},sik_{owner}}(xAttr)$

6 **return** $(c_{sk_{name}}, c_{\sigma \| name}, \tau_{name}), (c_{sk}, \sigma_{pp}, c_{pp}, l_{(puk_{node},prk_{node})}), c_{\sigma_{tk_{node}}}, c_{\sigma_{comprXAttr}}$

Algorithm 23: $\text{ProtonDrive.LinkAndNode.USC}_{prk_{parent}, vk_{owner}, hk_{parent}}($
$encNameKey, signcryptName, hmacName,$
$encPassphraseKey, signPassphrase,$
$encPassphrase, lockNodeKey,$
$enHmacKey, signcryptComprXAttr)$

Input: $prk_{parent} \in \text{OpenPGP.PKE}.\mathcal{PrK}$
$vk_{owner} \in \text{OpenPGP.DS}.\mathcal{PuK}$
$hk_{parent} \in \text{OpenPGP.HMAC}.\mathcal{K}$
$c_{sk_{name}} \in \text{OpenPGP.SE}.\mathcal{C}$
$c_{\sigma \| name} \in \text{OpenPGP.SE}.\mathcal{C}$
$\tau_{name} \in \text{OpenPGP.HMAC}.\mathcal{H}$
$c_{sk} \in \text{OpenPGP.PKE}.\mathcal{C}$
$\sigma_{pp} \in \text{OpenPGP.DS}.\mathcal{S}$
$c_{pp} \in \text{OpenPGP.SE}.\mathcal{C}$
$l_{(puk_{node}, prk_{node})} \in \text{OpenPGP.SE}.\mathcal{C}$
$c_{\sigma_{tk_{node}}} \in \text{OpenPGP.PKE}.\mathcal{C}$
$c_{\sigma_{comprXAttr}} \in \text{OpenPGP.PKE}.\mathcal{C}$

Output: $name \in \texttt{String}$
$xAttr \in \texttt{String}$
$(puk_{node}, prk_{node}) \in \text{OpenPGP.PKE}.\mathcal{K}$
$tk_{node} \in \text{OpenPGP.HMAC}.\mathcal{K}$

1 $name \leftarrow \text{ProtonDrive.Name.USCVfy}_{prk_{parent}, vk_{owner}, hk_{parent}}(c_{sk_{name}}, c_{\sigma \| name})$
2 $(puk_{node}, prk_{node}) \leftarrow \text{ProtonDrive.HybSig.USC}_{prk_{parent}, vk_{owner}}(c_{sk}, \sigma_{pp}, c_{pp}, l_{(puk_{node}, prk_{node})})$
3 $\sigma_{tk_{node}} \leftarrow \text{OpenPGP.PKE.Dec}_{prk_{node}}(c_{\sigma_{tk_{node}}})$
4 $tk_{node} \leftarrow \text{OpenPGP.AttSig.Vfy}_{vk_{owner}}(\sigma_{tk_{node}})$
5 $xAttr \leftarrow \text{ProtonDrive.XAttr.USC}_{prk_{parent}, vk_{owner}}(c_{\sigma_{comprXAttr}})$
6 **if** $name = \perp$ or $xAttr = \perp$ or $nodeKey = \perp$ or $hmacKey = \perp$ **then**
7 | **return** $\perp$
8 **return** $name, xAttr, (puk_{node}, prk_{node}), tk_{node}$

For each block, three new values are generated and stored: the first one is a symmetric encryption of that block, the second one is the hash of that encryption, and the third is an asymmetrically encrypted signature over the plaintext block.

We start by defining a helper function in Algorithm 24 for file splitting.

Algorithm 24: ProtonDrive.SplitFile(f)

Input: $f \in \texttt{Bytes}^*$
Output: $f^0, ..., f^n \in \texttt{Bytes}^{\leq \ell}$ `//` ℓ `is the block length of OpenPGP.SE, 4MiB`

$\ell \leftarrow$ OpenPGP.SE.*blockLength*
$n = \left\lceil \frac{|f|-1}{\ell} \right\rceil$
for i from 0 to $n-1$ **do**
  $f^i \leftarrow f[i \cdot \ell : (i+1) \cdot \ell]$
$f^n \leftarrow f[n \cdot \ell :]$
return $(f^0, ..., f^n)$

We define ProtonDrive.File, the signcryption scheme used for file encryption and authentication, composed of the four algorithms SGen, RGen, SC, and USC. The sender key generation SGen returns the DS key pair of the address of user creating or modifying that file, and the receiver key generation RGen the PKE key pair of the file node. The output of SC is composed of several components, namely the encrypted session key, encrypted file blocks, encrypted signatures on the file blocks, digests of each encrypted file block and a signature on the concatenation of all those digests, which is called the signed manifest. This is described in Algorithm 25. The actual implementations of the USC algorithm in Proton Drive return two results, the verification result on one side, and the unsigncrypted message on the other. The reason for this is that if a file does not contain signatures or has signatures which are invalid, which might happen for older files, then the user might still want to have access to the contents of the file, and simply be warned that something went wrong during its unsigncryption. We model this as returning $\perp$ when the verification result is $\perp$ and the unsigncrypted message otherwise. USC is described in Algorithm 26. Moreover, we give an illustration of the signcryption algorithm in Fig. 3.8.

Revision

A revision corresponds to a state of a file. It holds the file data and the information that protects the integrity of the file, as well as its extended attributes, which contain information such as the size of the unencrypted

Algorithm 25: $\text{ProtonDrive.File.SC}_{puk_{node}, sik_{writer}}(f)$

Input: $puk_{node} \in \text{OpenPGP.PKE}.\mathcal{PuK}$
$sik_{writer} \in \text{OpenPGP.DS}.\mathcal{K}$
$f \in \texttt{Bytes}^*$ with $n = \left\lceil \frac{|f|-1}{\text{OpenPGP.SE}.blockLength} \right\rceil$

Output: $c_{sk} \in \text{OpenPGP.PKE}.\mathcal{C}$
$c^i \in \text{OpenPGP.SE}.\mathcal{C}$ for $0 \leq i \leq n$
$c^i_\sigma \in \text{OpenPGP.PKE}.\mathcal{C}$ for $0 \leq i \leq n$
$h^i \in \text{OpenPGP.Hash}.\mathcal{H}$ for $0 \leq i \leq n$
$\sigma_{manifest} \in \text{OpenPGP.DS}.\mathcal{S}$

$sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
$c_{sk} \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk_{node}}(sk)$
$(f^0, ..., f^n) \leftarrow \text{ProtonDrive.SplitFile}(f)$
for i from 0 to n **do**
  $c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(f^i)$
  $h^i \leftarrow \text{OpenPGP.Hash.h}(c^i)$
  $\sigma^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik_{writer}}(f^i)$
  $c^i_\sigma \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk_{node}}(\sigma^i)$
$\sigma_{manifest} \leftarrow \text{OpenPGP.DS.Sign}_{sik_{writer}}(h^0 \| ... \| h^n)$
return $(c_{sk}, (c^0, ..., c^n), (c^0_\sigma, ..., c^n_\sigma), (h^0, ..., h^n), \sigma_{manifest})$

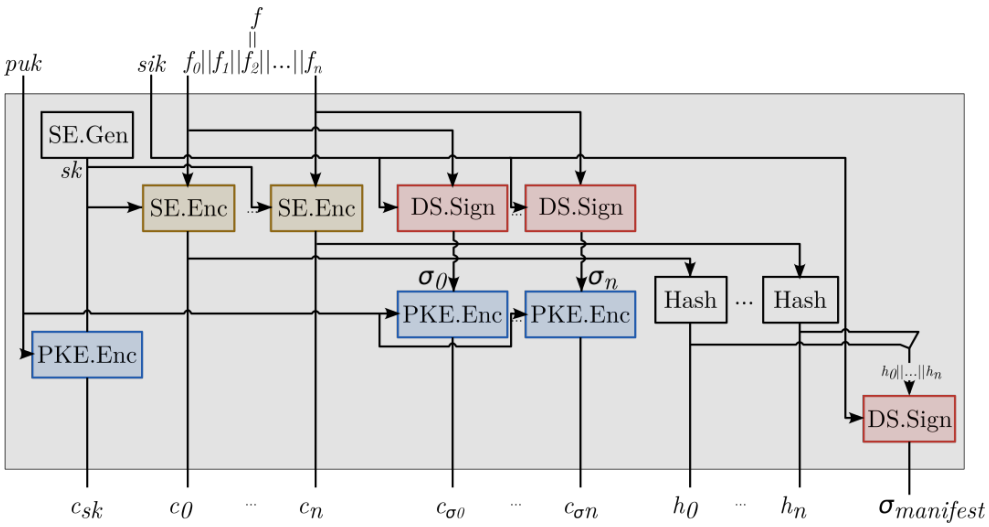

Figure 3.8: File data encryption.

Algorithm 26: ProtonDrive.File.USC$_{prk_{node}, vk_{writer}}(c_{sk}, (c^0, ..., c^n), (c^0_\sigma, ..., c^n_\sigma), (h^0, ..., h^n), \sigma_{manifest})$

Input: $prk_{node} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$vk_{writer} \in$ OpenPGP.DS.$\mathcal{K}$
$c_{sk} \in$ OpenPGP.SE.$\mathcal{K}$
$c^i \in$ OpenPGP.SE.$\mathcal{C}$ for $0 \leq i \leq n$
$c^i_\sigma \in$ OpenPGP.PKE.$\mathcal{C}$ for $0 \leq i \leq n$
$h^i \in$ OpenPGP.Hash.$\mathcal{H}$ for $0 \leq i \leq n$
$\sigma_{manifest} \in$ OpenPGP.DS.$\mathcal{S}$
Output: $f \in \texttt{Bytes}^*$

$valid \leftarrow \top$
$sk \leftarrow$ OpenPGP.PKE.Dec$_{prk_{node}}(c_{sk})$
if $sk = \perp$ **then**
  $valid \leftarrow \perp$
for i from 0 to n **do**
  $f^i \leftarrow$ OpenPGP.SE.Dec$_{sk}(c^i)$
  if $f^i = \perp$ **then**
    $valid \leftarrow \perp$
  $\sigma^i \leftarrow$ OpenPGP.PKE.Dec$_{prk_{node}}(c^i_\sigma)$
  if $\sigma^i = \perp$ **then**
    $valid \leftarrow \perp$
  if OpenPGP.DS.Vfy$_{vk_{writer}}(f^i, \sigma^i) = \perp$ **then**
    $valid \leftarrow \perp$
  if OpenPGP.Hash.h$(c^i) \neq h^i$ **then**
    $valid \leftarrow \perp$
if OpenPGP.DS.Vfy$_{vk_{writer}}(h^0 \| ... \| h^n, \sigma_{manifest}) = \perp$ **then**
  $valid \leftarrow \perp$
if $valid = \perp$ **then**
  return $\perp$
$f \leftarrow f^0 \| ... \| f^n$
return f

file and the last modification dates. Modifications to a file can be made by creating a new revision; keeping multiple allows to have a versioning on the file, and roll back if needed. Each file has at least one revision, for its last valid state, and at most one draft, which is the temporary object created during the modification of a file which is transformed into a revision when uploaded definitively.

We call ProtonDrive.Revision the signcryption scheme which is used to create and retrieve data from a revision. The key generation algorithm returns the file node key as the PKE key pair and the address signature key pair of the address writing to the file as the DS key pair. We define the encryption algorithm in Algorithm 27 and the decryption algorithm in Algorithm 28.

Algorithm 27: $\text{ProtonDrive.Revision.SC}_{puk_{node}, sik_{writer}}(f, xAttr)$

Input: $puk_{node} \in \text{OpenPGP.PKE}.\mathcal{PuK}$
$sik_{writer} \in \text{OpenPGP.DS}.\mathcal{K}$
$f \in \texttt{Bytes}^*$ with $n = \left\lceil \frac{|f|-1}{\text{OpenPGP.SE}.blockLength} \right\rceil$
$xAttr \in \texttt{String}$

Output: $c_{sk} \in \text{OpenPGP.PKE}.\mathcal{C}$
$c^i \in \text{OpenPGP.SE}.\mathcal{C}$ for $0 \leq i \leq n$
$c^i_\sigma \in \text{OpenPGP.PKE}.\mathcal{C}$ for $0 \leq i \leq n$
$h^i \in \text{OpenPGP.Hash}.\mathcal{H}$ for $0 \leq i \leq n$
$\sigma_{manifest} \in \text{OpenPGP.DS}.\mathcal{S}$
$c_{\sigma_{comprXAttr}} \in \text{OpenPGP.PKE}.\mathcal{C}$

1 $(c_{sk}, (c^0, ..., c^n), (c^0_\sigma, ..., c^n_\sigma), (h^0, ..., h^n), \sigma_{manifest}) \xleftarrow{\$} \text{ProtonDrive.File.SC}_{puk_{node}, sik_{writer}}(f)$

$c_{\sigma_{comprXAttr}} \xleftarrow{\$} \text{ProtonDrive.XAttr}_{pk_{node}, sk_{writer}}(xAttr)$

return $(c_{sk}, (c^0, ..., c^n), (c^0_\sigma, ..., c^n_\sigma), (h^0, ..., h^n), \sigma_{manifest}), c_{\sigma_{comprXAttr}}$

3.2.5 Photos

The photo feature consists of a parallel file tree with its own root share and root folder, which is entirely dedicated to storing pictures. The files that are stored in this tree can only be pictures, and may store more information in their xAttr than a regular file, such as the time and position at which the picture was taken. Moreover, there is no check on duplicates for folder names in the Photo partition.

Algorithm 28: ProtonDrive.Revision.USC$_{prk_{node}, vk_{writer}}(c_{sk}, (c_0, ..., c_n), (c_\sigma^0, ..., c_\sigma^n), (h_0, ..., h_n), \sigma_{manifest}, c_{\sigma_{comprXAttr}})$

Input: $prk_{node} \in$ OpenPGP.PKE.$\mathcal{PuK}$
$vk_{writer} \in$ OpenPGP.DS.$\mathcal{K}$
$c_{sk} \in$ OpenPGP.SE.$\mathcal{K}$
$c^i \in$ OpenPGP.SE.$\mathcal{C}$ for $0 \leq i \leq n$
$c_\sigma^i \in$ OpenPGP.PKE.$\mathcal{C}$ for $0 \leq i \leq n$
$h^i \in$ OpenPGP.Hash.$\mathcal{H}$ for $0 \leq i \leq n$
$\sigma_{manifest} \in$ OpenPGP.DS.$\mathcal{S}$
$c_{\sigma_{comprXAttr}} \in$ OpenPGP.PKE.$\mathcal{C}$
Output: $f \in$ `Bytes`*
$xAttr \in$ `String`

1 $f \leftarrow$ ProtonDrive.File.USC$_{prk_{node}, vk_{writer}}(c_{sk}, (c^0, ..., c^n), (c_\sigma^0, ..., c_\sigma^n), (h^0, ..., h^n), \sigma_{manifest})$

$xAttr \leftarrow$ ProtonDrive.XAttr.USC$_{prk_{node}, vk_{writer}}(c_{\sigma_{comprXAttr}})$

if $f = \perp$ or $xAttr = \perp$ **then**
  return $\perp$

return $f, xAttr$

3.3 Threat model

We consider two threat models in our analysis. The first one, which we call Client-to-Server communication, corresponds to the case where the user and server are both honest, and a powerful man-in-the middle (MitM) attacker able to bypass any protection on the communication channel can read and write on the connection between the two. The second one is when one or more users are communicating and the server is considered to be potentially adversarial or compromised. We refer to this as End-to-End (E2E) communication. We make this distinction to be able to express differences in capabilities between a malicious server and other attackers.

3.3.1 Assumptions

We operate under certain assumptions, without which no security guarantee can be given.

- The client application is the application provided by Proton. More precisely, an honest user runs one of the official Proton Drive clients, and is not using a client implementation provided by an attacker. This is important because if a user can be tricked into using a malicious client,

Client-to-Server	End-to-End
The adversary can observe any traffic between the client and the server.	The adversary has access to everything that the server stores.
The adversary can send messages to any client and to the server.	The server can send messages to any client.
The adversary can modify messages between the client and server.	The server can modify anything it stores.
The adversary can drop messages between the client and server.	The server can ignore client messages.

Figure 3.9: The capabilities of an adversary in a Client-to-Server setting and in an End-to-End setting.

then an attacker can trivially retrieve its credentials, and therefore gain complete access to the volume of that user.

- When queried for the public key corresponding to a given user email address, the server returns the correct one. Ensuring that the right key is accessible is the objective of Key Transparency, a project which aims to provide integrity over the public keys of addresses that are kept by the server, and we therefore consider it to be out of scope for our considerations. This closes off the scope of key overwriting attacks on keys linked to addresses, and consequently any trivial attack where the key server simply replaces public keys with its own. Note that with sharing between users not being implemented, this only has a very limited impact on our considerations, because a user does not need to query the key server to retrieve its own keys.

We list the capacities of the adversary in the Client-to-Server and End-to-End settings in Fig. 3.9.

3.3.2 Security Claims

In their blog post on the Proton Drive threat model [52], Proton states that they ”[...] have no way of reading your data or using it to build a profile on you for advertising purposes [...]”, and that ”[...] if an attacker gains access to data flows between your device and our servers, they will not be able to decrypt user files”, meaning that neither Proton themselves (in an End-to-End setting) nor eavesdroppers (in a Client-to-Server setting) can get information on the contents of user files. This is best modeled as indistinguishability un-

der chosen ciphertext attack, because attackers in both models are able to modify or add ciphertexts. This is also generally recognized to be a desirable security property for E2EE cloud storage services, where the server has access to all of the encrypted data of the users.

On authenticity protection, Proton states on its web page [51] that "Proton Drive uses cryptographic signatures to verify the authenticity of your files and folders, meaning that you'll immediately detect any efforts to tamper with your files." For an adversary to not be able to modify the files of a user without it being detectable, we need the file authentication mechanism to provide at least weak unforgeability, meaning that it is not possible for an adversary to create a valid ciphertext corresponding to a plaintext that has never been stored before. If that property is not provided, it can lead to framing attacks, where the attacker puts compromising content in its target storage. We therefore consider that the file encryption aims to be protected with WUF-CMA tags or signatures (defined in Fig. 2.10 and Section 2.2.4).

Additionally, we consider that no entity can have access to keys which do not belong to it. In a setting where we are dealing with both signature and encryption keys, and where the cryptographic schemes we consider are signcryption schemes (as presented in Section 2.2.5), this implies outsider security notions, specifically OUT-IND-CCA (defined in Fig. 2.14) for confidentiality and OUT-WUF-CMA (defined in Fig. 2.15) for authenticity.

We give the exact games corresponding to the expected security properties for the objects we want them to apply to, and evaluate whether the properties are satisfied in Chapter 4. The properties we expect are summarized in Fig. 3.10.

We only ask for outsider security on signcryption schemes because the keys getting accessed by non-authorized entities is considered as a breach of the system. However, it is necessary to point out that this might need to be reconsidered once sharing between Proton users user will be implemented. Indeed, in that context, it is most probable that the setting would be such that several users sharing writing rights on a file or folder will all have access to the same encryption (or receiver) key, while each having to use their address signature key as sender key. This would then require insider integrity, where the adversary has the same goal as in OUT-WUF-CMA, but with the additional knowledge of the encryption key.

Additionally, note that the confidentiality of the structure of the file system is not protected, and that its integrity is only mostly ensured. The weakness in integrity we are referring to is the lack of binding between a file or folder and its name, which would allow shuffling of names among the children of one same parent folder.

Client-Server	End-to-End	Property
No external eavesdropper can learn any information about the data stored or the keys from the data it can observe.	The server cannot learn any information about the data stored or the keys.	OUT-IND-CCA
No MITM which has not been granted writing permission to a file or folder by its owner can modify or add a file or folder to it in such a way that the added object decrypts without errors.	The server cannot modify or add any object in the volume of any user in such a way that the added object decrypts without errors.	OUT-WUF-CMA

Figure 3.10: Summary of the security expected from the cryptographic schemes used in Proton Drive, and the corresponding game-based security notions.

Chapter 4

Security Proofs

In this chapter, we assess the security of the file encryption algorithm used for the Proton Drive protocol. We focus on the confidentiality and authenticity provided by ProtonDrive.File, a signcryption scheme built entirely upon the OpenPGP library.

4.1 File Encryption

There are two properties that we wish for from file encryption: confidentiality and authenticity in an outsider security setting. Specifically, given the threat model which we discussed in Section 3.3, we identify OUT-IND-CCA and OUT-WUF-CMA as the security goals to aim for in the compromised or malicious provider threat model.

We attempt to split the proof for OUT-IND-CCA security into confidentiality and authenticity, following a strategy similar to that used for symmetric encryption schemes, where AE security implies IND-CCA security. We start by successfully proving that ProtonDrive.File is OUT-IND-CPA, provided that both OpenPGP.SE and OpenPGP.PKE are IND-CPA, in Section 4.1.1. But then, ProtonDrive.File turns out to fail OUT-SUF-CMA security because of an attack which we present in Section 4.1.2. This immediately implies that it is also not OUT-IND-CCA, as explained in Section 4.1.3, and makes showing that the combination of OUT-IND-CPA and OUT-SUF-CMA implies OUT-IND-CCA for signcryption schemes out of scope for this thesis. To evaluate the level of authenticity protection provided by the scheme, we finally prove that ProtonDrive.File still satisfies OUT-WUF-CMA security in Section 4.1.4.

4.1.1 Confidentiality (Chosen Plaintext Attack)

As we show later on, ProtonDrive.File is not OUT-IND-CCA. We do however give a proof of a weaker security notion, OUT-IND-CPA (defined in Fig. 2.13).

Theorem 4.1 (OUT-IND-CPA of ProtonDrive.File) *Let* ProtonDrive.*File be the signcryption scheme used for signcryption of Proton Drive files as presented in Algorithm 25 and Algorithm 26. Let $\mathcal{A}$ be an adversary against the* OUT-IND-CPA *security of* ProtonDrive.*File making at most q queries to its oracle LoR, each of which has a message length of m, which is split into n blocks by* ProtonDrive.*SplitFile. Then there exist adversaries $\mathcal{B}_{PKE}$ and $\mathcal{B}_{SE}$ such that*

$$\begin{aligned}\mathcal{A}dv^{\text{OUT-IND-CPA}}_{\text{ProtonDrive.}File}(\mathcal{A}) \leq 2 \cdot \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.}PKE}(\mathcal{B}_{PKE}) \\ + \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.}SE}(\mathcal{B}_{SE}).\end{aligned} \tag{4.1}$$

Adversary $\mathcal{B}_{PKE}$ makes at most $(n+1) \cdot q$ queries to its oracle LoR, and the adversary $\mathcal{B}_{SE}$ makes at most $n \cdot q$ queries to its oracle LoR.

Proof (Theorem 4.1) We proceed by doing a three-hop game-hopping proof. To this end, we define four games as follows:

G_0 OUT-IND-CPA$_{\text{ProtonDrive.File}}$ with b set to 0.

G_1 The game G_0 with two modifications. First, the encrypted session key c_{sk} is the encryption of a new independent session key sk'. Second is that the encrypted signatures c^i_σ are computed on the second file f_1 instead of being computed on f_0. These two changes are applied to the parts of ProtonDrive.File pertaining to public-key encryption.

G_2 The modification of game G_1 such that the symmetrically encrypted file blocks are computed on the second file f_1 instead of f_0. This also has the consequence of changing the values of the ciphertext block digests h^i to the hashes of the ciphertext blocks computed on f_1 and the signed manifest $\sigma_{manifest}$ to the signature on the concatenation of these new hash blocks.

G_3 The modification of game G_2 which restores c_{sk} to be the encryption of the session key used for symmetric encryption rather than an independent one. This game is equivalent to OUT-IND-CPA$_{\text{ProtonDrive.File}}$ with b set to 1.

For each of the three pairs of consecutive games, we show that an adversary successfully distinguishing between the two games can also be used to construct a successful adversary against either IND-CPA$_{\text{ProtonDrive.PKE}}$ or IND-CPA$_{\text{ProtonDrive.SE}}$, and that the advantage of the distinguishers is

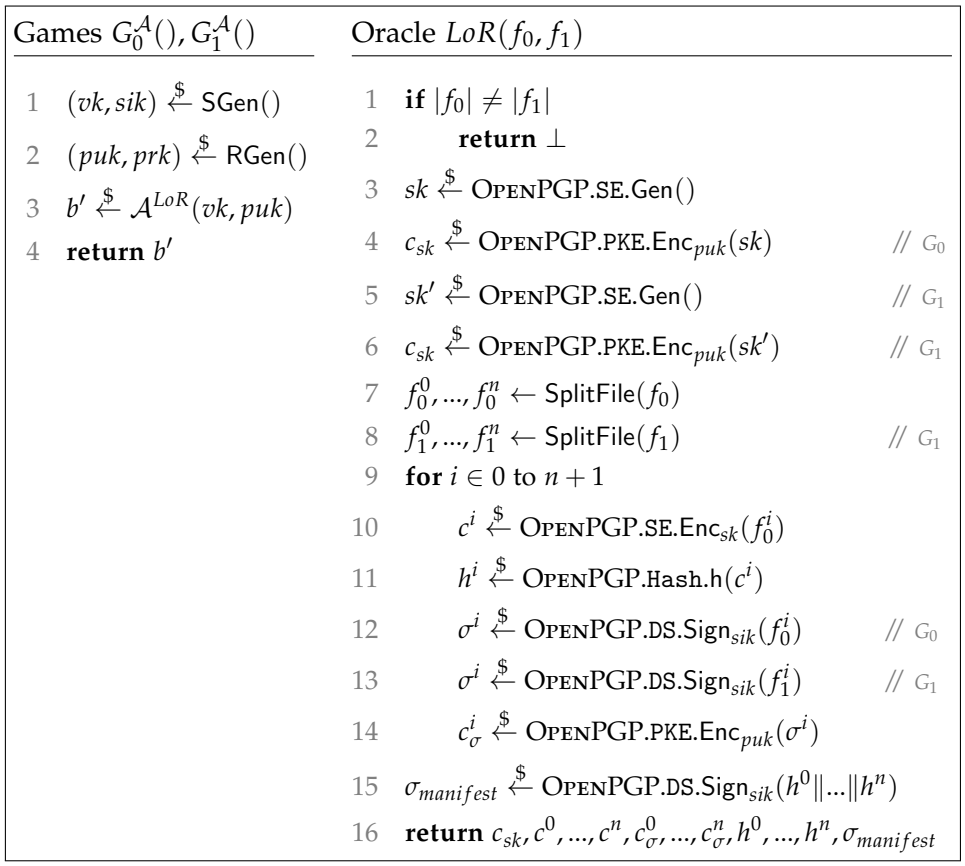

Figure 4.1: The first two games used for the confidentiality proofs.

therefore bounded by the sum of the advantages of the constructed adversaries playing against the IND-CPA game on the underlying cryptographic schemes.

We start by giving the two games involved in the first hop in Fig. 4.1

Given an adversary $\mathcal{A}$ which can distinguish between these two games, G_0 and G_1, we build an adversary $\mathcal{B}_{\text{PKE},1}$ against IND-CPA$_{\text{PKE}}$ in Fig. 4.2. That new adversary generates the sender keys of the signcryption scheme. Then, it calls the adversary $\mathcal{A}$ with the simulated oracle LoR', which replicates the behavior or the LoR oracles of game G_0 and G_1, and calls the LoR oracle of IND-CPA$_{\text{PKE}}$ in the places where they differ, such that the output of the oracle is the same as that of G_0 when $b = 0$ and the same as that of G_1 when $b = 1$. The full details are given in Fig. 4.2.

Adversary $\mathcal{B}_{\text{PKE},1}$ perfectly simulates game G_0 if $b = 0$ and G_1 if $b = 1$ for the LoR oracle of IND-CPA$_{\text{PKE}}$. Furthermore, since $\mathcal{B}_{\text{PKE},1}$ halts and returns the same output as $\mathcal{A}$, we have that its advantage against game IND-CPA$_{\text{PKE}}$ is at least as good as the advantage of $\mathcal{A}$ distinguishing G_0 from G_1. Eq. (4.2)

Adversary $\mathcal{B}_{\text{PKE},1}(puk)$

1 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
2 $b' \xleftarrow{\$} \mathcal{A}^{LoR'}(vk, puk)$
3 **return** b'

Simulated oracle $LoR'(f_0, f_1)$

1 **if** $|f_0| \neq |f_1|$
2 **return** $\perp$
3 $sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
4 $sk' \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
5 $c_{sk} \leftarrow LoR(sk, sk')$
6 $f_0^0, ..., f_0^n \leftarrow \mathsf{SplitFile}(f_0)$
7 $f_1^0, ..., f_1^n \leftarrow \mathsf{SplitFile}(f_1)$
8 **for** $i \in 0$ to $n+1$
9 $c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(f_0^i)$
10 $h^i \xleftarrow{\$} \text{OpenPGP.Hash.h}(c^i)$
11 $\sigma_0^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(f_0^i)$
12 $\sigma_1^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(f_1^i)$
13 $c_\sigma^i \xleftarrow{\$} LoR(\sigma_0^i, \sigma_1^i)$
14 $\sigma_{manifest} \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(h^0 \| ... \| h^n)$
15 **return** $c_{sk}, c^0, ..., c^n, c_\sigma^0, ..., c_\sigma^n, h^0, ..., h^n, \sigma_{manifest}$

Figure 4.2: The simulated adversary against IND-CPA$_{\text{PKE}}$ using adversary $\mathcal{A}$.

follows from this.

$$Pr[G_0(\mathcal{A}) \Rightarrow 1] - Pr[G_1(\mathcal{A}) \Rightarrow 1] \leq \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.PKE}}(\mathcal{B}_{\text{PKE},1}). \tag{4.2}$$

We continue onto our second hop. The games G_1 and G_2 are given in Fig. 4.3.

Given an adversary $\mathcal{A}$ which can distinguish between games G_1 and G_2, we build an adversary $\mathcal{B}_{\text{SE}}$ against IND-CPA$_{\text{SE}}$. That new adversary generates the sender and receiver keys of the signcryption scheme. It then calls $\mathcal{A}$ with the simulated oracle LoR', which reproduces the behavior of the LoR oracles of games G_1 and G_2. For the part where the aforementioned oracles differ, LoR' calls the LoR oracle of IND-CPA$_{\text{SE}}$, such that the output of the oracle is the same as that of G_1 when $b = 0$ and the same as that of G_2 when $b = 1$. The full details are given in Fig. 4.4. Note that the first change previously made from G_0 to G_1 (on line 6 of Fig. 4.1) allows us to generate a value for c_{sk} in the simulated oracle LoR' even though the symmetric encryption key is not known.

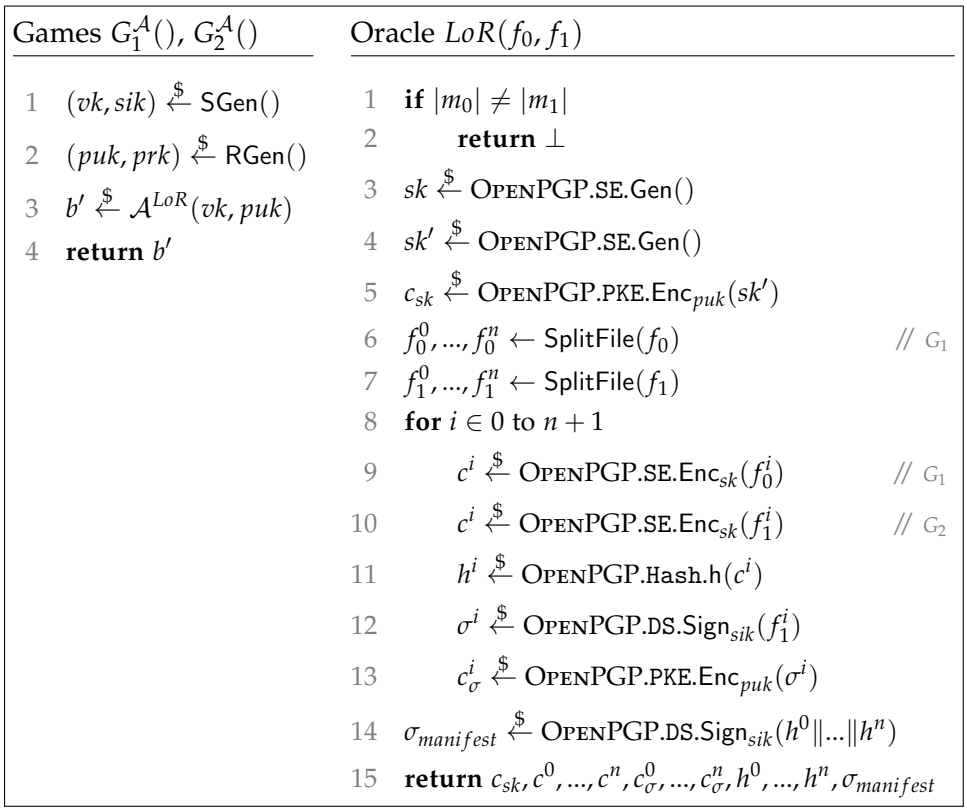

Figure 4.3: Games G_1 and G_2 of the confidentiality proof on PROTONDRIVE.File.

Adversary $\mathcal{B}_{\text{SE}}$ perfectly simulates game G_1 if $b = 0$ and G_2 if $b = 1$ for the LoR oracle of IND-CPA$_{\text{SE}}$. Furthermore, since $\mathcal{B}_{\text{SE}}$ halts and returns the same output as $\mathcal{A}$, we have that its advantage against game IND-CPA$_{\text{SE}}$ is at least as good as the advantage of $\mathcal{A}$ distinguishing G_1 from G_2. Eq. (4.3) follows from this.

$$Pr[G_1(\mathcal{A}) \Rightarrow 1] - Pr[G_2(\mathcal{A}) \Rightarrow 1] \leq \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.SE}}(\mathcal{B}_{\text{SE}}). \tag{4.3}$$

Our last hop is between games G_2 and G_3, which we give in Fig. 4.5.

If the adversary $\mathcal{A}$ is able to distinguish between games G_2 and G_3, we can build an adversary $\mathcal{B}_{\text{PKE},2}$ against IND-CPA$_{\text{PKE}}$. $\mathcal{B}_{\text{PKE},2}$ starts by generating the sender keys for the signcryption scheme. It then calls the adversary $\mathcal{A}$ with the simulated oracle LoR', which recreates the behavior of the LoR oracle from games G_2 and G_3, and uses the LoR oracle from IND-CPA$_{\text{PKE}}$ on places where the aforementioned oracles differ. This implies that the output of the simulated oracle is the same as that of G_2 when $b = 0$ and the same as that of G_3 when $b = 1$. We give the full details of this reduction in Fig. 4.6.

Adversary $\mathcal{B}_{\text{PKE},2}$ perfectly simulates game G_2 if $b = 0$ and G_3 if $b = 1$ for the LoR oracle of IND-CPA$_{\text{PKE}}$. Furthermore, since $\mathcal{B}_{\text{PKE},2}$ halts and returns the

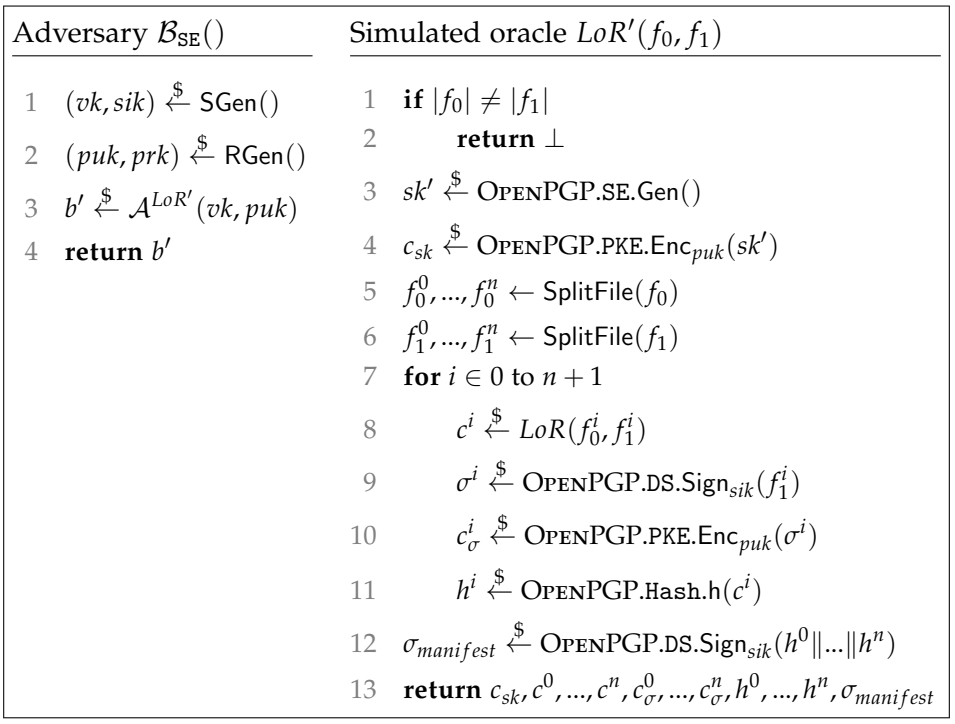

Figure 4.4: The simulated adversary against IND-CPA$_{\text{SE}}$ using adversary $\mathcal{A}$.

same output as $\mathcal{A}$, we have that its advantage against game IND-CPA$_{\text{PKE}}$ is at least as good as the advantage of $\mathcal{A}$ distinguishing G_2 from G_3. Eq. (4.4) follows from this.

$$Pr[G_2(\mathcal{A}) \Rightarrow 1] - Pr[G_3(\mathcal{A}) \Rightarrow 1] \leq \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.PKE}}(\mathcal{B}_{\text{PKE},2}). \tag{4.4}$$

Finally, we can compute a bound on the advantage of the adversary $\mathcal{A}$ playing OUT-IND-CPA$_{\text{ProtonDrive.File}}$. To this end, we state and prove a simple lemma about the addition of the advantages of two adversaries playing the same game.

Lemma 4.2 *Given an indistinguishability game G, and two adversaries $\mathcal{A}_0$ and $\mathcal{A}_1$ playing that game, then there exists an adversary $\mathcal{A}$ such that*

$$\mathcal{A}dv^G(\mathcal{A}_0) + \mathcal{A}dv^G(\mathcal{A}_1) \leq 2 \cdot \mathcal{A}dv^G(\mathcal{A}).$$

Proof (Lemma 4.2) To show this, we show how to construct such an adversary $\mathcal{A}$, and prove that it satisfies our bound. We define adversary $\mathcal{A}$ as picking either $\mathcal{A}_0$ or $\mathcal{A}_1$ uniformly as random, and returning the value that the adversary it chose returns.

Games $G_2^{\mathcal{A}}()$, $G_3^{\mathcal{A}}()$

1 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
2 $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
3 $b' \xleftarrow{\$} \mathcal{A}^{LoR}(vk, puk)$
4 **return** b'

Oracle $LoR(f_0, f_1)$

1 **if** $|f_0| \neq |f_1|$
2 **return** $\perp$
3 $sk \xleftarrow{\$} \textsc{OpenPGP.SE.Gen}()$
4 $sk' \xleftarrow{\$} \textsc{OpenPGP.SE.Gen}()$ // G_2
5 $c_{sk} \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(sk')$ // G_2
6 $c_{sk} \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(sk)$ // G_3
7 $f_1^0, ..., f_1^n \leftarrow \mathsf{SplitFile}(f_1)$
8 **for** $i \in 0$ to $n+1$
9 $c^i \xleftarrow{\$} \textsc{OpenPGP.SE.Enc}_{sk}(f_1^i)$
10 $h^i \xleftarrow{\$} \textsc{OpenPGP.Hash.h}(c^i)$
11 $\sigma^i \xleftarrow{\$} \textsc{OpenPGP.DS.Sign}_{sik}(f_1^i)$
12 $c_\sigma^i \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
13 $\sigma_{manifest} \xleftarrow{\$} \textsc{OpenPGP.DS.Sign}_{sik}(h^0 \| ... \| h^n)$
14 **return** $c_{sk}, c^0, ..., c^n, c_\sigma^0, ..., c_\sigma^n, h^0, ..., h^n, \sigma_{manifest}$

Figure 4.5: Games G_2 and G_3 of the confidentiality proof for the scheme ProtonDrive.File.

Adversary $\mathcal{A}()$

1 $b \xleftarrow{\$} \{0, 1\}$
2 $b' \xleftarrow{\$} \mathcal{A}_b()$
3 **return** b'

Then, we have

$$\begin{aligned} 2 \cdot \mathcal{A}dv^G(\mathcal{A}) &= 4 \cdot (Pr[G(\mathcal{A})] - 1/2) \\ &= 4 \cdot (1/2 \cdot Pr[G(\mathcal{A}_0)] + 1/2 \cdot Pr[G(\mathcal{A}_1)] - 1/2) \\ &= 2 \cdot (Pr[G(\mathcal{A}_0)] - 1/2) + 2 \cdot (Pr[G(\mathcal{A}_1)] - 1/2) \\ &= \mathcal{A}dv^G(\mathcal{A}_0) + \mathcal{A}dv^G(\mathcal{A}_1). \end{aligned}$$

We have therefore successfully constructed an adversary satisfying our condition. □

We can now show the bound on $\mathcal{A}dv^{\text{OUT-IND-CPA}}_{\text{ProtonDrive.File}}(\mathcal{A})$. For legibility, we call P_i the probability that $G_i(\mathcal{A})$ returns 1.

From the definition of the advantage of OUT-IND-CPA$_{\text{ProtonDrive.File}}$ with the advantage rewriting lemma, we can extend the bound to Eq. (4.5) using

Adversary $\mathcal{B}_{\text{PKE}}(puk)$	Simulated oracle $LoR'(f_0, f_1)$
1 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$	1 **if** $\lvert f_0\rvert \neq \lvert f_1\rvert$
2 $b' \xleftarrow{\$} \mathcal{A}^{LoR'}(vk, puk)$	2 **return** $\perp$
3 **return** $b = b'$	3 $sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
	4 $sk' \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
	5 $c_{sk} \xleftarrow{\$} LoR(sk', sk)$
	6 $f_0^0, ..., f_0^n \leftarrow \mathsf{SplitFile}(f_0)$
	7 $f_1^0, ..., f_1^n \leftarrow \mathsf{SplitFile}(f_1)$
	8 **for** $i \in 0$ to $n+1$
	9 $c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(f_1^i)$
	10 $h^i \xleftarrow{\$} \text{OpenPGP.Hash.h}(c^i)$
	11 $\sigma^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(f_1^i)$
	12 $c_\sigma^i \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
	13 $\sigma_{manifest} \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(h^0 \Vert ... \Vert h^n)$
	14 **return** $c_{sk}, c^0, ..., c^n, c_\sigma^0, ..., c_\sigma^n, h^0, ..., h^n, \sigma_{manifest}$

Figure 4.6: The simulated adversary against IND-CPA$_{\text{PKE}}$ using adversary $\mathcal{A}$.

a telescoping sum. We can further upper bound this by Eq. (4.6) using the bounds Eqs. (4.2) to (4.4) that we found thanks to our reductions Figs. 4.2, 4.4 and 4.6.

$$\begin{aligned}
\mathcal{A}dv^{\text{OUT-IND-CPA}}_{\text{ProtonDrive.File}}(\mathcal{A}) &= Pr[b' = 1 | b = 0] - Pr[b' = 1 | b = 1] \\
&= P_0 - P_3 \\
&= (P_0 - P_1) + (P_1 - P_2) + (P_2 - P_3) \qquad (4.5) \\
&\leq \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.PKE}}(\mathcal{B}_{\text{PKE},1}) \\
&\quad + \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.SE}}(\mathcal{B}_{\text{SE}}) \qquad (4.6) \\
&\quad + \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.PKE}}(\mathcal{B}_{\text{PKE},2})
\end{aligned}$$

Finally, we can apply our result from Lemma 4.2 for game IND-CPA$_{\text{PKE}}$ to build adversary $\mathcal{B}_{\text{PKE}}$ from $\mathcal{B}_{\text{PKE},1}$ and $\mathcal{B}_{\text{PKE},2}$. This allows us to simplify Eq. (4.6) to Eq. (4.7).

$$\mathcal{A}dv^{\text{OUT-IND-CPA}}_{\text{ProtonDrive.File}}(\mathcal{A}) \leq 2 \cdot \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.PKE}}(\mathcal{B}_{\text{PKE}}) + \mathcal{A}dv^{\text{IND-CPA}}_{\text{OpenPGP.SE}}(\mathcal{B}_{\text{SE}}) \qquad (4.7)$$

□

Adversary $\mathcal{A}^{sc,usc}(puk)$

1 $f \xleftarrow{\$} \text{ProtonDrive.File}.\mathcal{M}$

2 $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma, \sigma_{manifest}) \xleftarrow{\$} sc(f)$

3 $(c'_{sk}, c^{0\prime}, ..., c^{n\prime}, h^{0\prime}, ..., h^{n\prime}, c^{0\prime}_\sigma, ..., c^{n\prime}_\sigma, \sigma'_{manifest}) \xleftarrow{\$} sc(f)$

4 **return** $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^{0\prime}_\sigma, ..., c^{n\prime}_\sigma, \sigma_{manifest})$

Figure 4.7: An efficient adversary $\mathcal{A}$ for the game OUT-SUF-CMA$_{\text{ProtonDrive.File}}$.

4.1.2 Strong Unforgeability

We first observe that the scheme ProtonDrive.File is not OUT-SUF-CMA secure as we defined in Fig. 2.16. Indeed, we can construct an adversary $\mathcal{A}$ which performs a forgery as follows: let the adversary make two queries to the signcryption oracle, both with the same file. Then the adversary can forge a new valid ciphertext by taking the values for c_{sk}, c^i, h^i and $\sigma_{manifest}$ from the result of the first query it made, and the values for c^i_{sigma} from the second. With high probability (that of the two queries returning different values for at least one of the c^i_{sigma}) this generates a new ciphertext and therefore breaks OUT-SUF-CMA security. We describe that adversary in details in Fig. 4.7.

Let us now quantify the advantage of adversary $\mathcal{A}$ playing against the game OUT-SUF-CMA$_{\text{ProtonDrive.File}}$. We define

$$\mathcal{S}_{f^b} = \{c \in \text{OpenPGP.PKE}.\mathcal{C} | \text{OpenPGP.PKE.Enc}_{puk}(\text{OpenPGP.DS.Sign}_{sik}(f^b)) = c\}$$

to be the set of all possible images of some file block f^b when passed through the composition of the signature algorithm with key sik and the PKE encryption algorithm with key puk. We are specifically interested in the cardinality of this set, and observe that it does not depend on the file block f^b, and that we necessarily have $|\mathcal{S}_{f^b}| > 1$. Indeed, if this was not the case, then encryption using OpenPGP.PKE would be deterministic, which would break IND-CPA$_{\text{PKE}}$. We also note that assuming that the sources of randomness used in OpenPGP.PKE and OpenPGP.DS are uniform, then the probabilities of $c^i_\sigma = c^{i\prime}_\sigma$ for all i's are independent.

By correctness of OpenPGP.SE, OpenPGP.DS, and OpenPGP.PKE, we have that the ciphertext returned by the adversary does not return an error on unsigncryption. This allows us to compute the advantage of $\mathcal{A}$ against

OUT-SUF-CMA$_{\text{ProtonDrive.File}}$ to be

$$\begin{aligned}\mathcal{A}dv^{\text{OUT-SUF-CMA}}_{\text{ProtonDrive.File}}(\mathcal{A}) &= 2 \cdot \Big(Pr[\exists i \mid 0 \leq i \leq n \text{ and } c^i_\sigma \neq c^{i\prime}_\sigma] - 1/2\Big) \\ &= 2 \cdot \Big(1 - Pr[\forall i \mid 0 \leq i \leq n \text{ and } c^i_\sigma = c^{i\prime}_\sigma] - 1/2\Big) \\ &= 2 \cdot \Big(1/2 - Pr[c^i_\sigma = c^{i\prime}_\sigma]^{(n+1)}\Big) \\ &= 1 - 2 \cdot \left(\frac{1}{|S_{f^b}|}\right)^{(n+1)}.\end{aligned}$$

We note that the advantage of the adversary can be made arbitrarily close to 1 by picking a longer input file.

4.1.3 Confidentiality (Chosen Ciphertext Attack)

The previous result is not only interesting, it also leads us to deduce that ProtonDrive.File is not OUT-IND-CCA secure as we defined it in Fig. 2.14. Indeed, we can use the same construction to fool the unsigncryption oracle into giving us the decryption of a ciphertext built from two distinct queries to the *LoR* oracle of OUT-IND-CCA$_{\text{ProtonDrive.File}}$, thus allowing us to find out which chosen plaintext was encrypted. By the same reasoning as before, we would get the same advantage from building such an adversary $\mathcal{A}$

$$\mathcal{A}dv^{\text{OUT-IND-CCA}}_{\text{ProtonDrive.File}}(\mathcal{A}) = 1 - 2 \cdot |S_{f^b}|^{-(n+1)},$$

which can also be made arbitrarily close to 1 by increasing the value of n.

4.1.4 Weak Unforgeability

We proceed by showing that while ProtonDrive.File is not OUT-SUF-CMA, it still achieves OUT-WUF-CMA security (defined in Fig. 2.15).

Theorem 4.3 (OUT-WUF-CMA of ProtonDrive.File) *Let* ProtonDrive.*File be the signcryption scheme used for signcryption of Proton Drive files as stated in Algorithm 25 and Algorithm 26. Let $\mathcal{A}$ be an adversary against the* OUT-WUF-CMA *security of* ProtonDrive.*File making at most q queries to its oracle sc for each of which the queried message m is split into n blocks by* ProtonDrive.*SplitFile. Then there exist adversaries $\mathcal{B}_{SE}$, $\mathcal{B}_{DS,1}$, $\mathcal{B}_{Hash}$, and $\mathcal{B}_{DS,2}$ such that*

$$\begin{aligned}\mathcal{A}dv^{\text{OUT-WUF-CMA}}_{\text{ProtonDrive.}File}(\mathcal{A}) \leq\ & min\{\mathcal{A}dv^{\text{KROB}}_{\text{OpenPGP.}SE}(\mathcal{B}_{SE}), \mathcal{A}dv^{\text{SUF-CMA}}_{\text{OpenPGP.}DS}(\mathcal{B}_{DS,1})\} \\ & + \mathcal{A}dv^{\text{CR}}_{\text{OpenPGP.}Hash}(\mathcal{B}_{Hash}) \\ & + \mathcal{A}dv^{\text{SUF-CMA}}_{\text{OpenPGP.}DS}(\mathcal{B}_{DS,2}).\end{aligned} \tag{4.8}$$

The adversaries $\mathcal{B}_{DS,1}$ and $\mathcal{B}_{DS,1}$ make at most $(n+1) \cdot q$ queries to their signing oracle. $\mathcal{B}_{SE}$ and $\mathcal{B}_{Hash}$ make no queries as they do not have any oracle.

To prove Theorem 4.3, we use the fundamental lemma of game playing. It can also be found under the names of difference lemma or the indistinguishable until bad lemma. We use it as it is presented by Bellare and Rogaway in [11, 12] and adapt the notation to our usage.

Lemma 4.4 (Fundamental lemma of game playing) *Let two games G_0 and G_1 be identical until bad, meaning that G_0 and G_1 are games containing a flag bad, and are identical except for the few steps immediately following the setting of bad to true in G_1. More precisely, we have that G_0 and G_1 are strictly identical games, except when bad is set to true, in which case G_1 executes one or more steps that are not present in G_0. Then, for any adversary $\mathcal{A}$ playing the games G_0 and G_1, we have*

$$Pr[G_0(\mathcal{A})] - Pr[G_1(\mathcal{A})] \leq Pr[bad].$$

Proof (Lemma 4.4) We refer to Bellare and Rogaway [11, Lemma 2] for the proof of this lemma. □

Proof (Theorem 4.3) Once again, we define several games for our proof. Our starting point is the game OUT-WUF-CMA$_{\text{ProtonDrive.File}}$. We unfold the definition of OUT-WUF-CMA from Fig. 2.15 with the implementation of ProtonDrive.File given in Algorithm 25 and Algorithm 26 in Fig. 4.8.

The jump to the next game, G_1, is the most substantial, but is really only a change of form. We introduce new variables to increase the granularity of our apprehension of the capacities of the adversary. The first new group of variables includes three separate booleans, $valid_{m,\sigma}$, $valid_h$, and $valid_{man}$, which replace the single *valid* boolean which we have in G_0. If all the validity checks performed on the ciphertext pass, then all three booleans are set to true, and we therefore have that $valid_{m,\sigma}$ and $valid_h$ and $valid_{man} = valid$ at the end of the game. To accommodate this change, we replace the setting of m to $\perp$ when *valid* is $false$ by returning $\perp$ if any of these three booleans are set to $\perp$.

We also introduce three booleans $forged_{m,\sigma}$, $forged_h$, and $forged_{man}$, each indicating whether the adversary performed a successful forgery in one of the three possible attack paths. We distinguish the notions of validity and forgery on a part of a ciphertext because in order to win OUT-WUF-CMA, all parts of the ciphertext must be valid, but only one successful forgery is necessary. Consequently, if the plaintext is new and the validity checks pass, we require at least one of these three booleans to be set to true as a winning condition. These $forged$ booleans represent a valid forgery on some part of the ciphertext, and as such they encompass both the freshness of a value or set thereof, and their validity.

The last category of new variables is a group of sets keeping track of the output of the signcryption oracle, such as the associations of session key values sk with ciphertext block values $(c^0, ..., c^n)$. These sets allow us to check

Game $G_0^{\mathcal{A}}()$

$\mathcal{S} \leftarrow \emptyset$
$(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
$(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
$c \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
$valid \leftarrow \top$
$sk \leftarrow \text{OpenPGP.PKE.Dec}_{prk}(c_{sk})$
if $sk = \bot$
  $valid \leftarrow \bot$
for i from 0 to n
  $m^i \leftarrow \text{OpenPGP.SE.Dec}_{sk}(c^i)$
  if $m^i = \bot$
    $valid \leftarrow \bot$
  $\sigma^i \leftarrow \text{OpenPGP.PKE.Dec}_{prk}(c^i_\sigma)$
  if $\sigma^i = \bot$
    $valid \leftarrow \bot$
  if $\text{OpenPGP.DS.Vfy}_{vk}(m^i, sigma^i) = \bot$
    $valid \leftarrow \bot$
  if $\text{OpenPGP.Hash.h}(c^i) \neq h^i$
    $valid \leftarrow \bot$
if $\text{OpenPGP.DS.Vfy}_{vk}(h^0\|...\|h^n, \sigma_{manifest}) = \bot$
  $valid \leftarrow \bot$
if $valid = \bot$
  $m \leftarrow \bot$
else
  $m \leftarrow m^0\|...\|m^n$
return $m \neq \bot$ and
    $m \notin \mathcal{S}$

Oracle $sc(m)$

$sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
$c_{sk} \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(sk)$
$m^0, ..., m^n \leftarrow \text{ProtonDrive.SplitFile}(m)$
for i from 0 to n
  $c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(m^i)$
  $h^i \leftarrow \text{OpenPGP.Hash.h}(c^i)$
  $\sigma^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(m^i)$
  $c^i_\sigma \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
$\sigma_{manifest}$
  $\leftarrow \text{OpenPGP.DS.Sign}_{sik}(h^0\|...\|h^n)$
$c \xleftarrow{\$} \mathsf{SC}_{sik,puk}(m)$
$\mathcal{S} \xleftarrow{\cup} \{m\}$
return $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n,$
  $c^0_\sigma, ...c^n_\sigma, \sigma_{manifest})$

Figure 4.8: The starting game for the authenticity proof, G_0. This corresponds to OUT-WUF-CMA$_{\text{ProtonDrive.File}}$.

Game G_0	$\equiv$	Game $G_{0.5}$	$\equiv$	Game G_1
1 [...]		1 [...]		1 [...]
2 **return** $m \neq \perp$		2 **if** $m \in \mathcal{S}$		2 **if** $(sk, c^0, ..., c^n) \in \mathcal{S}_{sk,c}$
and $m \notin \mathcal{S}$		3 **return** $\perp$		or $m \in \mathcal{S}$
		4 **else**		3 **return** $\perp$
		5 **if** $m \neq \perp$		4 **elseif** $(sk, c^0, ..., c^n) \notin \mathcal{S}_{sk,c}$
		6 **return** $\top$		5 **if** $m \neq \perp$
		7 **else**		6 $bad \leftarrow \top$
		8 **return** $\perp$		7 **return** $\top$
				8 **else**
				9 **return** $\perp$

Figure 4.9: Equivalences between game return values used to go from G_0 in Fig. 4.8 to G_1 in Fig. 4.10. $\mathcal{S}$ is the set of all the plaintexts queried to the signcryption oracle, and $\mathcal{S}_{sk,c}$ that of all the combinations of session key sk and ciphertext blocks $(c^0, ..., c^n)$ returned by that same oracle.

whether an adversary returning a valid ciphertext managed to perform a forgery on a particular component of the ciphertext, or whether it reused some value it received as part of one of its queries to the signcryption oracle.

We also rewrite the returning part of the game and merge the validity checks to it. This change is based on a series of equivalences between winning conditions, which we give in Fig. 4.9.

In Fig. 4.9, the first change from G_0 to game $G_{0.5}$ simply makes the return values more explicit by splitting the return statement from G_0 into cases. G_1 then expands on the distinction between m having already been queried before or not, by using the combination $(sk, c^0, ..., c^n)$ of session keys and ciphertext blocks that have been returned by the signcryption oracle. The key observation is the following: if the oracle has already returned a ciphertext with the combination $(sk, c^0, ..., c^n)$ before, then the corresponding plaintext m has already been queried. This is due to the correctness of OpenPGP.SE. Therefore, if this is the case, we return $\perp$, and we also return $\perp$ in the cases where $(sk, c^0, ..., c^n)$ has never appeared before but corresponds to a plaintext m which has already been queried to the oracle. If this is not the case, we then check whether the values returned by the adversary form a valid signcryption. If it does not, we return $\perp$, and otherwise we set a *bad* flag to $\top$ to indicate that the adversary succeeded and return $\top$. This is a setup for the following steps, where we will isolate every winning path for the adversary and does not affect the return value in G_1.

Each subsequent game serves to eliminate one of the potential attack angles.

The three possible ways for an adversary to forge a ciphertext for a fresh plaintext are each represented by one of the *bad* flags. Hereafter, we explain what path each game discards.

G_2 When $bad_{m,\sigma}$ is set, the adversary returns a ciphertext containing exactly the same ciphertext blocks $(c^0, ..., c^n)$ as one of the ciphertexts it got from the oracle. Let us call m the plaintext which was sent to the oracle in that query and sk the session key that was used. Since we already tested whether the plaintext m' corresponding to the value returned by the adversary was queried before, it is necessarily different from m, and the session key sk' needs to be different from sk by correctness of OpenPGP.SE. This means that the adversary has found a new session key sk' under which all the ciphertext blocks decrypt correctly, which is not possible if the underlying SE scheme is key robust.

Additionally, for the ciphertext to be valid, the adversary needs to have found valid encrypted signature blocks. That implies that it also managed to perform a forgery on the signature scheme OpenPGP.DS.

G_3 When bad_h is set, the adversary returns a ciphertext containing a new combination of ciphertext blocks but the ciphertext hash digests combination has appeared in one of the previous queries. This means that the adversary has managed to find a hash collision.

G_4 When bad_{man} is set, the adversary returns a ciphertext containing a new combination of ciphertext blocks and the ciphertext hash digests combination has not appeared in any of the previous queries. In that case, the adversary has successfully forged the manifest signature.

Games G_1 through G_4 are all given in full detail in Fig. 4.10.

We start by computing a bound on the probability that the $bad_{m,\sigma}$ flag is set to *true*. Let us call sk the session key encrypted in the ciphertext returned by the adversary, and $(c^0, ..., c^n)$ the ciphertext blocks in that ciphertext. The $bad_{m,\sigma}$ flag can only be set inside the if statement on line 17. Because of this, we know that there has been an oracle query returning a ciphertext with a session key sk' different from sk and the same ciphertext blocks $(c^0, ..., c^n)$ as the adversary. In that case, due to the check on line 11, we know that the plaintext blocks $(m^0, ..., m^n)$ corresponding to the ciphertext of the adversary are new, and so are the signatures $(\sigma^0, ..., \sigma^n)$ on those. This means two security properties are broken, with triplets (c^i, sk, sk') breaking the key robustness of OpenPGP.SE for all i's, and at least one index k for which (m^k, σ^k) breaks SUF-CMA$_{\text{OpenPGP.DS}}$. Therefore, there are two ways to bound the probability of $bad_{m,\sigma}$. We will build two adversaries, $\mathcal{B}_{\text{SE}}$ and $\mathcal{B}_{\text{DS},1}$, from each of which we will deduce a bound on that probability.

We construct a first adversary $\mathcal{B}_{\text{SE}}$ calling $\mathcal{A}$ with a simulated oracle sc', which it runs with sender and receiver key values that it generated. We

Games $G_1^{\mathcal{A}}()$ - $G_4^{\mathcal{A}}()$

$\mathcal{S}, \mathcal{S}_{sk,c}, \mathcal{S}_c, \mathcal{S}_h \leftarrow \emptyset$
$(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
$(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
$(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma,$
$\quad \sigma_{man}) \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
$valid_{m,\sigma}, valid_h, valid_{man} \leftarrow \bot$
$forged_{m,\sigma}, forged_h, forged_{man} \leftarrow \bot$
$sk \leftarrow \textsc{OpenPGP.PKE.Dec}_{prk}(c_{sk})$
for i from 0 to n
  $m^i \leftarrow \textsc{OpenPGP.SE.Dec}_{sk}(c^i)$
  $\sigma^i \leftarrow \textsc{OpenPGP.PKE.Dec}_{prk}(c^i_\sigma)$
if $(sk, c^0, ..., c^n) \in \mathcal{S}_{sk,c}$ or $m^0 \| ... \| m^n \in \mathcal{S}$
  return $\bot$
if $sk \neq \bot$
  if $m^i \neq \bot$ and $\sigma^i \neq \bot \quad \forall 0 \leq i \leq n$
    if $\textsc{OpenPGP.DS.Vfy}_{vk}(m^i, \sigma^i) \quad \forall 0 \leq i \leq n$
    $valid_{m,\sigma} \leftarrow \top$
      if $(sk, c^0, ..., c^n) \notin \mathcal{S}_{sk,c}$ and $(c^0, ..., c^n) \in \mathcal{S}_c$
        $forged_{m,\sigma} \leftarrow \top$
        $bad_{m,\sigma} \leftarrow \top$
        $forged_{m,\sigma} \leftarrow \bot$ // G_2 - G_4
if $\textsc{OpenPGP.Hash.h}(c^i) = h^i \quad \forall 0 \leq i \leq n$
  $valid_h \leftarrow \top$
  if $(c^0, ..., c^n) \notin \mathcal{S}_c$ and $(h^0, ..., h^n) \in \mathcal{S}_h$
    $forged_h \leftarrow \top$
    $bad_h \leftarrow \top$
    $forged_h \leftarrow \bot$ // G_3 - G_4
if $\textsc{OpenPGP.DS.Vfy}_{vk}(h^0 \| ... \| h^n, \sigma_{man})$
  $valid_{man} \leftarrow \top$
  if $(h^0, ..., h^n) \notin \mathcal{S}_h$
    $forged_{man} \leftarrow \top$
    $bad_{man} \leftarrow \top$
    $forged_{man} \leftarrow \bot$ // G_4
return $valid_{m,\sigma}$ and $valid_h$ and $valid_{man}$ and
  $(forged_{m,\sigma}$ or $forged_h$ or $forged_{man})$

Oracle $sc(m)$

$sk \xleftarrow{\$} \textsc{OpenPGP.SE.Gen}()$
$c_{sk} \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(sk)$
$m^0, ..., m^n \leftarrow \textsc{ProtonDrive.SplitFile}(m)$
for i from 0 to n
  $c^i \xleftarrow{\$} \textsc{OpenPGP.SE.Enc}_{sk}(m^i)$
  $h^i \leftarrow \textsc{OpenPGP.Hash.h}(c^i)$
  $\sigma^i \xleftarrow{\$} \textsc{OpenPGP.DS.Sign}_{sik}(m^i)$
  $c^i_\sigma \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
$\sigma_{manifest}$
  $\leftarrow \textsc{OpenPGP.DS.Sign}_{sik}(h^0 \| ... \| h^n)$
$c \xleftarrow{\$} \mathsf{SC}_{sik,puk}(m)$
$\mathcal{S} \xleftarrow{\cup} \{m\}$
$\mathcal{S}_{sk,c} \xleftarrow{\cup} \{(sk, c^0, ..., c^n)\}$
$\mathcal{S}_c \xleftarrow{\cup} \{(c^0, ..., c^n)\}$
$\mathcal{S}_h \xleftarrow{\cup} \{(h^0, ..., h^n)\}$
return $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n,$
  $c^0_\sigma, ... c^n_\sigma, \sigma_{manifest})$

Figure 4.10: The games G_1 through G_4 that are used for the proof of OUT-WUF-CMA.

Adversary $\mathcal{B}_{\text{SE}}()$

1 $\mathcal{S} \leftarrow \emptyset$
2 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
3 $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
4 $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma, \sigma_{manifest}) \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
5 $sk \leftarrow \text{OpenPGP.PKE.Dec}_{prk}(c_{sk})$
6 Find sk' such that $(sk', c^0) \in \mathcal{S}$ and $sk \neq sk'$
7 **return** (c^0, sk, sk')

Simulated oracle $sc'(m)$

1 $sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
2 $c_{sk} \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(sk)$
3 $m^0, ..., m^n \leftarrow \text{ProtonDrive.SplitFile}(m)$
4 **for** i from 0 to n
5 $\quad c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(m^i)$
6 $\quad h^i \leftarrow \text{OpenPGP.Hash.h}(c^i)$
7 $\quad \sigma^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(m^i)$
8 $\quad c^i_\sigma \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
9 $\sigma_{manifest} \leftarrow \text{OpenPGP.DS.Sign}_{sik}(h^0 \| ... \| h^n)$
10 $\mathcal{S} \xleftarrow{\cup} \{(sk, c^0)\}$
11 **return** $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ...c^n_\sigma, \sigma_{manifest})$

Figure 4.11: The simulated adversary against $\text{KROB}_{\text{OpenPGP.SE}}$ using adversary $\mathcal{A}$.

let $\mathcal{B}_{\text{SE}}$ keep track of all the pairs (sk, c^0) appearing in the queries to the simulated oracle in a set $\mathcal{S}$. Note that we do not need to keep track of the values of ciphertext blocks for other indices because we know from line 17 that all c^i's from the ciphertext of the adversary are identical to the ciphertext blocks that were yielded from some query to the oracle, i.e. the c^0 of the adversary is equal to that of the oracle. Then, when $\mathcal{A}$ returns a ciphertext, we retrieve the session key sk' it uses by decrypting the encrypted session key and find the session key sk with which the first ciphertext block c^0 has been yielded by the simulated oracle. Finally, the adversary $\mathcal{B}_{\text{SE}}$ returns the triplet (c^0, sk, sk'). The full details are given in Fig. 4.11.

From this reduction, we deduce

$$Pr[bad_m] \leq \mathcal{A}dv^{\text{KROB}}_{\text{OpenPGP.SE}}(\mathcal{B}_{\text{SE}}). \tag{4.9}$$

We then construct a second adversary $\mathcal{B}_{\text{DS},1}$ calling $\mathcal{A}$ with a simulated oracle sc', which it runs with receiver key values that it generated. Signature performed by sc' are delegated to the signature oracle of $\text{SUF-CMA}_{\text{OpenPGP.DS}}$. We let $\mathcal{B}_{\text{DS},1}$ keep track of all the pairs (m^i, σ^i) appearing in the queries to the simulated oracle in a set $\mathcal{S}$. Then, when $\mathcal{A}$ returns a ciphertext, we decrypt the c^i's and c^i_σ's until we find a pair (m^i, σ^i) which does not appear in the set $\mathcal{S}$. The adversary $\mathcal{B}_{\text{DS},1}$ returns that pair. The full details are given in Fig. 4.12.

Adversary $\mathcal{B}_{\mathsf{DS},1}(vk)$

$\mathcal{S} \leftarrow \emptyset$
$(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
$(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma, \sigma_{manifest}) \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
$sk \leftarrow \textsc{OpenPGP.PKE}_{prk}(c_{sk})$
for i from 0 to n
  $m^i \leftarrow \textsc{OpenPGP.SE.Dec}_{sk}(c^i)$
  $\sigma^i \leftarrow \textsc{OpenPGP.PKE.Dec}_{prk}(c^i_\sigma)$
  if $(m^i, \sigma^i) \notin \mathcal{S}$
    return (m^i, σ^i)
return $\perp$

Simulated oracle $sc'(m)$

$sk \xleftarrow{\$} \textsc{OpenPGP.SE.Gen}()$
$c_{sk} \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(sk)$
$m^0, ..., m^n \leftarrow \textsc{ProtonDrive.SplitFile}(m)$
for i from 0 to n
  $c^i \xleftarrow{\$} \textsc{OpenPGP.SE.Enc}_{sk}(m^i)$
  $h^i \leftarrow \textsc{OpenPGP.Hash.h}(c^i)$
  $\sigma^i \xleftarrow{\$} sign(m^i)$
  $\mathcal{S} \xleftarrow{\cup} \{(m^i, \sigma^i)\}$
  $c^i_\sigma \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
$\sigma_{manifest} \leftarrow sign(h^0 \| ... \| h^n)$
return $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ...c^n_\sigma, \sigma_{manifest})$

Figure 4.12: The simulated adversary against SUF-CMA$_{\textsc{OpenPGP.DS}}$ using adversary $\mathcal{A}$.

From this reduction, we deduce

$$Pr[bad_{m,\sigma}] \leq \mathcal{A}dv^{\text{SUF-CMA}}_{\textsc{OpenPGP.DS}}(\mathcal{B}_{\mathsf{DS},1}). \tag{4.10}$$

We now have two bounds on $Pr[bad_{m,\sigma}]$, Eq. (4.9) and Eq. (4.10). We can combine the two into one by observing that whichever advantage is the lowest will give us the best bound out of the two, and therefore keep

$$Pr[bad_{m,\sigma}] \leq min\{\mathcal{A}dv^{\text{KROB}}_{\textsc{OpenPGP.SE}}(\mathcal{B}_{\mathsf{SE}}), \mathcal{A}dv^{\text{SUF-CMA}}_{\textsc{OpenPGP.DS}}(\mathcal{B}_{\mathsf{DS},1})\}. \tag{4.11}$$

Next, we bound the probability that the bad_h flag is raised. When this happens, the adversary $\mathcal{A}$ has performed a successful forgery where the ciphertext block hashes $(h^0, ..., h^n)$ have already appeared in some ciphertext which has been yielded by the oracle sc, but with ciphertext blocks $(c^0, ..., c^n)$ which are different from the ciphertext blocks $(c^{0\prime}, ..., c^{n\prime})$ that the adversary has returned. This means that there is an index k for which $c^k \neq c^{k\prime}$, and that there is a hash collision on those two values.

We can build an adversary $\mathcal{B}_{\mathtt{Hash}}$ calling adversary $\mathcal{A}$ with a simulated oracle sc', which it runs with sender and receiver key values that it generated. We let $\mathcal{B}_{\mathtt{Hash}}$ keep track of all the ciphertext block to digest pairs returned by the simulated oracle in a set $\mathcal{S}$. When $\mathcal{A}$ returns a ciphertext with ciphertext

Adversary $\mathcal{B}_{\texttt{Hash}}()$

1 $\mathcal{S} \leftarrow \emptyset$
2 $(vk, sik) \xleftarrow{\$} \mathsf{SGen}()$
3 $(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
4 $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma, \sigma_{manifest}) \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
5 **if** $\exists k, c'.(c', h^k) \in \mathcal{S}$ and $c' \neq c^k$
6 **return** (c', c^k)
7 **return** $\perp$

Simulated oracle $sc'(m)$

1 $sk \xleftarrow{\$} \text{OpenPGP.SE.Gen}()$
2 $c_{sk} \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(sk)$
3 $m^0, ..., m^n \leftarrow \text{ProtonDrive.SplitFile}(m)$
4 **for** i from 0 to n
5 $c^i \xleftarrow{\$} \text{OpenPGP.SE.Enc}_{sk}(m^i)$
6 $h^i \leftarrow \text{OpenPGP.Hash.h}(c^i)$
7 $\sigma^i \xleftarrow{\$} \text{OpenPGP.DS.Sign}_{sik}(m^i)$
8 $c^i_\sigma \xleftarrow{\$} \text{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
9 $\sigma_{manifest} \leftarrow \text{OpenPGP.DS.Sign}_{sik}(h^0 \| ... \| h^n)$
10 $c \xleftarrow{\$} \mathsf{SC}_{sik,puk}(m)$
11 **for** i from 0 to n
12 $\mathcal{S} \xleftarrow{\cup} \{(c^i, h^i)\}$
13 **return** $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ...c^n_\sigma, \sigma_{manifest})$
14 **return** $\perp$

Figure 4.13: The simulated adversary against $\text{CR}_{\text{OpenPGP.Hash}}$ using adversary $\mathcal{A}$.

blocks $(c^0, ..., c^n)$ and hashes $(h^0, ..., h^n)$, we search in $\mathcal{S}$ what hash digest h^k has already been returned for a ciphertext block $c \neq c^k$. Our adversary $\mathcal{B}_{\texttt{Hash}}$ then returns the pair (c, c^k). The full details of this reduction are presented in Fig. 4.13.

From this reduction, we deduce

$$Pr[bad_h] \leq \mathcal{A}dv^{\text{CR}}_{\text{OpenPGP.Hash}}(\mathcal{B}_{\text{SE}}). \tag{4.12}$$

We give a fourth reduction in order to bound the probability that the bad_{man} flag is raised. This happens when the adversary $\mathcal{A}$ performs a successful forgery where the ciphertext block hashes combination $(h^0, ..., h^n)$ has never been returned as part of a query before. This means that the signed manifest $\sigma_{manifest}$ is a new valid signature on the concatenation of hashes.

We can build an adversary $\mathcal{B}_{\text{DS},2}$ playing $\text{SUF-CMA}_{\text{OpenPGP.DS}}$ which calls adversary $\mathcal{A}$ with a simulated oracle sc' running with receiver key values (puk, prk) which it generates, and delegates all signatures to the oracle $sign$ of the $\text{SUF-CMA}_{\text{OpenPGP.DS}}$ game. Then, if the adversary returns a ciphertext

Adversary $\mathcal{B}_{\mathsf{DS},2}(vk)$

$(puk, prk) \xleftarrow{\$} \mathsf{RGen}()$
$(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ..., c^n_\sigma, \sigma_{manifest}) \xleftarrow{\$} \mathcal{A}^{sc}(vk, puk)$
return $(h^0 \| ... \| h^n, \sigma_{manifest})$

Simulated oracle $sc'(m)$

$sk \xleftarrow{\$} \textsc{OpenPGP.SE.Gen}()$
$c_{sk} \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(sk)$
$m^0, ..., m^n \leftarrow \textsc{ProtonDrive.SplitFile}(m)$
for i from 0 to n
  $c^i \xleftarrow{\$} \textsc{OpenPGP.SE.Enc}_{sk}(m^i)$
  $h^i \leftarrow \textsc{OpenPGP.Hash.h}(c^i)$
  $\sigma^i \xleftarrow{\$} sign(m^i)$
  $c^i_\sigma \xleftarrow{\$} \textsc{OpenPGP.PKE.Enc}_{puk}(\sigma^i)$
$\sigma_{manifest} \leftarrow sign(h^0 \| ... \| h^n)$
$c \xleftarrow{\$} \mathsf{SC}_{sik,puk}(m)$
for i from 0 to n
  $\mathcal{S} \xleftarrow{\cup} \{(c^i, h^i)\}$
return $(c_{sk}, c^0, ..., c^n, h^0, ..., h^n, c^0_\sigma, ...c^n_\sigma, \sigma_{manifest})$

Figure 4.14: The simulated adversary against SUF-CMA$_{\textsc{OpenPGP.DS}}$ using adversary $\mathcal{A}$.

with hashes $(h^0, ..., h^n)$ and signed manifest $\sigma_{manifest}$, $\mathcal{B}_{\mathsf{DS},2}$ simply returns the pair $(h^0 \| ... \| h^n, \sigma_{manifest})$.

From this reduction, we have

$$Pr[bad_{man}] \leq \mathcal{A}dv^{\text{SUF-CMA}}_{\textsc{OpenPGP.DS}}(\mathcal{B}_{\mathsf{DS},2}). \tag{4.13}$$

Finally, we can bound the advantage of an adversary $\mathcal{A}$ playing the game

OUT-WUF-CMA$_{\text{ProtonDrive.File}}$.

$$\begin{aligned}
\mathcal{A}dv^{\text{OUT-WUF-CMA}}_{\text{ProtonDrive.File}}(\mathcal{A}) &= Pr[G_0^{\mathcal{A}}()] && (4.14)\\
&= (Pr[G_0^{\mathcal{A}}()] - Pr[G_1^{\mathcal{A}}()]) + \\
&\quad (Pr[G_1^{\mathcal{A}}()] - Pr[G_2^{\mathcal{A}}()]) + \\
&\quad (Pr[G_2^{\mathcal{A}}()] - Pr[G_3^{\mathcal{A}}()]) + && (4.15)\\
&\quad (Pr[G_3^{\mathcal{A}}()] - Pr[G_4^{\mathcal{A}}()]) + \\
&\quad Pr[G_4^{\mathcal{A}}()] \\
&= Pr[bad_{m,\sigma}] + Pr[bad_h] + Pr[bad_{man}] && (4.16)\\
&\leq min\{\mathcal{A}dv^{\text{KROB}}_{\text{OpenPGP.SE}}(\mathcal{B}_{\text{SE}}), \mathcal{A}dv^{\text{SUF-CMA}}_{\text{OpenPGP.DS}}(\mathcal{B}_{\text{DS},1})\} \\
&\quad + \mathcal{A}dv^{\text{CR}}_{\text{OpenPGP.Hash}}(\mathcal{B}_{\text{SE}}) && (4.17)\\
&\quad + \mathcal{A}dv^{\text{SUF-CMA}}_{\text{OpenPGP.DS}}(\mathcal{B}_{\text{DS},2})
\end{aligned}$$

Eq. (4.14) is an application of the definition of advantage, where we have OUT-WUF-CMA$_{\text{ProtonDrive.File}} = G_0$. Eq. (4.15) is a telescopic extension of the sum such that all of our games appear in it. In Eq. (4.16), $Pr[G_0^{\mathcal{A}}()] - Pr[G_1^{\mathcal{A}}()]$ cancels out because as previously argued, G_0 and G_1 are equivalent, $Pr[G_1^{\mathcal{A}}()] - Pr[G_2^{\mathcal{A}}()]$ is replaced by $Pr[bad_{m,\sigma}]$, $Pr[G_2^{\mathcal{A}}()] - Pr[G_3^{\mathcal{A}}()]$ by $Pr[bad_h]$, and $Pr[G_3^{\mathcal{A}}()] - Pr[G_4^{\mathcal{A}}()]$ by $Pr[bad_{man}]$ using Lemma 4.4 because the game pairs G_1-G_2, G_2-G_3, and G_3-G_4 are all equivalent until bad, and $Pr[G_4^{\mathcal{A}}()] = 0$ because G_4 always returns $\perp$ due to all $forged$ booleans having value $false$. For our last step Eq. (4.17), we simply apply the bounds Eq. (4.11), Eq. (4.12), and Eq. (4.13) which we previously computed. □

4.2 Likeliness of the Assumptions

In Section 4.1, we have used some assumptions for our proofs. Hereafter, we discuss how likely it is that these assumptions hold, and therefore whether the bounds we computed are meaningful.

4.2.1 IND-CPA$_{\text{OpenPGP.SE}}$

We know that AES-CFB is IND-CPA secure assuming that AES-256 is a pseudo random permutation and that the IV it uses is not predictable. Wooding [55] gives a proof of this. The modifications brought to AES-CFB by OpenPGP.SE are such that we cannot apply a simple reduction to show that OpenPGP.SE is also IND-CPA secure. This is due to the fact that OpenPGP.SE uses a fixed IV composed of all zeroes, and the first block of the plaintext it passes to AES-CFB, which is randomly generated, serves as the randomness source. We leave proving that OpenPGP.SE is IND-CPA to future work, but consider it to be a reasonable assumption.

4.2.2 IND-CPA$_{\mathsf{OpenPGP.PKE}}$

As described in Section 2.3.2, OpenPGP.PKE uses a combination of key exchange, key wrapping, key generation and symmetric encryption to encrypt data. At its core however, it passes plaintexts to OpenPGP.SE with a randomly generated session key for encryption. Therefore, its IND-CPA security can be reduced to that of OpenPGP.SE.

4.2.3 SUF-CMA$_{\mathsf{OpenPGP.DS}}$

The algorithms used for signatures in OpenPGP is Ed25519, an EdDSA elliptic curve algorithm using Curve 25519. Brendel et al. have proven Ed25519 to be SUF-CMA secure in [15].

4.2.4 KROB$_{\mathsf{OpenPGP.SE}}$

The construction of OpenPGP.SE, as we presented it in Section 2.4.4, includes some elements aiming to provide integrity on the ciphertext. In particular, as explained in RFC 4880 [27, Section 5.7], the repetition of two of the random bytes at the start of the input to AES-CFB is intended to help detect incorrect session keys. Note that this is not designed as a way to produce key robustness, but rather as a quick check on whether the right key way used; the probability that the two repeated bytes are the same as the two bytes right before when using a bad session key is 2^{-16}, which is too high to ensure that no two keys can give a valid decryption of a ciphertext. OpenPGP.SE also adds two constant bytes and a SHA-1 digest to the input to AES-CFB. While it is highly likely that this further decreases the chances of an adversary breaking key robustness on that scheme, we do not make any claim that this suffices to make OpenPGP.SE key robust, and leave it to future work to prove or disprove it.

More notably however, the advantage of an adversary playing KROB$_{\text{OpenPGP.SE}}$ only intervenes in Eq. (4.17) if it is smaller than that of SUF-CMA$_{\text{OpenPGP.DS}}$. This means that even if OpenPGP.SE is not KROB secure, we still have that ProtonDrive.File is OUT-WUF-CMA secure as long as OpenPGP.DS is SUF-CMA secure.

4.2.5 CR$_{\mathsf{OpenPGP.Hash}}$

The algorithm used for OpenPGP.Hash is SHA256. While there is no proof that it is collision resistant, it is generally assumed to fulfill that property.

4.2.6 Conclusion

We computed the advantages of adversaries playing against OUT-IND-CPA, OUT-SUF-CMA, OUT-IND-CCA, and OUT-WUF-CMA on the scheme O-

penPGP.File. For both OUT-SUF-CMA and OUT-IND-CCA, we showed how to construct an adversary which gets an advantage that is arbitrarily close to 1. In contrast, we proved that ProtonDrive.File achieves better bounds for the two other security properties.

For OUT-IND-CPA$_{\text{ProtonDrive.File}}$, the security relies on both OpenPGP.SE and OpenPGP.PKE being IND-CPA secure. Furthermore, as we argued in Section 4.2.2, we can conjecture that the IND-CPA security of OpenPGP.PKE relies on the IND-CPA security of OpenPGP.SE. Therefore, we conclude that the OUT-IND-CPA security of ProtonDrive.File can be reduced to OpenPGP.SE being IND-CPA secure, which, while it has not been proven to be true, is not an unreasonable assumption to make.

For OUT-WUF-CMA$_{\text{ProtonDrive.File}}$, the bound that we get uses the KROB security of OpenPGP.SE, the collision resistance of OpenPGP.Hash, and the SUF-CMA security of OpenPGP.DS. We know that OpenPGP.DS is SUF-CMA secure and make the assumption that OpenPGP.Hash is collision resistant. Since the advantage of an adversary against OpenPGP.DS is established to be negligible, we get a good bound on the advantage of an adversary against OUT-WUF-CMA$_{\text{ProtonDrive.File}}$ even if OpenPGP.SE is not KROB secure.

Chapter 5

Caveats

In this chapter, we address some weaknesses we found in the design of Proton Drive, some of which could lead to attacks.

5.1 File Encryption is not OUT-IND-CCA

As we showed in Section 4.1.3, the file encryption algorithm used by Proton Drive does not satisfy the definition of OUT-IND-CCA from Section 2.2.5, which we determined to be the security goal to strive for based on the threat model we presented in Section 3.3. Note that this is due to a technicality in the definition of OUT-IND-CCA, and that we did not manage to find real-world attacks linked to this property not being satisfied. Because of the lack of formal models for cloud storage, it is hard to say exactly what confidentiality notion ProtonDrive.File would need to satisfy for the whole protocol to fit the E2EE definition. Rather than highlighting an imminent risk, this observation should be seen as a good opportunity to refine the understanding and application of security definitions within the algorithm design process.

We also note that due to laxness in the format of OpenPGP messages, which are valid with different padding lengths, a redesign of the ProtonDrive.File algorithm with the objective to satisfy OUT-IND-CCA would be inefficient without a revision of the underlying cryptographic library. Indeed, the malleability in the message format could be exploited to query the oracle for the decryption of an encryption query result, similarly to the attack we presented in Section 4.1.3.

5.2 Signatures

The way signature verification results are treated in Proton Drive at the time of writing is lacking. Proton makes the choice of giving a warning to the user

rather than preventing access to files which fail signature checks. The reason for this is that some files are expected to have bad signatures or lack them altogether, even without having been produced in an irregular way, typically for files which were added before the incorporation of some signatures to the protocol. The visibility of this warning is subject to variations depending on the client (web, Android, iOS, Windows, or Linux), but it is generally insufficient. This can lead to users being unaware of signature check failures, which is a problem because signatures are needed for authenticity.

As we pointed out in Section 3.1, OpenPGP encryption by itself does not provide ciphertext integrity. Because the public part of asymmetric keys is not encrypted for storage, public keys are stored in clear on the server, meaning that a strong adversary which has access to the data stored on the server can gain access to them. If a user is not alerted of failed signature checks, an adversary can easily create new folders or files with invalid signatures, and add them to the file system without being detected.

There are two ways in which this can be a problem. The first one is for framing attacks, where an attacker puts compromising content in the file system of a user. The second one however is a confidentiality breach. If the adversary adds a new folder to the file system of a user, it knows the encryption keys associated to that folder, which will be used to encrypt all the children of that folder. This means that the adversary will be able to decrypt anything that the user then puts in that folder. If this attack is executed to substitute the root of the file system, then the adversary can gain access to everything the user subsequently stores.

To mitigate this, we propose two modifications. The most urgent one is to increase the visibility of warnings on all clients, to ensure that users cannot ignore them when something goes wrong. Secondly, we suggest to provide the users with the option to review the contents of a file that fails signature checks and decide whether to re-encrypt or discard the file, but note that this option is not risk-free for files of uncertain origin.

Another thing to note about signatures is that a given user has one address signature key which it uses by default for all its signatures across all Proton applications. This lack of key separation may be a problem if signatures from one application can be used by an adversary to create a valid value for another application. Proton plans to add context to signatures in order to alleviate this problem.

5.3 File Names

In the current implementation of Proton Drive, there is nothing binding a folder or file to its name, as both objects are encrypted separately. The only bound on folder and file names is to their parent, because they use its keys

for encryption, as described in Section 3.2.3. Because of this, the names associated to the children of a given folder can be swapped between the children. While this is not a very powerful attack, it remains a breach of integrity on the data stored in Proton Drive. If Proton Drive switches to using authenticated encryption with associated data (AEAD) for symmetric encryption, one way to prevent this attack could be to include the HMAC of the name as associated data in the encryption of the file or folder.

5.4 Compression before Encryption

For the encryption of extended attributes (as described in Section 3.2.3), the data is compressed before encryption. This pattern is known to lead to compression side-channel attacks, which are described by Kelsey [34]. We refer to CRIME/BREACH attacks [24, 29] for concrete examples.

5.5 OpenPGP Format Oracles

Attacks described by Mister et al. [41] and Maury et al. [37] exploit format oracles on OpenPGP.SE in order to achieve full plaintext recovery. If the implementation of one of the Proton Drive clients were to reveal information on the format of the decrypted ciphertext, we could could end up with such a format oracle. For example, this could be the case if the decryption was streamed, because format checks would then happen before the integrity check, which uses the SHA-1 hash placed at the end of the input to AES-CFB. Further examination of the source code of all Proton Drive clients is necessary in order to determine whether such format oracles exist.

Chapter 6

Conclusion

In this thesis, we presented a white paper of the Proton Drive protocol and analyzed the security properties of the file encryption algorithm it uses.

We provided a comprehensive description of the cryptographic protocol underlying Proton Drive, outlining both its key hierarchy and its data encryption algorithms. Notably, two design choices make the Proton Drive protocol convoluted and laborious to analyze. The first one is utilizing OpenPGP for encryption, which imposes the use of some algorithm combination patterns in places where one single cryptographic primitive could have been used. The second is the key hierarchy structure, which follows the structure of the underlying file system. This means that every time a new file or folder is added, the key hierarchy grows in size, which is not only a problem for its complexity, but also takes up more space than necessary.

Next, we defined two security goals based on the claims made by Proton AG on the security of Proton Drive [52, 51], one for confidentiality and one for authenticity. We then evaluated whether the file encryption algorithm meets these goals. For confidentiality, file encryption falls short of the objective of OUT-IND-CCA (Section 4.1.3), but we show that it satisfies a weaker property, OUT-IND-CPA (Section 4.1.1). For authenticity, we prove that the file encryption algorithm meets the security goal OUT-WUF-CMA (Section 4.1.4), but also show that it does not satisfy the stronger property OUT-SUF-CMA (Section 4.1.2). While we could not find attacks associated with the lack of OUT-IND-CCA security, we still consider it a prudent choice of security to aim for, but note that the choice of OpenPGP as a cryptographic library prevents a redesign to achieve this goal, due to the malleability of OpenPGP messages. We generally suggest pondering the ways in which formal security goals can be proven as part of the design process. Not only does this contribute to the immediate assessment of the strengths and weaknesses of the design, but it also facilitates future analyses by keeping the algorithm simpler.

Finally, we pointed to other weaknesses in Proton Drive. The most detrimental is without a doubt the way in which signature verification is handled. Proton makes the choice to warn users rather than reject files when signature verification fails. The problem with this is that the warnings are not visible enough, effectively rendering the presence of signatures and signature verification useless. Users not being aware of failed signature verification nullifies the authenticity which signatures are intended to bring, and, in the worst cases, can lead to complete access to the contents of the file system. We urge that the user interface of all Proton Drive clients be changed to adequately reflect the importance of signature verification, and suggest that users be prompted to approve and reencrypt invalid files. Overall, this approach to signature verification illustrates how conflicts between security and convenience can lead to potential vulnerabilities in an implementation.

Future Work. Given the intricate nature of the Proton Drive protocol, we focused on the file encryption, and a large part of the protocol remains to be analyzed. This includes the file sharing mechanisms, user authentication mechanism, key encryption algorithms, and name and metadata encryption algorithms.

Furthermore, Proton Drive is an evolving protocol, and its evolution will call for revisions of the security goals. Notably, sharing and collaboration on files between Proton users is not implemented yet, but addition of these features to the protocol is planned. This introduction will introduce new challenges, particularly when granting writing permissions to a file for multiple users. This will result in a setting where several people have access to the same encryption key pair, which, for signcryption schemes, increases the security requirements for authenticity from outsider WUF-CMA to insider WUF-CMA.

It is crucial to note that even successfully proving the security of all these components of the protocol would not entirely rule out attacks. Indeed, implementation mistakes could lead to cryptographic attacks as well. Addressing this concern will require a comprehensive examination of the source code of all five Proton Drive clients.

Bibliography

[1] Nextcloud : End-to-end encryption design. https://nextcloud.com/wp-content/uploads/2022/03/Security-Whitepaper-WebVersion-072018.pdf, March 2022. (accessed: 05.07.2023).

[2] 1Password. 1password security design, release v0.4.6 (25.10.2023). https://1passwordstatic.com/files/security/1password-white-paper.pdf, 2023. (accessed: 30.01.2024).

[3] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 480–497. Springer, Heidelberg, February 2010.

[4] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. Caveat implementor! Key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 190–218. Springer, Heidelberg, April 2023.

[5] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *2022 IEEE Symposium on Security and Privacy*, pages 87–106. IEEE Computer Society Press, May 2022.

[6] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[7] Apple. icloud data security overview. https://support.apple.com/en-us/HT202303, March 2023. (accessed: 24.01.2024).

[8] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: Malleable encryption goes awry. Cryptology ePrint Archive, Report 2022/959, 2022. https://eprint.iacr.org/2022/959.

[9] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 602–619. Springer, Heidelberg, August 2006.

[10] Mihir Bellare and Philipp Rogaway. *Introduction to Modern Cryptography*. University of California, Davis, May 2005.

[11] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. https://eprint.iacr.org/2004/331.

[12] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[13] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 784–813. Springer, Heidelberg, August 2022.

[14] Bitwarden. Bitwarden security whitepaper. https://bitwarden.com/help/bitwarden-security-white-paper/. (accessed: 30.01.2024).

[15] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. In *2021 IEEE Symposium on Security and Privacy*, pages 1659–1676. IEEE Computer Society Press, May 2021.

[16] Lara Bruseghini, Daniel Huigens, and Kenneth G. Paterson. Victory by KO: Attacking OpenPGP using key overwriting. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 411–423. ACM Press, November 2022.

[17] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. Titanium: A metadata-hiding file-sharing system with malicious security. Cryptology ePrint Archive, Report 2022/051, 2022. https://eprint.iacr.org/2022/051.

[18] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. Cryptology ePrint Archive, Report 2020/083, 2020. https://eprint.iacr.org/2020/083.

[19] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.

[20] Daniele Coppola. Breaking cryptography in the wild: Nextcloud. https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/report_DanieleCoppola.pdf?cve=title, May 2023.

[21] Dashlane. Dashlane's security principles and architecture, v2.1.2 (11.12.2023). https://www.dashlane.com/download/whitepaper-en.pdf, 2023. (accessed: 30.01.2024).

[22] Alexander W. Dent. A note on game-hopping proofs. Cryptology ePrint Archive, Report 2006/260, 2006. https://eprint.iacr.org/2006/260.

[23] Dropbox. Dropbox business security whitepaper, version v2023.01. https://assets.dropbox.com//www/en-us/business/solutions/solutions/dfb_security_whitepaper.pdf, 2023. (accessed: 13.07.2023).

[24] Thai Duong and Julianno Rizzo. The crime attack. https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit.

[25] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Robust encryption, revisited. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 352–368. Springer, Heidelberg, February / March 2013.

[26] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. Cryptology ePrint Archive, Report 2017/288, 2017. https://eprint.iacr.org/2017/288.

[27] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. OpenPGP Message Format. RFC 4880, November 2007.

[28] Rémi Géraud, David Naccache, and Razvan Rosie. Robust encryption, extended. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 149–168. Springer, Heidelberg, March 2019.

[29] Yoel Gluck, Neal Harris, and Angelo Prado. Breach : Reviving the crime attack. http://css.csail.mit.edu/6.858/2020/readings/breach.pdf.

[30] Google. Google cloud default encryption at rest. https://cloud.google.com/docs/security/encryption/default-encryption#googles_default_encryption, September 2022. (accessed: 24.01.2023).

[31] Miro Haller. Cloud storage systems: From bad practice to practical attacks, March 2022.

[32] Mario Heiderich and Nadim Kobeissi. Review report passbolt security white paper 02.2021. https://help.passbolt.com/assets/files/PBL-01-report.pdf, February 2021.

[33] IDrive. Strong security and privacy with private key. https://www.idrive.com/online-backup-security. (accessed: 24.01.2024).

[34] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 263–276. Springer, Heidelberg, February 2002.

[35] Nadim Kobeissi. An analysis of the ProtonMail cryptographic architecture. Cryptology ePrint Archive, Report 2018/1121, 2018. https://eprint.iacr.org/2018/1121.

[36] Takahiro Matsuda, Kanta Matsuura, and Jacob C. N. Schuldt. Efficient constructions of signcryption schemes and signcryption composability. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT 2009*, volume 5922 of *LNCS*, pages 321–342. Springer, Heidelberg, December 2009.

[37] Florian Maury, Jean-René Reinhard, Olivier Levillain, and Henri Gilbert. Format oracles on OpenPGP. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 220–236. Springer, Heidelberg, April 2015.

[38] David A. McGrew and John Viega. The security and performance of the Galois/counter mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355. Springer, Heidelberg, December 2004.

[39] MEGA. Mega security white paper. https://mega.nz/SecurityWhitepaper.pdf, June 2022. (accessed: 05.07.2023).

[40] Serge Mister and Robert Zuccherato. An attack on CFB mode encryption as used by OpenPGP. Cryptology ePrint Archive, Report 2005/033, 2005. https://eprint.iacr.org/2005/033.

[41] Serge Mister and Robert J. Zuccherato. An attack on CFB mode encryption as used by OpenPGP. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 82–94. Springer, Heidelberg, August 2006.

[42] Nextcloud. Nextcloud encryption and hardening. https://nextcloud.com/encryption/. (accessed: 24.01.2024).

[43] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1289–1306, Anaheim, CA, August 2023. USENIX Association.

[44] pCloud. Features: Encryption. https://www.pcloud.com/features/encryption.html. (accessed: 24.01.2024).

[45] PreVeil. Preveil security and design: a description of the preveil architecture, version 1.5 (07.2023). https://www.preveil.com/wp-content/uploads/2019/10/PreVeil_Security_Whitepaper-v1.5.pdf, 2023. (accessed: 30.01.2023).

[46] Redacted. Audit report ville de genève audit nextcloud. https://nextcloud.com/wp-content/uploads/2022/03/Audit_city_of_Geneva_754-001-v2.0_Redacted.pdf, December 2019. (accessed: 24.01.2024).

[47] Marek Rzepecki, Piotr Izak, and Michał Bentkowski. Securitum security report, penetration testing of beta.protonmail.com, account.protonmail.com and calendar.protonmail.com web applications, in blackbox and whitebox approach. https://res.cloudinary.com/dbulfrlrz/images/v1685439700/wp/securitum-protonmail-security-audit/securitum-protonmail-security-audit.pdf, May 2021.

[48] Professor Kumkum Saxena, Dev Rajdev, Divesh Bhatia, and Manav Bahl. Protonmail: Advance encryption and security. In *2021 International Conference on Communication information and Computing Technology (ICCICT)*, pages 1–6, 2021.

[49] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. https://eprint.iacr.org/2004/332.

[50] Sync. What can i do to ensure my files are encrypted and my sync account is secure? https://www.sync.com/help/

what-can-i-do-to-ensure-my-files-are-encrypted-and-my-sync-account-is-secur (accessed: 24.01.2024).

[51] Proton Team. Proton drive security. https://proton.me/drive/security.

[52] Proton Team. Proton drive threat model. https://proton.me/blog/proton-drive-threat-model, Mar 2023.

[53] Tresorit. Tresorit encryption whitepaper. https://cdn.tresorit.com/202208011608/tresorit-encryption-whitepaper.pdf. (accessed: 30.01.2024).

[54] Nikos Virvilis, Stelios Dritsas, and Dimitris Gritzalis. Secure cloud storage: Available infrastructures and architectures review and evaluation. In Steven Furnell, Costas Lambrinoudakis, and Günther Pernul, editors, *TrustBus 2011*, volume 6863 of *LNCS*, pages 74–85. Springer, Heidelberg, August / September 2011.

[55] Mark Wooding. New proofs for old modes. Cryptology ePrint Archive, Report 2008/121, 2008. https://eprint.iacr.org/2008/121.

[56] Paul Wouters, Daniel Huigens, Justus Winter, and Niibe Yutaka. OpenPGP. Internet-Draft draft-ietf-openpgp-crypto-refresh-10, Internet Engineering Task Force, June 2023. Work in Progress.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.