Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Security Assessment of the Sharekey Collaboration App

Master Thesis

Pascal Schärli

April, 2023

Advisors: Prof. Kenny Paterson, Dr. Bernhard Tellenbach

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

Abstract

This thesis reviews the security and privacy of Sharekey, which is a collaboration platform focusing on privacy and security, allowing users to communicate and share data securely. We conduct a security analysis of Sharekey's protocol and implementation, identifying vulnerabilities and threats to the security and privacy of its users. We begin by providing an overview of Sharekey's cryptographic protocols which we reviewed using a combination of manual code review and automated testing. Our analysis did not contradict Sharekey's main claim of user data confidentiality towards Sharekey, however, we found ways of breaking other properties, uncovering a total of 19 security issues in Sharekey's protocol and implementation. As an example, we discovered a series of exploits that would allow an adversary to destroy a significant number of files and demand a ransom for their recovery. To address these issues, we provide recommendations for improving Sharekey's security posture. Finally, we provide the starting grounds for a redesign of the file-sharing protocol, by proposing improvements to the protocol with stronger integrity guarantees, supported through modeling with Tamarin Prover. Overall this thesis shows the complexity of building secure protocols and the importance of scrutiny towards new protocols.

Acknowledgements

I would like to express my deepest gratitude to Dr. Bernhard Tellenbach, my supervisor, for his invaluable guidance throughout the entire process of writing this thesis. His insightful feedback, extensive knowledge and expertise in the field, and willingness to have extensive weekly meetings going in-depth have been essential to the successful completion of this work. I am grateful for the opportunity to have worked with him.

I am also thankful to Prof. Kenny Paterson for his invaluable guidance and support throughout the review process. It cannot be overstated how his exceptional lectures were what inspired me to do my thesis in this field in the first place. I am grateful for the opportunity to work with him and for the insights I have gained through his mentorship.

I would also like to extend my thanks to Sharekey for their exceptional level of collaboration throughout this research. They provided invaluable support in setting up the necessary infrastructure and remained transparent and communicative throughout the project. Sharekey approached the identified security issues with seriousness and allowed for productive discussions on potential improvements. Their strong commitment to security research on their product is admirable and distinguishes them from other companies. I especially want to thank Hervé for traveling across countries to meet with us in person, and Eugène for his invaluable technical support and grounded discussions.

Furthermore, I would like to thank Xenia for her support and guidance in using Tamarin Prover. Her expertise and insightful tips on how to improve my models and proofs have been highly helpful for the successful design of the protocol in this thesis.

Finally, I am deeply grateful to Julia for her thorough review and intense scrutiny of every sub- and superscript throughout my thesis. Her dedication and attention to detail have been invaluable in improving the overall quality of this thesis and presentation. She went out of her way to spend hours helping me review the thesis, and I am extremely grateful for her support.

Contents

Chapter 1

Introduction

Collaboration apps have become increasingly popular in recent years, allowing users to communicate and share information with colleagues and friends across the globe [1–3]. As these apps handle sensitive information, they must be designed with security in mind to protect user data and maintain privacy. However popular apps like Microsoft Teams, Zoom, and Slack do not offer end-to-end encryption for all of the data shared through them [4, 5], which can be particularly problematic for critical conversations that require confidentiality, such as business negotiations or discussions of government affairs. Creating such secure protocols is not easy with vulnerabilities being found in established applications [6–8].

This is where Sharekey, a new player in the collaboration app domain, comes in. Their collaboration app offers a combination of messaging, file sharing, audio, and video calls with strong claims on the security and privacy of the data shared through their app [9, 10]. As with any other app, it is important to subject Sharekey to a critical review to ensure that these claims hold up under scrutiny. If problems are found, they should be fixed promptly and transparently to maintain user trust and confidence in the app's security.

1.1 Rationale

As a relatively new platform, Sharekey is still in the process of establishing its position as a trustworthy app with high-security standards. To ensure future users can rely on the platform's security, the strong security claims made by Sharekey must be thoroughly evaluated for their validity. Being an entity tasked with evaluating the cyber risks and opportunities presented by emerging technologies, the Cyber-Defence Campus (CYD) is interested in conducting a thorough examination of Sharekey's security features, especially given that the Swiss-based startup offers a promising solution in the security sector that is worthy of consideration.

Sharekey is designed to provide a high degree of security and flexibility in communication among subjects who require confidentiality, such as high-level business executives or government officials. However, past vulnerabilities of established players in the industry, such as Threema, Mega, or Telegram [6–8], combined with the increased risk of targeted attacks on C-level staff due to their level of access to sensitive information and decision-making authority [11], make it imperative to review Sharekey's security claims thoroughly.

1.2 Related Work

Sharekey is a relatively young protocol, which has not yet been extensively researched in academia, though a security review of Sharekey was conducted by Aumasson [12] in July 2022. The review focused on the verification of the following security properties:

- All data must be stored in Switzerland.
- A user key should never leave their device.
- Sharekey's servers should be unable to decrypt any stored messages.
- User files and folders must be undecryptable by Sharekey.
- A link generated for a file should allow the receiver(s) to decrypt the file, but not Sharekey.

The review assumes two kinds of adversaries, starting with the "honest-but-curios" model for adversaries in Sharekey. In particular, this assumes, that Sharekey provides users with applications built from the reviewed code and that Sharekey themselves is also executing the reviewed code on their side. Adversaries at Sharekey are only able to observe traffic, without modifying it to gain knowledge. The other adversary the review considered were the users of the app. They were modeled as active adversaries, allowing them to deviate from the given protocol.

In this setting, the reviewers concluded that Sharekey provides reasonable assurance that it cannot access users' data, and none of the identified issues contradicted this claim. However, the report did find several issues, all of which were classified as having low severity:

1. The application's cache stored references to URLs sent in messages in cleartext.
2. Sharekey had an account existence oracle that allowed an attacker to determine whether a given email address had a Sharekey account.

3. Public keys received from third parties, such as users, were not validated, which could potentially allow them to be set to invalid values such as zero or points, not on the curve.
4. The integrity of encrypted messages was checked right after encryption by attempting to decrypt them. Through the library used, the Message Authentication Code (MAC) tag of the message is also verified in the same step. The review argued that the integrity of encrypted messages should not be checked immediately after encryption, but rather at a later stage of the transmission process, where the message was more likely to be corrupted. It additionally argues, that the signature of messages and timestamps was not checked when verifying the message integrity, even though signature verification would be sufficient to verify message integrity.

The review argues that Sharekey has to trade off the security of its protocol with the usability of its app, which can impose certain limitations on the theoretical security guarantees. Specifically, Sharekey's design requires users to log in and restore the client's full state on a new device using their long-term secret, as well as using the application on multiple devices concurrently. The report argues that this limits the theoretical security guarantees for risk mitigation in case of compromise of a user's device. Nonetheless, the reviewers consider this compromise acceptable and consider Sharekey to be significantly safer than most collaboration platforms.

1.3 Structure and Contributions

The purpose of this thesis is to investigate the security claims made by Sharekey. While some of their claims have been validated by a prior cryptographic audit [12], this work performs a more in-depth review of the cryptographic properties of their protocols.

We start by providing an overview of the Sharekey app in Chapter 2. We first introduce the features and front-end view of the app. This is followed up with the security claims of Sharekey and finally documentation of the protocol that Sharekey uses.

Then we continue describing our review of the Sharekey protocol in Chapter 3. We describe how we implemented our client for the Sharekey Protocol to perform manual and automated tests on the protocol, followed by a list of 19 security findings that we discovered in our review process. We then discuss the severity of the findings, how they are being remediated by Sharekey, and discuss the validity of Sharekey's security and privacy claims in light of our findings.

This is followed up in Chapter 4 by threat modeling for messaging and file sharing in Sharekey, where we thoroughly evaluate several attacker models distinguished by the different assets they reside in. We describe several security properties and argue which adversaries manage to break them. We discuss the limitations of our adversarial modeling that arise from the difficulty of guaranteeing the validity of the application delivered to the end user. Finally, we discuss the trade-off between usability and security and explain certain design decisions by Sharekey that lead to better usability while compromising on some security properties.

Our last contribution is presented in Chapter 5, where we describe our process of designing an updated version of the file-sharing protocol to improve file integrity. We describe the suggested protocol and support our claims through modeling and symbolic verification with Tamarin Prover [13–15]. We discuss our findings and suggest some directions to continue the design process of the new protocol. Lastly, we contemplate more generally three possible strategies that Sharekey could follow in further development of their protocols giving our opinion on the advantages and disadvantages of each.

We conclude in Chapter 6, where we reflect and summarize our results and present our main takeaways.

1.4 Involved Parties

This thesis is the result of a collaboration between several organizations, all of which have played an important role in ensuring the success of this project. This section describes each of the organizations and their roles and contributions to this work.

1.4.1 Applied Cryptography Group

The Applied Cryptography Group at ETH Zürich provided invaluable support to this project. Guided by Professor Kenny Paterson, their supervision and mentorship were important to the successful execution of this thesis.

Their research focuses on Applied Cryptography, working with standards bodies, such as the Internet Engineering Task Force (IETF) [16] and the Internet Research Task Force (IRTF) [17] to lead the development of new cryptographic standards for the Internet, such as their work on the TLSv1.3 standard [18–20]. Moreover, the group's expertise extends to auditing cryptographic systems deployed in the industry. In their capacity as responsible security researchers, they collaborate with vendors to remediate security issues that are uncovered during their analysis. They have found security issues in projects such as Threema, Mega, or Telegram [6–8].

1.4.2 Cyber-Defence Campus

This specific project was suggested and co-supervised by the CYD. The CYD acted as a mediator between the involved parties.

The CYD was founded in 2019 to foster collaboration between the Swiss Federal Department of Defence (DDPS), industry, and academia in research, development, and training for cyber defence. Its mission is to build a network in the field of cyber-defence, identify emerging trends early, undertake research and innovation in cyber technologies, and train cyber specialists.

1.4.3 Sharekey

Sharekey, a Switzerland-based startup and collaboration suite, was integral to this thesis. Sharekey granted access to their product and provided full transparency to all aspects of their product (e.g., source code, roadmap, development infrastructure). Their support enabled a thorough examination of their collaboration app and served as a critical component of this thesis.

The Sharekey collaboration suite offers encrypted services to a customer base of 5,000 board members and C-suite executives throughout Europe[1]. The application has been subject to a previous security audit [12]. Sharekey claims to employ App-To-App encryption technology that provides users with a private connection similar to a Virtual Private Network (VPN) that secures the transmission of messages between users and their contacts [21]. They position themselves as an alternative to other messaging and collaboration platforms such as Teams, Slack, Dropbox, and WhatsApp, which Sharekey claims to be known for tracking user activity and monetizing corporate data [21]. According to Sharekey, they operate under Swiss law and are therefore not subject to US jurisdiction, especially the CLOUD Act [9, 21, 22]. The application is marketed as a secure and private platform that protects user data from unauthorized access, third-party data mining, or breaches.

[1]as stated by Sharekey upon request on 2023-01-10

Chapter 2

Sharekey Collaboration App

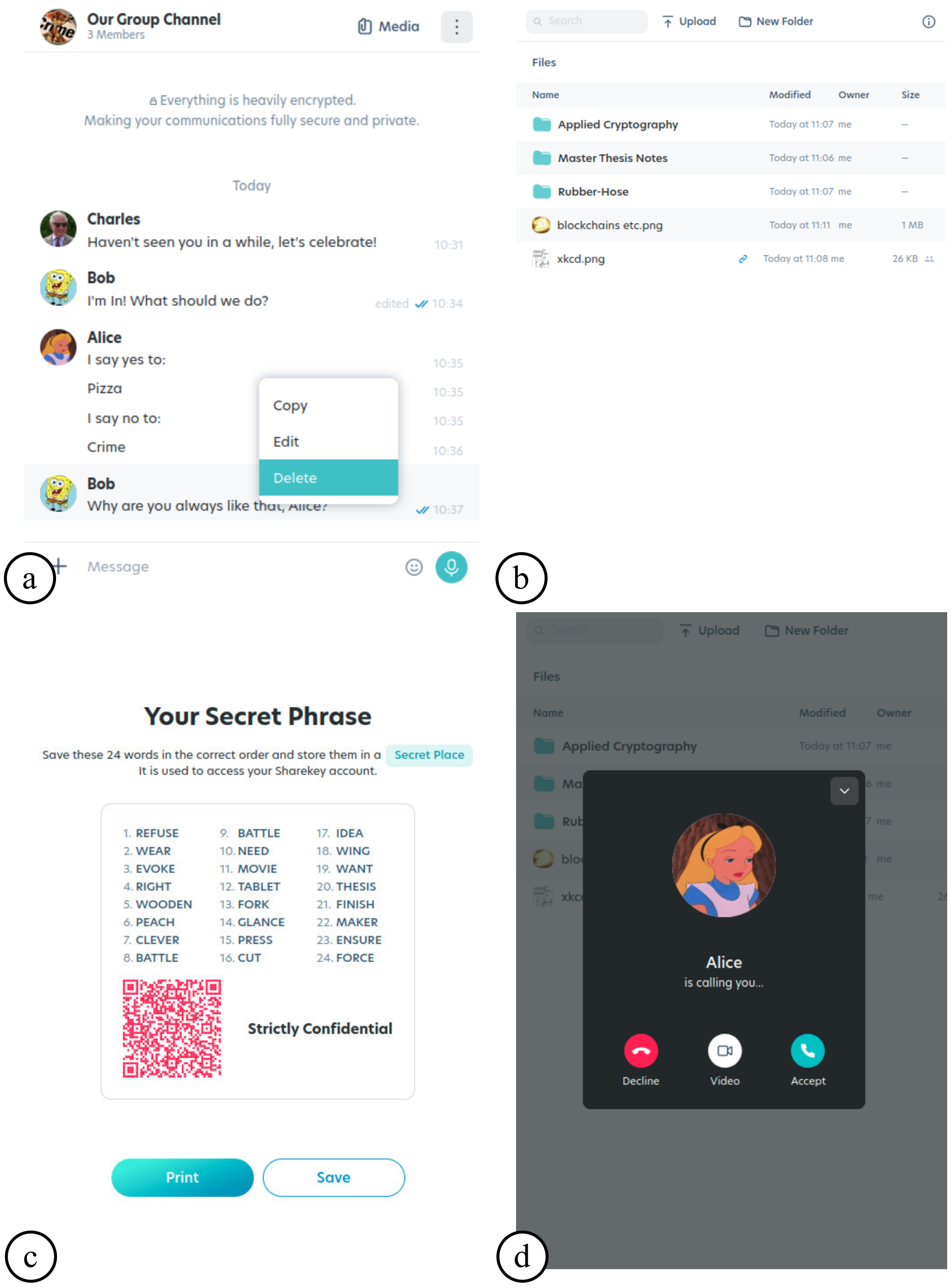
This chapter provides an overview of the Sharekey app. The app allows users to securely share files and messages with others. Sharekey claims to prioritize the security and privacy of its users, with features such as data storage in Switzerland, user keys that never leave the device, and messages and files that are undecryptable for Sharekey. The protocol used by Sharekey includes several algorithms that we extracted from the source code as part of our analysis and typeset in LaTeX to be able to check them without having to dive into the source code. These algorithms will be explained in detail in Section 2.3.

2.1 Features

The main features of the Sharekey app are messaging, file-sharing, and audio and video calls (beta). Furthermore, a fourth main feature is that in contrast to other messaging apps (e.g., Signal or Threema), Sharekey has native multi-device support. To give an idea of what these features look like from a Sharekey user's perspective, Figure 2.1 shows an example screenshot for each of these features.

At the core of the application lies its messaging protocol, allowing participants to engage in instant messaging (compare Figure 2.1a) Similar to other popular messaging platforms, such as WhatsApp, Signal, Telegram, or Threema, Sharekey's messaging feature permits direct chats with two members or group chats with one or more members. Additionally, the senders of messages can edit and delete their messages.

Sharekey's file-sharing feature extends beyond its messaging functionality, resembling established cloud-based services like Dropbox and Nextcloud (compare Figure 2.1b) Users can store files and create folder structures, which can be shared within Sharekey channels. Furthermore, Sharekey of-



Our Group Channel
3 Members

🔒 Everything is heavily encrypted.
Making your communications fully secure and private.

Today

Charles
Haven't seen you in a while, let's celebrate! 10:31

Bob
I'm In! What should we do? edited ✓ 10:34

Alice
I say yes to: 10:35
Pizza 10:35
I say no to: 10:35
Crime 10:36

Copy
Edit
Delete

Bob
Why are you always like that, Alice? ✓ 10:37

Files

Name	Modified	Owner	Size
Applied Cryptography	Today at 11:07	me	–
Master Thesis Notes	Today at 11:06	me	–
Rubber-Hose	Today at 11:07	me	–
blockchains etc.png	Today at 11:11	me	1 MB
xkcd.png	Today at 11:08	me	26 KB

Your Secret Phrase

Save these 24 words in the correct order and store them in a Secret Place
It is used to access your Sharekey account.

1. REFUSE	9. BATTLE	17. IDEA
2. WEAR	10. NEED	18. WING
3. EVOKE	11. MOVIE	19. WANT
4. RIGHT	12. TABLET	20. THESIS
5. WOODEN	13. FORK	21. FINISH
6. PEACH	14. GLANCE	22. MAKER
7. CLEVER	15. PRESS	23. ENSURE
8. BATTLE	16. CUT	24. FORCE

Strictly Confidential

Alice
is calling you...

Figure 2.1: Screenshots from the main functionalities of the Sharekey application. The screenshots provide an overview of the main features of the Sharekey app, containing (**a**) messaging in group channel and deleting a message, (**b**) personal files and folders, (**c**) secret phrase, which is used to log in from a new device, and (**d**) audio and video calls as a beta feature.

fers external sharing capabilities through access links, allowing people without Sharekey accounts to access shared files.

The app also includes a beta version of a call feature, allowing users to engage in audio and video calls (compare Figure 2.1d) As this feature was still in the early stages of development during the period of the thesis, only a brief overview of this functionality was possible.

Finally, they offer multi-device compatibility, allowing users to access their accounts and data on several devices (compare Figure 2.1c) The application provides users with a personal passphrase comprising 24 English words, which serves as an encryption key to access all encrypted data within Sharekey. The application provides users with the ability to recover the state of their data on any new device and remain synchronized across all connected devices. In Sharekey, a user's identity is only linked to their personal secret key, which means that their account can be accessed on multiple devices without the need for workarounds, such as Signal's approach which requires the creation of a separate participant for each device used by a user [23]. Sharekey's protocol does not distinguish between different devices used by the same user.

2.2 Security Claims

Sharekey makes several claims regarding the functionality and security of its encrypted messaging application, which can be found in their security paper [10] and their privacy policy [9]. While the privacy policy is freely accessible on the Sharekey website, the security paper can only be accessed after providing an email address. We will outline the main claims of the two documents separately in the following subsections, as they are not identical. Specifically, the privacy policy, which was created in July 2020, contains stronger promises than the security paper created in March 2021.

2.2.1 Privacy Policy

In their privacy policy from June 2020 [9], Sharekey makes several claims about the privacy and security of their application. The privacy policy starts with the following summary, presented here verbatim and from the perspective of Sharekey.

1	Privacy by Default	Your privacy is very important to us.
2	Your Data. Your Asset.	Your data belong to you, not to Sharekey or anyone else.

3	Zero-Knowledge Encryption	Your data is encrypted on your device, we do not hold the decryption keys.
4	No Access	Only you can access your data. Sharekey has no access to them.
5	No Backdoor	Our software contains no backdoor.
6	No Metadata	Your metadata is also encrypted on your device with your encryption keys.
7	No Social Graph	We cannot build any social relationship between Users, so-called Social Network or Social Graph.
8	No Ads	Your data or the content of your communications and storage is never sold or rented to anyone, it will never be used for any third-party advertising.
9	No Cookies	No tracking and no cookies on our website.
10	Limited Technical Data	So-called "log files" are kept for a maximum of 7 days, and only to facilitate troubleshooting, improve the service and prevent misuse.
11	Transparency	We are transparent about the data we collect and what we do and don't do with it.
12	No CLOUD Act [24]	We follow Swiss and European privacy laws.

2.2.2 Security Paper

In contrast to their privacy policy, Sharekey's security paper from March 2021 [10] offers a more detailed technical analysis of Sharekey's current capabilities, along with a roadmap with their plans for future development. This section provides a summary of the key claims presented in the security paper, focusing on those most relevant to this thesis. We are paraphrasing their Overview and Security Design chapters while maintaining the original meaning as closely as possible. We make use of the same technical terminology as used in the security paper, even if it is not commonly used in other contexts.

13	App-to-App Encryption	Sharekey employs App-to-App Encryption for all data exchanges, which differs from end-to-end encryption where some parts of the exchanged information, typically the metadata, is sent in clear and read by intermediaries. With Sharekey only minimal metadata is visible to Sharekey, which does not compromise user privacy in any way.
14	Local Encryption	Sharkey ensures that all data supported by the platform is encrypted on the user's device, except for downloaded files opened in external applications. An attacker with physical device access must know the user's password or secret phrase to access this data.
15	Cryptographic Properties	Sharekey's crypto layer implements standardized data encryption, authentication, and integrity verification for all Sharekey functions. This allows for a privacy-focused collaboration platform including encrypted cloud-based storage that is both secure and easy to use.
16	Encrypted Backup	Sharekey stores a complete history of a user's communications and stored files and folders in encrypted form to restore a user's data on a new device.
17	Forward Secrecy	Sharekey allows users to access their conversations on different devices, allows to recover old messages on new devices, and ensures synchronization between all devices. To achieve this, the shared key used in Channels is not ratcheted, and their protocol does not provide forward secrecy.

18	Metadata	Sharekey requires the absolute minimum metadata on its users, just enough to be able to offer the necessary features. They receive and process less metadata than similar communication apps and plan to further reduce the amount of metadata as much as they can. For channels, they store the owner of that channel, the name of the channel, and a pseudonymized list of its members. For files and folders, they store the owner of the object, the list of users it is shared with, and the role of each user.

2.3 Current Protocol

This section describes the protocol that was used by Sharekey as of the start of this thesis in October 2022. While Sharekey is planning on making their client open source, the clients' JavaScript source is not yet available except in minified form as delivered to their web clients. The native applications also rely on the same JavaScript as delivered to the web clients. Therefore we decided to translate the parts of their clients that are most relevant for this thesis into LaTeX algorithms to explain how they work and as a reference when explaining our findings.

2.3.1 Notation

We chose the symbol names as descriptive as possible while still keeping them short and therefore use some abbreviations. Here we introduce the notation Table 2.1 provides an overview and a short explanation of the symbols used throughout the algorithms introduced in this section.

We use superscripts to signify ownership and subscripts to describe the contents. For example, with $ctxt^{user}_{ptxt}$ we describe the cipher text of a plain text $ptxt$, with keys owned by $user$. These relations can also be understood from the algorithms the symbols are used in.

For some algorithms, we make use of string constants, for example, to index elements in the database or as constants. Such strings will be surrounded by quotation marks *"·"*, for example: *"foo bar"*.

2.3.2 Encryption and Signatures

Sharekey uses the NaCL cryptographic library [25] for its cryptographic primitives. NaCl is designed to provide high-level cryptographic primi-

Symbol	**Description**
$ctxt$	ciphertext
$ctxt_{obj}$	ciphertext of object
n_{obj}	nonce / number used once
enc_{obj}	The pair $(ctxt_{obj}, n_{obj})$
$encsig_{obj}$	The tuple $(ctxt_{obj}, n_{obj}, sig_{ctxt_{obj}})$
$ptxt$	plaintext
sk	secret key
ssk	shared secret key
sk_{enc}	secret key used for (symmetric) authenticated encryption
sk_{sig}	secret key used for (asymmetric) signatures
pk_{sig}	public key used to verify (asymmetric) signatures
sig_{obj}	signature of some data
sk_{priv}	private key of a user used for (symmetric) encryption
sk_{DH}	secret key of a user for (asymmetric) key exchanges
pk_{DH}	public key of a user for (asymmetric) key exchanges
sk_{phrase}	secret passphrase of user
h_{obj}	a SHA256 hash of obj
$ctxtsig_{obj}$	encrypted then signed object, containing ciphertext, nonce and signature
pwd	password used to derive the key to encrypt either the keys stored in local storage or to encrypt the encryption keys of files shared with external links
msg	plaintext message sent through a Sharekey channel
ts	timestamp

Table 2.1: Symbol names and description for Sharekey algorithms.

tives that are easy to use and provide strong security. The library provides one choice for each primitive, which is selected to provide high security and protection against timing attacks, in contrast to alternative libraries such as OpenSSL which provide users with many options of cryptographic primitives, which may not provide any security at all in some combinations. Sharekey utilizes TweetNaclJS, a JavaScript implementation of Bernstein's TweetNaCl library, which Sharekey adapts by facade methods that offer a simplified API to the underlying algorithms.

Encryption (see Algorithm 1) uses authenticated encryption that combines the *XSalsa20* stream cipher [26], with a *Poly1305* MAC [27] over the cipher text. The nonces are of length 192 bits and are randomly chosen through the randomness exposed by the TweetNaCl.js library.

Algorithm 1 Encryption

```
procedure ENCRYPT($ptxt$, $sk_{enc}$)
    // using nacl.randomBytes
    $n_{ptxt} \leftarrow_{\$} \{0,1\}^{192}$
    // using nacl.secretbox
    $ctxt_{ptxt} \leftarrow$ xsalsa20-poly1305-encrypt$(ptxt, n, sk_{enc})$        ▷ NaCl
    return $(ctxt_{ptxt}, n)$
end procedure
```

Decryption (see Algorithm 2) decrypts ciphertexts produced by Algorithm 1 after performing a check on the MAC tag of the message. If the MAC verification fails, the function will also fail and raise an error.

Algorithm 2 Decryption

```
procedure DECRYPT($ctxt_{ptxt}$, $n_{ptxt}$, $sk_{enc}$)
    // using nacl.secretbox.open
    $ptxt \leftarrow$ xsalsa20-poly1305-decrypt$(ctxt_{ptxt}, n_{ptxt}, sk_{enc})$
    return $ptxt$
end procedure
```

Messages are signed with the *Ed25519* [28] signature scheme (see Algorithm 3 for the signature and Algorithm 4 for its verification).

Algorithm 3 Signatures

```
procedure SIGN($data$, $sk_{sig}$)
    // using nacl.sign.detached
    $sig \leftarrow$ Ed25519-sign$(data, sk_{sig})$
    return $sig$
end procedure
```

Algorithm 4 Signature Verification

```
procedure VERIFY-SIGNATURE($data$, $sig$, $pk_{sig}$)
    // using nacl.sign.detached.verify
    $valid \leftarrow$ Ed25519-verify$(data, sig, pk_{sig})$
    if not $valid$ then
        raise Exception
    end if
end procedure
```

Many times encryption and signature are executed simultaneously, which therefore legitimizes a separate encrypt-then-sign function (see Algorithm 5). This function first invokes encryption on the message to produce the ciphertext $ctxt_{ptxt}$, followed by a signature of only the $ctxt_{ptxt}$, not including the nonce n_{ptxt}.

Algorithm 5 Encrypt-then-sign

```
procedure ENCRYPT-THEN-SIGN($ptxt$, $sk_{enc}$, $sk_{sig}$)
    $(ctxt_{ptxt}, n_{ptxt}) \leftarrow ENCRYPT(ptxt, sk_{enc})$
    $sig_{ctxt_{ptxt}} \leftarrow SIGN(ctxt_{ptxt}, sk_{sig})$
    $encsig_{ptxt} \leftarrow (ctxt_{ptxt}, n_{ptxt}, sig_{ctxt_{ptxt}})$
    return $encsig_{ptxt}$
end procedure
```

Algorithm 6 describes Sharekey's implementation for generating uniformly distributed random numbers within a range $\{0, \ldots, limit - 1\}$. Such a function is necessary to ensure the generation of output values that are uniformly distributed for ranges whose sizes are not powers of two. Their implementation disfavors the output 0. If we call the number of possible wrap-around $wraps = \lfloor \frac{2^{31}-1}{limit} \rfloor$, then the probability of the output being 0 is

$P[RANDOM\text{-}NUMBER\text{-}BELOW(limit) = 0] = \frac{wraps-1}{wraps \cdot limit - 1}$,

whereas the probability of the output being anything else is bigger with

$P[RANDOM\text{-}NUMBER\text{-}BELOW(limit) \neq 0] = \frac{wraps}{wraps \cdot limit - 1}$.

However, this function is currently only used to generate the random passphrase, where every word is randomly chosen from a dictionary of 1700 words. Therefore the only limit used for this function is $limit = 1700$, so the probability that the output is 0 is less than $\frac{1}{2 \cdot 10^9}$ smaller than the probability for any other value.

Algorithm 6 Random number generation

```
procedure RANDOM-NUMBER-BELOW($limit$)
    $rand \leftarrow 0$
    while $rand = 0$ or $\frac{rand}{limit} > \lfloor \frac{2^{31}-1}{limit} \rfloor$ do
        // using window.crypto.getRandomValues:
        $rand \leftarrow_{\$} \{0,1\}^{32}$
    end while
    return $rand$ mod $limit$
end procedure
```

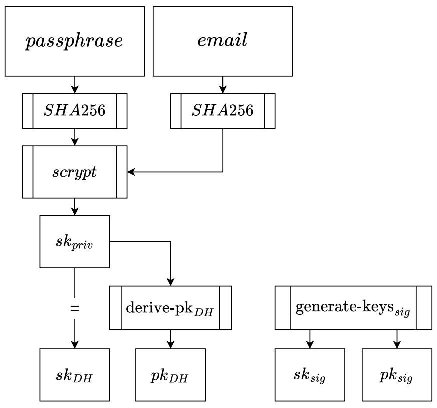


Figure 2.2: Scheme describing the derivation of personal keys from the passphrase and email address.

2.3.3 User Key generation

When signing up for a new account the participant generates a secret phrase with 24 words, sampled from a dictionary with 1700 English words, which provides around 256 bits of security. As shown in Figure 2.2 and Algorithm 7, the user derives their private key sk_{priv} from their secret phrase and their e-mail address using the scrypt key derivation function [29], specifically the scrypt.js library [30]. As seen in Algorithm 8, the Diffie-Hellman (DH) secret is set to $sk_{DH} = sk_{priv}$ and the corresponding pk_{DH} is derived using the NaCl library. sk_{sig} is randomly sampled and the corresponding pk_{sig}) derived from it using NaCl. Finally, the keys are encrypted and signed with sk_{priv} and sk_{sig} and uploaded to the Sharekey server as seen in Algorithm 9. The communication between client and server is secured with the Transport Layer Security (TLS) protocol, version 1.2 or 1.3 [31].

Algorithm 7 Personal private key derivation

```
procedure DERIVE-PRIVATE-KEY($sk_{phrase}, email$)
    $h_{sk_{phrase}} \leftarrow SHA256(sk_{phrase})$
    $salt \leftarrow SHA256(email)$
    $cost_{cpu} \leftarrow 2^{15}$
    $cost_{memory} \leftarrow 4$
    $cost_{parallel} \leftarrow 1$
    $length \leftarrow 32$
    return $SCRYPT(h_{sk_{phrase}}, salt, cost_{cpu}, cost_{memory}, cost_{parallel}, length)$
end procedure
```

Algorithm 8 Personal key material generation

```
procedure GENERATE-PERSONAL-KEYS($email$)
    $dictionary \leftarrow$ 1700 English words
    $p_1 \leftarrow dictionary[RANDOM\text{-}NUMBER\text{-}BELOW(1700)]$
    $p_2 \leftarrow dictionary[RANDOM\text{-}NUMBER\text{-}BELOW(1700)]$
    $\vdots$
    $p_{24} \leftarrow dictionary[RANDOM\text{-}NUMBER\text{-}BELOW(1700)]$
    $sk_{phrase} \leftarrow [p_1, p_2, \ldots, p_{24}]$
    $sk_{priv} \leftarrow DERIVE\text{-}PRIVATE\text{-}KEY(sk_{phrase}, email)$
    $sk_{DH} \leftarrow sk_{priv}$
    // using nacl.box.keyPair.fromSecretKey
    $pk_{DH} \leftarrow derive\text{-}pk_{DH}(sk_{DH})$
    // using nacl.sign.keyPair
    $(sk_{sig}, pk_{sig}) \leftarrow generate\text{-}keys_{sig}()$
    return $(sk_{phrase}, email, sk_{priv}, sk_{DH}, pk_{DH}, sk_{sig}, pk_{sig})$
end procedure
```

Algorithm 9 Personal keys storage

```
procedure UPLOAD-PERSONAL-KEYS($email, sk_{priv}, pk_{DH}, sk_{sig}, pk_{sig}$)
    $encsig_{sk_{sig}} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(sk_{sig}, sk_{priv}, sk_{sig})$
    $encsig_{email} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(email, sk_{priv}, sk_{sig})$
    upload$(SHA256(email), encsig_{email}, encsig_{sk_{sig}}, pk_{sig}, pk_{DH})$
end procedure
```

When logging in to a Sharekey account from a new device, the client retrieves the encrypted secret keys and unencrypted public keys of that user. While the user's public key could also be derived from their secret keys, they are set to the values obtained from the Sharekey server.

Algorithm 10 Personal keys retrieval

```
procedure DOWNLOAD-PERSONAL-KEYS($sk_{phrase}, email$)
    $(h_{email}, encsig_{email}, encsig_{sk_{sig}}, pk_{sig}, pk_{DH}) \leftarrow$ download$(SHA256(email))$
    $sk_{priv} \leftarrow DERIVE\text{-}PRIVATE\text{-}KEY(sk_{phrase}, email)$
    $(ctxt_{sk_{sig}}, n_{sk_{sig}}, sig_{ctxt_{sk_{sig}}}) \leftarrow ctxt_{sk_{sig}}$
    $sk_{sig} \leftarrow DECRYPT(ctxt_{sk_{sig}}, n_{sk_{sig}}, sk_{priv})$
    $sk_{DH} \leftarrow sk_{priv}$
    return $(sk_{phrase}, email, sk_{priv}, sk_{DH}, pk_{DH}, sk_{sig}, pk_{sig})$
end procedure
```

2.3.4 Key Storage

As a means of comfort, users don't have to enter this sk_{phrase} every time they visit the web application. This is done by encrypting all secret key material with a user-chosen password, as shown in Algorithm 11. The key material can then be read and decrypted from local storage again with the knowledge of the password as shown in Algorithm 12. Users can also choose to log in without a password, in which case the key material is stored unencrypted in the local storage. On mobile applications, the keys are stored in the keychain of the device, enabling unlocking with the available means, including faceID or fingerprint, which was not further investigated, as this thesis focused on the web client.

Algorithm 11 Store personal key material in local storage

```
procedure STORE-PERSONAL-KEYS($pwd$, $email$, $pk_{DH}$, $sk_{DH}$, $pk_{sig}$, $sk_{sig}$, $sk_{phrase}$)
    $keys \leftarrow (pk_{DH}, sk_{DH}, pk_{sig}, sk_{sig}, sk_{phrase})$
    $sk_{pwd} \leftarrow$ DERIVE-PRIVATE-KEY$(pwd, email)$
    $(cipher_{keys}, n_{keys}) \leftarrow ENCRYPT(keys, sk_{pwd})$
    $save\text{-}locally((cipher_{keys}, n_{keys}))$
end procedure
```

Algorithm 12 Load personal key material in local storage

```
procedure LOAD-PERSONAL-KEYS($pwd, email$)
    $(cipher_{keys}, n_{keys}) \leftarrow LOAD\text{-}LOCALLY()$
    $sk_{pwd} \leftarrow$ DERIVE-PRIVATE-KEY$(pwd, email)$
    $keys \leftarrow DECRYPT(cipher_{keys}, n_{keys}, sk_{pwd})$
    $(pk_{DH}, sk_{DH}, pk_{sig}, sk_{sig}, sk_{phrase}) \leftarrow keys$
    return $(pk_{DH}, sk_{DH}, pk_{sig}, sk_{sig}, sk_{phrase})$
end procedure
```

2.3.5 Channel Creation

Sharekey channels are where members can message each other. There is at most one direct channel between each pair of participants and unlimited group channels between an arbitrary number of participants. Every channel has a shared secret key *ssk*, which is known by all participants and derived as seen in Subsection 2.3.6. This is implemented as follows as described by Algorithm 13 for direct channels and Algorithm 14 for group channels. We differentiate between the creator and other participants through the superscript $\cdot^{c}$ for the channel creator, $\cdot^{p}$ for the second participant in a direct channel, and $\cdot^{p_i}$ for the i-th participant in a group channel. The creator of a direct channel is whoever initiates the direct channel. Channels also have some additional metadata, mainly the group avatar image, which we abstracted as *metadata* here. The metadata is processed in *process-metadata*, which crops the avatar, uploads it, and obtains the Uniform Resource Locator (URL) of the uploaded image.

Algorithm 13 Creation of direct channel

```
**procedure** CREATE-DIRECT($metadata$, $sk^c_{DH}$, $pk^c_{DH}$, $sk^c_{sig}$, $pk^c_{sig}$, $pk^p_{DH}$, $pk^p_{sig}$)
    // Derive shared key
    $ssk \leftarrow DERIVE\text{-}SHARED\text{-}KEY(pk^p_{DH}, sk^c_{DH})$

    // Encrypt shared key for creator
    $encsig^c_{ssk} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(ssk, sk^c_{priv}, sk^c_{sig})$
    $keys^c \leftarrow (pk^c_{DH}, pk^c_{sig}, encsig^c_{ssk})$

    // Encrypt shared key for participant
    $encsig^p_{ssk} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(ssk, ssk, sk^c_{sig})$
    $keys^p \leftarrow (pk^p_{DH}, pk^p_{sig}, encsig^p_{ssk})$

    // Update metadata and upload
    $metadata' \leftarrow process\text{-}metadata(metadata)$
    **upload**$(metadata', \{keys^c, keys^p\})$
**end procedure**
```

Algorithm 14 Creation of group channel

```
procedure CREATE-GROUP($name, description, metadata, (sk^c_{DH}, pk^c_{DH}, sk^c_{sig}, pk^c_{sig}), \{(pk^{p_1}_{DH}, pk^{p_1}_{sig}), \ldots, (pk^{p_n}_{DH}, pk^{p_n}_{sig})\}$)
    // Encrypt shared key for creator
    $ssk \leftarrow GENERATE\text{-}GROUP\text{-}KEY()$
    $encsig^c_{ssk} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(ssk, sk^c_{priv}, sk^c_{sig})$
    $keys^c \leftarrow (pk^c_{DH}, pk^c_{sig}, encsig^c_{ssk})$

    // Encrypt shared key for participants
    for all $pk^i_{DH} \in \{pk^{p_1}_{DH}, \ldots, pk^{p_n}_{DH}\}$ do
        $ssk^{p_i}_{shared} \leftarrow DERIVE\text{-}SHARED\text{-}KEY(pk^{p_i}_{DH}, sk^c_{DH})$        ▷ Cached
        $encsig^{p_i}_{ssk} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(ssk, ssk^{p_i}_{shared}, sk^c_{sig})$
        $keys^{p_i} \leftarrow (pk^{p_i}_{DH}, pk^{p_i}_{sig}, encsig^{p_i}_{ssk})$
    end for

    // Encrypt data
    $encsig_{name} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(name, ssk, sk^c_{sig})$
    $encsig_{desc} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(description, ssk, sk^c_{sig})$
    $encsig_{time} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(String(now()), ssk, sk^c_{sig})$

    // Upload avatar, obtain URL
    $metadata' \leftarrow process\text{-}metadata(metadata)$
    upload($(encsig_{name}, encsig_{desc}, encsig_{time}, metadata', \{keys^c, keys^0, \ldots, keys^n\})$)
end procedure
```

2.3.6 Channel Key generation

Sharekey offers direct channels for communication between two participants and group channels for communication between any number of participants. The communication within these channels is encrypted using a static shared secret key *ssk*, which is known by all participants. As seen in Algorithms 15 and 16, key derivation differs between direct channels, where keys are derived from the public and private DH keys of the participants, using *x25519*, a DH key exchange based on curve *Curve25519* [32]. The *x25519* key exchange allows to derive keys offline from pk^B_{DH} and sk^A_{DH} and is symmetric, meaning that $x25519(pk^B_{DH}, sk^A_{DH}) = x25519(pk^A_{DH}, sk^B_{DH})$. On the other hand, group channels simply use a randomly generated secret key, which is then distributed to all participants. Similar to Subsection 2.3.5, the superscript $\cdot^A$ and $\cdot^B$ describe ownership by A or B respectively. We use A and B instead of c and p, as we do not know and care which of A and B is the creator of that channel.

Algorithm 15 Key derivation for direct channels

```
procedure DERIVE-SHARED-KEY($pk^B_{DH}$, $sk^A_{DH}$)
    $ssk \leftarrow x25519(pk^B_{DH}, sk^A_{DH})$
    return $ssk$
end procedure
```

Algorithm 16 Key derivation for group channels

```
procedure GENERATE-GROUP-KEY
    // using window.crypto.getRandomValues
    $ssk \leftarrow_\$ \{0,1\}^{256}$
    return $ssk$
end procedure
```

2.3.7 Messages

Messages are encrypted and signed using the shared key *ssk* of the communication channel and the sender's secret signature key sk_{sig}. The message timestamp is set by the sender and also encrypted and signed separately. The receiver obtains an encrypted message and timestamp, which they then decrypt. Signature verifications were not performed. If decryption fails, the message content is changed to the text "Message decryption failed" and the message timestamp is set to 0, which displays as a usual message which would have been sent on January 1st, 1970 with the content "Message decryption failed". The details behind sending a message can be found in Algorithm 17, while Algorithm 18 describes how encrypted messages retrieved by Sharekey are decrypted.

Algorithm 17 Send Message

```
procedure MESSAGE-SEND($msg$, $ts$, $ssk$, $sk_{sig}$)
    $encsig_{msg} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(msg, ssk, sk_{sig})$
    $encsig_{ts} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(ts, ssk, sk_{sig})$
    upload($encsig_{msg}, encsig_{ts}$)
end procedure
```

Algorithm 18 Receive Message

```
procedure MESSAGE-DECRYPT($encsig_{msg}$, $encsig_{ts}$)
    $(ctxt_{msg}, n_{msg}, sig_{msg}) \leftarrow encsig_{msg}$
    $(ctxt_{ts}, n_{ts}, sig_{ts}) \leftarrow encsig_{ts}$
    $msg \leftarrow DECRYPT(ctxt_{msg}, n_{msg}, ssk)$
    $ts \leftarrow DECRYPT(ctxt_{ts}, n_{ts}, ssk)$
     return $(msg, ts)$
end procedure
```

2.3.8 Loading a channel

When loading a channel, the participant can obtain the encrypted channel information from the server and decrypt the shared secret key *ssk*, which can later be used to encrypt and decrypt messages. While the derivation of the shared key is symmetric and it would therefore be possible to obtain the secret key for a direct channel without loading it from the server, Sharekey decided to load the encrypted key from the server. This would in theory also allow for changing the key at some point, which is however not implemented yet. This algorithm populates and uses $cache_{pk}$, which caches the public keys of other users they are messaging.

This is described in Algorithm 19 for direct channels and in Algorithm 20 for group channels. Similar to Subsection 2.3.5, the superscript $\cdot^{A}$ and $\cdot^{B}$ describe ownership by A or B respectively.

Algorithm 19 Obtain secret key of direct channel, from the perspective of A

```
procedure LOAD-DIRECT($id$, $sk^A_{DH}$, $cache_{pk}$)
    $(metadata', \{keys^A, keys^B\}) \leftarrow \textbf{download}(id)$

    // Add public keys of B to known keys if they have not been added yet
    if $B \in cache_{pk}$ then
        $(pk^B_{DH}, pk^B_{sig}) \leftarrow cache_{pk}[B]$
    else
        $(pk^B_{DH}, pk^B_{sig}, encsig^B_{ssk}) \leftarrow keys^B$
        $cache_{pk}[B] \leftarrow (pk^B_{DH}, pk^B_{sig})$
    end if

    // Obtain encryption key
    if $A$ is creator then
        $sk_{enc} \leftarrow sk^A_{priv}$
    else
        $sk_{enc} \leftarrow DERIVE\text{-}SHARED\text{-}KEY(pk^B_{DH}, sk^A_{DH})$
    end if

    // Decrypt and return shared secret key ssk
    $(pk^A_{DH}, pk^A_{sig}, encsig_{ssk}) \leftarrow keys^A$
    $(ctxt_{ssk}, n_{ssk}, sig_{ctxt_{ssk}}) \leftarrow encsig^A_{ssk}$
    $ssk \leftarrow DECRYPT(ctxt_{ssk}, n_{ssk}, sk_{enc})$
    return $ssk$
end procedure
```

Algorithm 20 Obtain secret key of group channel

```
procedure LOAD-GROUP($id$, $sk^A_{DH}$, $cache_{pk}$)
    $(metadata', \{keys^c, keys^1, \ldots, keys^n\}) \leftarrow$ download($id$)
    for all $keys^i \in \{keys^c, keys^1, \ldots, keys^n\}$ do
        // Add public keys of all other participants if they have not been added yet
        if $i \in cache$ then
            $(pk^i_{DH}, pk^i_{sig}) \leftarrow cache_{pk}[i]$
        else if $i \neq A$ then
            $(pk^i_{DH}, pk^i_{sig}, \_) \leftarrow keys^i$
            $cache_{pk}[i] \leftarrow (pk^i_{DH}, pk^i_{sig})$
        end if

        // Obtain encrypted key
        if $i = A$ then
            $(pk^A_{DH}, pk^A_{sig}, (ctxt_{ssk}, n_{ssk}, sig_{ctxt_{ssk}})) \leftarrow keys^i$
        end if
    end for

    // Obtain encryption key
    if $A$ is creator then
        $sk_{enc} \leftarrow sk^A_{DH}$
    else
        $sk_{enc} \leftarrow DERIVE\text{-}SHARED\text{-}KEY(pk^c_{DH}, sk^A_{DH})$
    end if

    // Decrypt and return shared secret key ssk
    $ssk \leftarrow DECRYPT(ctxt_{ssk}, n_{ssk}, sk_{priv})$
end procedure
```

2.3.9 File Key Hierarchy

In Sharekey, every file and folder has its own randomly generated secret key. This key is used to encrypt some metadata of the file or folder and to encrypt the file content, as described in Subsection 2.3.10. The user that creates such an object randomly generates the secret key and encrypts it with their private key sk_{priv} using *ENCRYPT-THEN-SIGN*. Additionally, the object's key is also encrypted with the secret key of the parent folder using *ENCRYPT-THEN-SIGN*, and both ciphers are stored with Sharekey. This allows someone that knows the secret key of a folder to also decrypt all secret keys of the folder's children. As a special case, every user also owns their root folder, which also has a randomly generated secret key sk_{root}. This root folder does not have a parent, therefore the user only encrypts sk_{root}

with their own private key sk_{priv} but not with the folder's parent key. An example of folder structure and how their keys relate to each other can be seen in Figure 2.3.

2.3.10 File Upload

When uploading a file, the creator creates a new random secret key sk_{enc} for that file, which they encrypt with their private key sk_{priv} and the secret key of the parent folder $sk_{enc,parent}$. They then submit their public signing key $sk_{user,sig}$, metadata encrypted with the file secret key $sk_{enc,file}$, and the encrypted keys $enc^{owner}_{sk_{enc,file}}$ and $enc^{parent}_{sk_{enc,file}}$ to the Sharekey server, which returns the identifier of the created file id_{file} as well as a list of all chunks on which file data can be uploaded. Sharekey splits its files into 10 Mb chunks, along with thumbnails and the preview which are stored in one chunk each. The user then uploads their encrypted data to the corresponding chunks, each of which is signed separately (compare Algorithm 21). The file upload server will then check that the chunk signature matches the public signature key $pk_{sig,file}$ that was provided by the user (compare Algorithm 22). The data will be stored in two parts. Their main database stores an entry with the file information and chunk ids as seen in Table A.2. Additionally, their file server holds the encrypted data for each chunk. There is also some additional metadata, which is abstracted into one *meta* field and will be discussed in Section 2.3.12.

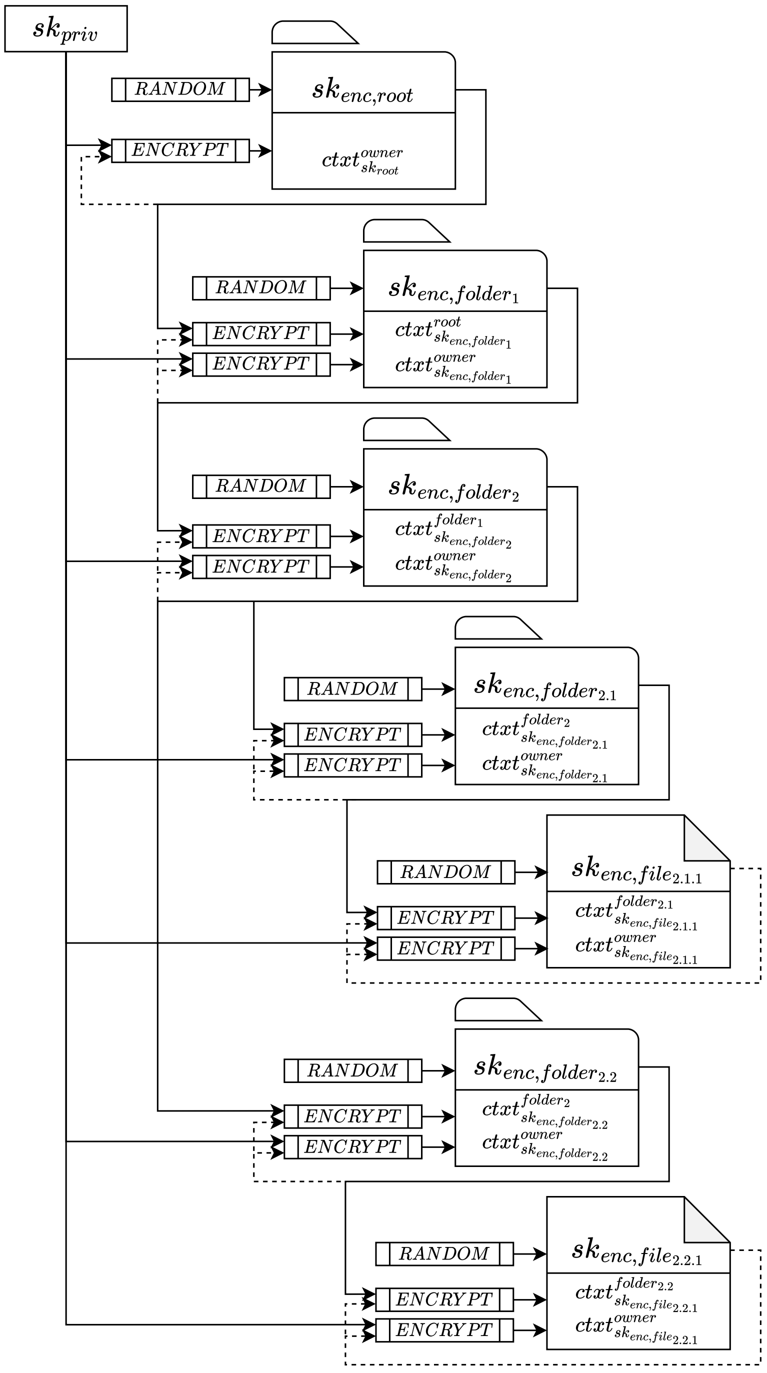

sk_{priv}
RANDOM
ENCRYPT
$sk_{enc,root}$
$ctxt^{owner}_{sk_{root}}$
$sk_{enc,folder_1}$
$ctxt^{root}_{sk_{enc,folder_1}}$
$ctxt^{owner}_{sk_{enc,folder_1}}$
$sk_{enc,folder_2}$
$ctxt^{folder_1}_{sk_{enc,folder_2}}$
$ctxt^{owner}_{sk_{enc,folder_2}}$
$sk_{enc,folder_{2.1}}$
$ctxt^{folder_2}_{sk_{enc,folder_{2.1}}}$
$ctxt^{owner}_{sk_{enc,folder_{2.1}}}$
$sk_{enc,file_{2.1.1}}$
$ctxt^{folder_{2.1}}_{sk_{enc,file_{2.1.1}}}$
$ctxt^{owner}_{sk_{enc,file_{2.1.1}}}$
$sk_{enc,folder_{2.2}}$
$ctxt^{folder_2}_{sk_{enc,folder_{2.2}}}$
$ctxt^{owner}_{sk_{enc,folder_{2.2}}}$
$sk_{enc,file_{2.2.1}}$
$ctxt^{folder_{2.2}}_{sk_{enc,file_{2.2.1}}}$
$ctxt^{owner}_{sk_{enc,file_{2.2.1}}}$

Figure 2.3: Example folder structure showing how keys relate to each other. $ctxt^{A}_{obj}$ denotes a ciphertext of obj that is encrypted by a secret belonging to A. Dashed lines are used for data that is being encrypted, whereas the encryption keys are connected through solid lines.

Algorithm 21 Upload file metadata and chunks

```
procedure USER-UPLOAD-FILE($sk_{enc,parent}$, $sk_{priv}$, $sk_{sig,owner}$, $pk_{sig,owner}$, $meta$, $data$)
    // Generate file key
    $sk_{enc,file} \leftarrow_{\$} \{0,1\}^{256}$
    $sk_{sig,file} \leftarrow sk_{sig,owner}$
    $pk_{sig,file} \leftarrow pk_{sig,owner}$

    // Encrypt file key and metadata
    $enc^{owner}_{sk_{enc,file}} \leftarrow ENCRYPT(sk_{enc,file}, sk_{priv})$
    $enc^{parent}_{sk_{enc,file}} \leftarrow ENCRYPT(sk_{enc,file}, sk_{enc,parent})$
    $encsig_{meta} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(meta, sk_{enc,file}, sk_{sig,file})$

    // Obtain IDs
    $N \leftarrow |data|$                                        ▷ Number of Chunks
    $safeObj \leftarrow API\text{-}UPLOAD\text{-}HEADER(enc^{owner}_{sk_{enc,file}}, enc^{parent}_{sk_{enc,file}}, ctxt_{meta}, pk_{sig,file}, N)$
    $(id_{file}, (id_{chunk_1}, \ldots, id_{chunk_N})) \leftarrow safeObj$

    // Upload Chunks
    for all $id_{chunk_i} \in (id_{chunk_1}, id_{chunk_2}, \ldots, id_{chunk_N})$ do
        $encsig_{chunk_i} \leftarrow ENCRYPT\text{-}THEN\text{-}SIGN(data_i, sk_{enc,file}, sk_{sig,file})$
        $API\text{-}UPLOAD\text{-}CHUNK(id_{file}, id_{chunk_i}, encsig_{chunk_i})$
    end for
end procedure
```

Algorithm 22 File header upload from servers perspective

```
procedure API-UPLOAD-HEADER($enc^{owner}_{sk_{enc,file}}$, $enc^{parent}_{sk_{enc,file}}$, $encsig_{meta}$, $pk_{file,sig}$, $N$)
    // Generate File key and encrypt it for Owner
    $id_{file} \leftarrow_{\$} \{0,1\}^{size_{id}}$
    $ids_{chunk} \leftarrow_{\$} \{\{0,1\}^{size_{id}}\}^N$

    // Store information in database (DB)
    $DB[id_{file}]["header"] \leftarrow (enc^{owner}_{sk_{enc,file}}, enc^{parent}_{sk_{enc,file}}, encsig_{meta}, pk_{file,sig}, ids_{chunk})$

    // Return information to user
    return $(id_{file}, ids_{chunk})$
end procedure
```

Algorithm 23 File chunks upload from servers perspective

```
procedure API-UPLOAD-CHUNK($id_{file}, id_{chunk_i}, encsig_{chunk_i}$)
    $(enc^{owner}_{sk_{enc,file}}, encsig_{meta}, pk_{sig}, ids_{chunk}) \leftarrow DB[id_{file}]["header"]$

    // Verify Chunk ID
    if $id_{chunk} \notin ids_{chunk}$ then
        error
    end if

    // Verify Signature
    $(ctxt, n, sig) \leftarrow encsig_{chunk_i}$
    if not $VERIFY\text{-}SIGNATURE(ctxt, sig, pk_{sig})$ then
        error
    end if

    // Store Chunk
    $BUCKET[(id_{file}, id_{chunk})] \leftarrow encsig_{chunk_i}$
end procedure
```

2.3.11 File Download

Any user can download a file if they know its ID and if they know its encryption key $sk_{enc,file}$, they can also decrypt its content. In the official client only users that also know the encryption key $sk_{enc,file}$ will ever attempt to download the file, but with a modified client it is also possible to request this data without knowledge of the encryption key $sk_{enc,file}$. As shown in Algorithm 24, a user will first obtain the header of the back-end's Algorithm 25, followed by downloading each chunk individually using Algorithm 26 of the back-end. They can then with the knowledge of the encryption key $sk_{enc,file}$ decrypt the file's metadata and chunks.

Algorithm 24 Download a file from the user's perspective

```
procedure DOWNLOAD-FILE($id_{file}, sk_{enc,file}$)
    // Obtain File Header
    $header \leftarrow API\text{-}DOWNLOAD\text{-}HEADER(id_{file})$
    $(ctxt_{byUser,sk_{enc}}, encsig_{meta}, pk_{sig}, ids_{chunk}) \leftarrow header$

    // Decrypt Metadata
    $(ctxt_{meta}, n_{meta}, sig_{ctxt_{meta}}) \leftarrow encsig_{meta}$
    $meta \leftarrow DECRYPT(ctxt_{meta}, n_{meta})$

    // Download Chunks
    for all $id_{chunk_i} \in ids_{chunk}$ do
        $(ctxt_{data_i}, n_{data_i}, sig_{ctxt_{data_i}}) \leftarrow API\text{-}DOWNLOAD\text{-}CHUNK(id_{chunk_i})$
        $data_i \leftarrow DECRYPT(ctxt_{data_i}, sk_{enc,file})$
    end for

    $N \leftarrow |ids_{chunk}|$
    return $(meta, (data_1, \dots data_N))$
end procedure
```

Algorithm 25 Provide a user with the encrypted file headers

```
procedure API-DOWNLOAD-HEADER($id_{file}$)
    return $DB[id_{file}]["header"]$
end procedure
```

Algorithm 26 Provide a user with an encrypted chunk

```
procedure API-DOWNLOAD-CHUNK($id_{file}, id_{chunk}$)
    return $BUCKET[(id_{file}, id_{chunk})]$
end procedure
```

2.3.12 Metadata

This subsection focuses on the metadata of files and folders in Sharekey. While the majority of the data stored in Sharekey is encrypted, some metadata remains unencrypted. In this subsection, we will discuss the most notable entries for both files and folders. A complete list of all entries can be found in Table A.2 for files and Table A.1 for folders.

Files For files, the unencrypted metadata includes the file owner, MIME-Type, parent folder, file size, time of upload, and a list of channels where the file is shared in. In contrast, file contents, thumbnail and preview images, file name, creation, and last modified date are all encrypted. Additionally, each file chunk, the metadata, and the preview images are signed. Each of them has its signature, which is a signature over the raw ciphertext, with no information about what file they are part of or what type of data it contains.

Folders Folders in Sharekey contain metadata similar to that of files, except for file content. The unencrypted metadata for folders includes the number of children, the parent folder, and a list of channels the folder is shared in. Meanwhile, the name of the folder, as well as its last created and last modified dates, are encrypted. As with files, all encrypted data for folders is also signed separately. These signatures are done over the raw ciphertexts of the encrypted fields and are not linked to the object they belong to or the type of data that is signed. It is important to note that the directory structure can be reconstructed through the unencrypted parent folder attribute of both files and folders. To find the children of a folder, Sharekey queries its database of files and folders for objects that specify the queried folder as their parent.

2.3.13 External Links

Sharekey allows for the creation of external links, which allow participants to share objects with people that do not have a Sharekey account. These links can have usage limits, which are enforced by the Sharekey server and limits the number of times a file can be downloaded. The main challenge of sharing objects is relaying the secret key sk_{obj} to the other users without disclosing it to the Sharekey server. The first approach is described in Algorithm 27, where the encryption of the file secret sk_{obj} key with a secret key sk_{link} derived from a password pwd is stored in the Sharekey database. The second strategy, which is described in Algorithm 28, encodes the object's secret key in the URL of the external link.

Algorithm 27 Create External link with password

```
procedure CREATE-LINK-WITH-PASSWORD($id_{obj}, sk_{obj}, pwd, usage\text{-}limit$)
    // Encrypt object key with the key derived from password
    $sk_{link} \leftarrow SCRYPT(pwd, id_{obj})$
    $(ctxt_{sk_{obj}}, n_{sk_{obj}}) \leftarrow ENCRYPT(sk_{obj}, sk_{link})$

    // Encrypt password for later retrieval by anyone that knows object key
    $(ctxt_{pwd}, n_{pwd}) \leftarrow ENCRYPT(pwd, sk_{obj})$

    // Generate signature keys for link
    $(sk_{sig}^{link}, pk_{sig}^{link}) \leftarrow gnerate\text{-}keys_{sig}()$
    $(ctxt_{sk_{sig}^{link}}, n_{sk_{sig}^{link}}) \leftarrow ENCRYPT(sk_{sig}^{link}, sk_{link})$
    upload($id_{obj}, (ctxt_{pwd}, n_{pwd}), ctxt_{sk_{obj}}, ctxt_{sk_{sig}^{link}}, usage\text{-}limit$)
    return $concat(base\text{-}url, "/", id_{obj})$
end procedure
```

Algorithm 28 Create External link without password

```
procedure CREATE-LINK-WITHOUT-PASSWORD($id_{obj}, sk_{obj}, usage\text{-}limit$)
    upload($id_{obj}, usage\text{-}limit$)
    return $CONCAT(base\text{-}url, "/", id_{obj}, "/", hex(sk_{obj}))$
end procedure
```

Chapter 3

Protocol Review

This chapter presents a comprehensive review of Sharekey's current protocol. For our review, we used a local deployment of Sharekey and a custom Python client to be able to explore potential attack vectors in a semi-automated and repeatable way, for example, to check whether an attack vector still works after changes by Sharekey. To achieve greater coverage, we also conducted automated tests to identify potential vulnerabilities in Sharekey's protocol. Then we present the specific discoveries we made regarding Sharekey's security and privacy. We conclude by discussing the remediation strategies currently or soon implemented by Sharekey, followed by an evaluation of the validity of their claims from Section 2.2 in light of the discovered findings.

The review only considers the front-end and back-end code running locally, not the actual deployment, and we have not checked if their infrastructure is properly configured. We have also touched on the other features and non-cryptographic parts of their application, but this was not the main focus of the thesis. Additionally, their call feature was in beta and not fully finished at the time of this work and therefore not prioritized. The review will consider a local deployment of Sharekey accessed through the web application, the Android, iOS, and Desktop applications were not tested.

3.1 Methods

3.1.1 Local Deployment

For our research, Sharekey granted us access to the front-end and back-end code of the application and helped us to set up a local deployment of their application stack on our own computing infrastructure. With access to database schemes, logs created by the back-end, and overall full control over the system, we had all the information needed and could do all the tests we wanted without having to worry about accidentally breaking anything. Overall, the local deployment was an invaluable resource for our research, allowing us to be more efficient in our assessments.

3.1.2 Custom Client

Despite having access to the official web client from Sharekey, we decided to create our own lightweight Python client with a command line interface (CLI) and application programming interface (API) to gain a better understanding of the protocol and to have more flexibility in testing. Our main approach in writing the client was to first perform an action in the web browser and record the traffic that was generated. We then looked at the source code of the web client to understand what was happening in the protocol. From this, we translated the necessary functionality into our own Python client, mimicking the behavior of the web client. This helped us gain a deeper understanding of the protocol and allowed us to build an easily accessible API to perform different tests.

During our work, we first investigated manually several attack vectors, before automated testing. The next paragraphs shortly introduce the used methods, while the details of our findings can be seen in Section 3.2.

3.1.3 Manual testing

During our testing, we thoroughly reviewed the back-end implementation of Sharekey and attempted to find edge cases and unexpected inputs to modify messages sent from our client. We identified all API endpoints relevant to the messaging and file upload features we tested and systematically reviewed the logic for each function.

There was no formal description of the protocol, so the protocol was deduced by reviewing the source code of Sharekey, as described in Section 2.3. There was also no concrete list of desired security properties for the protocols we analyzed, so we identified potentially desirable security properties for messaging and file upload which are listed in Section 4.2. From there we reviewed the protocol to find ways of breaking these properties.

For each function, we started by trying to break the properties as an adversary within the back-end first, by modifying the data stored in their database. We then tried to find ways to perform any attacks we found from a client. Lastly, we tried to find ways of breaking the properties that were only possible from clients but not the back-end. This provided us with attacks on the protocol from adversaries with either the capabilities of Sharekey or a user of their application.

Furthermore, we also tried to find vulnerabilities caused by the specific implementation of the protocol. One cause of such issues can be caused by malicious input to the API that would in most cases never be sent by honest clients. However thanks to our client, we had full control over the data sent to the back-end. Messages sent between clients and the back-end were in JavaScript Object Notation (JSON) and most functions implemented structure and content verification for the messages sent by the client. We tried to find values that while passing these verifications would result in unexpected behaviors.

Another attack vector we explored was the contents of encrypted data that could not be verified at the back-end. As an example, the plain text messages sent with Sharekey are in JSON format and since they are encrypted, the back-end cannot validate its structure, so the verification has to be handled by the receiving clients. We tried to find values for these encrypted fields that would break any of the security properties or let to otherwise unexpected behavior.

For the attacks we found with our Python client we created a CLI application that allowed us to exploit the issues we found. This allowed us to test the remediations that were brought forward by Sharekey to see if our original attacks would still work with their updated versions. An example of the execution of such an exploit can be found in Listing C.1.

3.1.4 Automated testing

The manual tests were extended through automated testing, to ensure coverage of all API endpoints, including those that were not otherwise a focus of the review.

One approach to do this was to extract all possible API endpoints from the back-end source code. We could then automate requests to each of these endpoints and process the answers we received. However, due to the structure and content verification by the back-end, most of these requests resulted in the same error message, caused by the invalid format of the data we provided. A much more insightful approach was therefore to collect inputs that pass the validation by using the web client ourselves and recording the messages to and from the back-end. We performed all actions reachable from the web client to obtain an extensive list of API endpoints listed in Section A.2, which are paired with values that would pass the back-end verification.

We extended our recorded API endpoint value pairs, by randomly substituting randomly chosen fields in the data we sent with values from the "Big List of Naughty Strings" [33]. We then performed authentication checks on this extended list of pairs, by performing the same request once without an authenticated session to Sharekey and then again with an authenticated session. Our tool would report requests where both calls resulted in the same response from Sharekey, indicating that this endpoint is accessible without authentication. We then manually checked the resulting endpoints to verify if it was justified that these were accessible without authentication.

In addition, our tool would report any request that would result in errors returned by the back-end. These errors were filtered to only contain errors that are not caused by the input validation to find errors that occur in other places of the application. The remaining errors were reviewed and checked to see if we manage to break any security properties using this or a similar input and to check for information leakage in the error message of the back-end.

Lastly, we also measured the response time of our requests, looking for inputs that would require a lot of resources in Sharekey.

Overall these approaches helped to obtain a broad view of the application but did not allow us to find issues that were as complex as the ones found through manual testing.

3.2 Results

This section lists the findings we made in our assessment of the Sharekey protocol and implementation, gives ideas on remediating the issues, and references the actions taken by Sharekey as of April 2023. The findings are of diverse nature, including cryptographic issues but also more general security and privacy concerns. Starting with the verifiability of the public key material, we continue discussing several findings regarding the file-sharing protocol. This is followed by our findings on the messaging protocol and concluded by some more general findings that are not directly related to the other two categories. The attacks that can be performed by Sharekey users can be reproduced using the API of our Python client, with some attacks streamlined into a CLI as seen in Section C.1 and the Source Code published in the GitHub repository of this thesis [34].

There have been efforts to remediate some of the following findings, which are shown in Section 3.3.2.

Finding 1. Public keys not verifiable
Each user owns a public DH key pk_{DH} to establish shared secrets and and a public signature key pk_{sig} used to verify messages by that user. However, these public keys are managed and distributed by Sharekey, and users have no possible way of verifying that the public keys of other participants are actually the keys created by that participant. An attacker within Sharekey could substitute these public keys with other keys for which they know the secret key and for example, perform a man-in-the-middle attack to eavesdrop and modify any messages sent between two participants. Therefore it

is important that participants can verify with each other that they have the same view of the public keys involved in their communication.

Remediation Ideas: One approach could be verification through a central certificate authority managed by the customer company. Another option could be out-of-band verification, where participants can compare and verify their public keys through a different trusted channel or in person.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 2. File chunks can be uploaded by other users
Files stored with Sharekey are split into chunks and uploaded using a file upload helper, which is a separate server from the general back-end. While the user and the general back-end maintain an authenticated session, this session is not extended to the file upload helper. The file upload helper, therefore, does not know the user id of the participants that communicate with it and needs another way of ensuring that only file owners can upload the file chunks. This is done by requiring each uploaded chunk to be signed by the file owner. However, the signatures that are checked just cover the raw ciphertext of the file. The file upload helper, therefore, accepts any data that is signed by the file owner as a valid chunk. An attacker with access to the chunk id of a file by some owner and any data signed by that owner can therefore impersonate the real owner by submitting that signed data instead of the real chunk content to the file upload helper. Data signed by that owner is readily available, for example, all messages sent by the file owner in any channel are also signed by the owner.

Remediation Ideas: The issue could be remediated by ensuring that the signature also contains some information on the intended use of the signed data. We believe that our suggested update to the file upload protocol as seen in Section 5.2 remediates this issue, by having a single signature for the entire file, that covers the order and content for each chunk of that file.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 3. File integrity not guaranteed
There is no way for participants to check the integrity of an entire file. While the different chunks that make up the file are signed by the file owner, there is no signature over the combination of these chunks. Participants would not be able to detect if some chunks are missing or reordered, which can lead to situations where a user believes they are downloading a file as it was uploaded by the file owner, even if the file is not the same anymore. An adversary in the back-end can reorder or even delete chunks within a file to

break its integrity. Furthermore, other users with access to the file ID could also break the integrity of the file by downloading several signed chunks and reuploading them using Finding 2.

Remediation Ideas: Integrity of the entire file could be achieved by having one signature that covers all chunks in a file including their intended order. We believe that our suggested update to the file upload protocol as seen in Section 5.2 might remediate this issue.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 4. File chunks overwritable
Instead of being writable just once, file chunks can be overwritten in Sharekey. This might not be an issue in itself, but in combination with Finding 2, this allows an attacker to modify existing files, also after their chunks have been uploaded. This can either be used to destroy a file by overwriting a chunk with data that while signed by the user is not a valid ciphertext under the file's key and will fail MAC verification. Another attack would be uploading data that was encrypted with the file key, producing chunks that decrypt without error and violate file integrity as described in Finding 3.

Remediation Ideas: This issue can be mitigated by ensuring that file chunks can only be uploaded once and cannot be overwritten.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 5. File tree traversable
Any participant with a Sharekey account can traverse folders back to one folder above the root folder. Whenever they know the ID of any file or folder, they can request that file's parent folder ID. From there they can request that folder's parent and so on until they reach a folder whose parent is the root folder. From there, they can again request all children of this folder and obtain the entire folder tree structure. While such an attacker does not learn any new secret keys from files or folders, they can learn about their existence and metadata such as file owner, file upload data, file, and folder owners, and in which channels these folders are shared. In Figure 2.3, someone with access to $file_{2.1.1}$ could also find information about $folder_2$, $folder_{2.1}$, $folder_{2.2}$ and $file_{2.2.1}$.

Remediation Ideas: Sharekey should not send the ID of parent folders to users that do not have permission to view this parent folder.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 6. Unauthenticated access to shared objects of channel
Any unauthenticated user with knowledge of a channel ID can obtain a list of all files and folders that were shared with this channel. In combination with Finding 4 and 5, this can be chained into an attack where an adversary locates as many file IDs as possible, downloading all chunks for each file and then overwriting them with some meaningless data. They could then ask for a ransom for restoring the files as they were.

Remediation Ideas: Currently, Sharekey collects enough metadata to know which users are part of what channel. Therefore they can and should verify if the participant who wants to view the list of shared objects in a channel is also a member of that channel. If they want to reduce the trust users have to have in Sharekey, they could also encrypt the list of shared objects in a channel with the channel key, ensuring that the list of shared objects in a channel stays confidential, which would also imply that they themselves would not have access to that list.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 7. File key leakage on passwordless links
Sharekey allows the creation of links for sharing files and folders. Per default, these links are protected by a password, and the file or folder's secret key is encrypted with a key derived from this password. However, it is also possible to share such links without a password, which would create a link of the following form: `http://app.sharekey.com/drive/<file-id>/<file-sk>`. The file's secret key is encoded as a hexadecimal string and added to the path of the file, allowing the web client to read the file's key from the URL. However, this also allows Sharekey to obtain the file secret key, as the entire path will be sent to Sharekey as part of the request.

Remediation Ideas: A better way might be to add the secret key to the URL as a page fragment: `http://app.sharekey.com/drive/<file-id>#<file-sk>`. Page fragments can still be read using JavaScript in the client's browser, but are not sent to the server as part of the request.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 8. File ID's swappable
An adversary with access to the Sharekey database could swap the content of files and folders. For example, if there are two files in the same folder that are shared in a channel, users cannot detect if Sharekey swaps these files. Users could ask for a file with identifier id_1 but have another file with identifier id_2 served to them.

Remediation Ideas: Integrity protection of a file should extend to its metadata fields including the file ID. Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 9. Messages editable by other users (back-end authentication)
Our evaluation revealed that messages can be edited without the authentication of the user that edits a message. Anyone who knows the message ID can change the message ciphertext. These message IDs are known to any participant in the channel that the message is sent with. These participants can change the ciphertext of the messages sent in that channel. Adversaries that know the encryption key of that channel, such as a malicious channel member, can create new ciphertexts for which the MAC verifies and that decrypt to a plaintext in a valid JSON format.

Remediation Ideas: Currently, Sharekey collects enough metadata to know the sender of a message. Therefore they can and should verify if the participant who wants to edit a message is also the sender of that message.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 10. Messages editable by other users (signature checks)
In Sharekey, message content and message timestamps are each signed by the message sender. However, these signatures are not verified when messages are received and there is therefore no guarantee, that the message was actually sent by the supposed message owner. While Sharekey uses an authenticated encryption scheme, which ensures that message ciphertext can not be forged and only created by users that know the secret key, this key is known to all participants in a Channel. Together with Finding 9 this would allow a malicious channel member to edit a message of another member, with full control over the new message content.

Remediation Ideas: Whenever the protocol provides a signature alongside the encrypted data, the signature should be verified by the receiving participant. Messages with malformed signatures should not be accepted and marked as such.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 11. Messages deletable by other users (back-end authentication)
Sharekey allows users to delete their messages. However, similar to Finding 9, there is no authentication check to determine who is deleting a message, meaning anyone can delete a message if they know the message ID.

Similar to Finding 9, these IDs are known to all participants in the channel the message is sent with. Another

Remediation Ideas: Also here Sharekey collects enough metadata to know the sender of a message. Therefore they can and should verify if the participant who wants to delete a message is also the sender of that message.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 12. No proof that messages were deleted by sender
Sharekey's messaging protocol does not contain any cryptographic proof that such deletion was initiated by the message owner. This means that other users cannot be convinced that the deleted message was indeed deleted by the message sender. Sharekey could potentially forge a message deletion.

Remediation Ideas: When a user deletes a message it could provide some data that specifies the intent to delete the message in question together with a signature over that data.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 13. Messages reorderable by the back-end
Messages do currently not contain any sequence information. As the timestamps and messages are encrypted separately, messages can be reordered by reassigning different encrypted timestamps to messages.

Remediation Ideas: One object should ideally only have one signature that covers all fields of that object. For messages, this means that there should be one signature that covers both the encrypted message content and the encrypted timestamp.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 14. No speaker consistency
In a protocol with speaker consistency, all participants agree on the sequence of messages sent by each participant. Messages do currently not contain any sequence information. As the timestamps and messages are encrypted separately, messages can be reordered by reassigning different encrypted timestamps to messages. Messages can be reordered differently for every participant, breaking this property.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

Finding 15. Not causality preserving
A messaging protocol that is causality preserving can avoid displaying a message before messages that causally precede it. In Sharekey messages are sorted by their timestamps. An adversary controlling a Sharekey user could send a message in a channel with a timestamp far in the future. The messages of honest participants that respond to that message would be shown before the adversarial message with the malicious timestamp.

Status: As discussed in Section 3.3.3, this property is hard to achieve using Sharekey's setup and there is no trivial way of how this property can be fulfilled as of now.

Finding 16. Invalid messages crashing web app
Invalid message contents could crash web applications. For example, a client that is trying to display the message with content {`"break": "app"`} crashed the browser tab in which it's running. The issue arises from the content rendering code trying to access the message text which does not exist.

Remediation Ideas: Any user data that cannot be validated in the back-end because it's encrypted has to be thoroughly validated by the client.

Status: Sharekey has already put forward a remediation for this issue as explained in Section 3.3.2.

Finding 17. Missing key separation
Every key should only hold one purpose, other keys should then be derived from the initial keys. Currently, sk_{DH} (`ECDHSK` used for DH key exchanges when negotiating channel keys) is the same as sk_{priv}, which is the key derived by the user's passphrase.

Remediation Ideas: One key should only be used for one purpose. Ideally, new keys are derived from the original if it is intended to be used for several purposes. Otherwise, there is a danger of losing security guarantees of the algorithms, due to their misuse in conjunction with the same secrets. While it might be difficult to change this for existing users, the current architecture would allow the separation of sk_{DH} and sk_{priv} for new users.

Status: As we did not find a way of exploiting this issue, the remediation of this issue is not prioritized but remains part of their long-term road map.

Finding 18. User-controlled HTML emails
Sharekey allows us to invite other users by email. Their clients offer a form where users can enter a list of email addresses and a custom message that will be included with the invitation. One restriction Sharekey imposes is

that the custom messages may not include any URLs. This could however be bypassed when parts of the hyper reference were URL-encoded. For example, someone could have created a custom message with the content `hi <a href=https://duckduckgo%2ecom>there</a>`. This passed the filter but since `%2e` is the URL-encoding for `.`, some email clients would translate the hyper reference to `https://duckduckgo.com`. Furthermore, the possibility to add HTML tags that will be interpreted as such by email clients opened the possibility for further modification of the message. By closing HTML tags preceding the custom message, the format of the message could be changed to fully control the style of the displayed message, as seen in Figure 3.1.

Status: Sharekey will put forward a remediation for this issue as explained in Section 3.3.2.

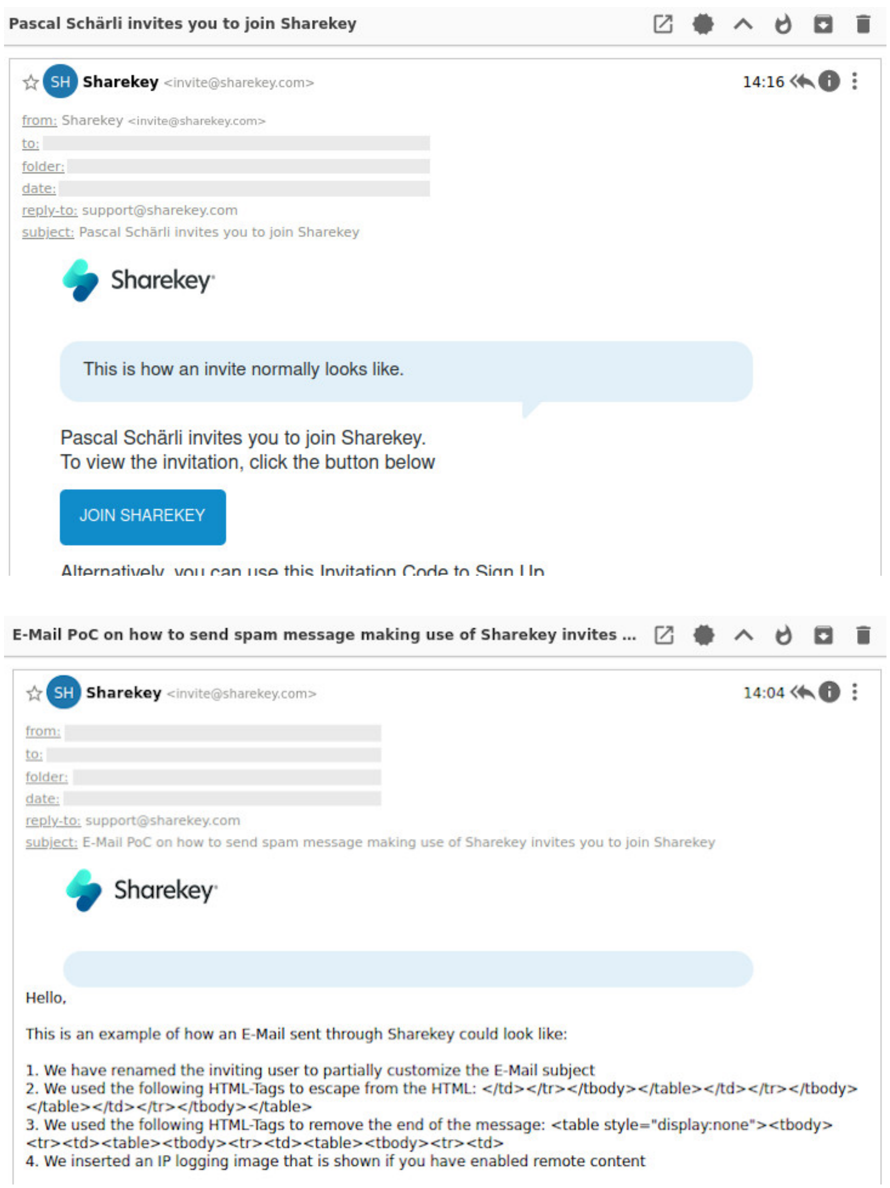
Finding 19. Account enumeration
It is possible to check if an email address is tied to a Sharekey account or not. While not possible in the official client, a modified client can request public and encrypted cryptographic keys for any email address hash. If the requested email address belongs to an existing user, the server returns the keys of that user, otherwise an empty message. The official client uses this functionality during login, which is why it can be called before an authenticated session between the user is established. This also means that an adversary does not need to have a Sharekey account to mount this attack.

Remediation Ideas: Instead of requesting their keys through their email address hash, the users could be required to encrypt their email address with a fixed nonce and sk_{priv} to request their key material. We verified that sk_{priv} is known to the users in this stage of the login process. While it is easy for an adversary to calculate the hash of an email address, they cannot easily obtain the email address encrypted by the user.

Status: The remediation of this issue is not prioritized but remains part of their long-term road map.

3.3 Discussion

We identified several attack vectors and managed to show their validity by writing working exploits for the findings regarding the adversarial Sharekey client. All findings listed in this thesis were confirmed by Sharekey. This section will discuss our perceived severities of the issues we found and show the actions taken by Sharekey. We will specifically address the feasibility of implementing message ordering properties, as there is no straightforward solution for this in the current environment. Furthermore, at the end of this



Pascal Schärli invites you to join Sharekey

Sharekey <invite@sharekey.com> 14:16

from: Sharekey <invite@sharekey.com>
to: [redacted]
folder: [redacted]
date: [redacted]
reply-to: support@sharekey.com
subject: Pascal Schärli invites you to join Sharekey

Sharekey

This is how an invite normally looks like.

Pascal Schärli invites you to join Sharekey.
To view the invitation, click the button below

JOIN SHAREKEY

Alternatively, you can use this Invitation Code to Sign Up

E-Mail PoC on how to send spam message making use of Sharekey invites ...

Sharekey <invite@sharekey.com> 14:04

from: [redacted]
to: [redacted]
folder: [redacted]
date: [redacted]
reply-to: support@sharekey.com
subject: E-Mail PoC on how to send spam message making use of Sharekey invites you to join Sharekey

Sharekey

Hello,

This is an example of how an E-Mail sent through Sharekey could look like:

1. We have renamed the inviting user to partially customize the E-Mail subject
2. We used the following HTML-Tags to escape from the HTML: </td></tr></tbody></table></td></tr></tbody></table></td></tr></tbody></table>
3. We used the following HTML-Tags to remove the end of the message: <table style="display:none"><tbody><tr><td><table><tbody><tr><td><table><tbody><tr><td>
4. We inserted an IP logging image that is shown if you have enabled remote content

Figure 3.1: Top: Email invite as intended by Sharekey. **Bottom**: Email invite with malicious HTML content, escaping from the original form.

section, we will also revisit the security and privacy claims and address their validity and feasibility.

3.3.1 Severity of Findings

We believe that some of our findings may be of high severity based on our analysis and understanding of the protocol. We have found ways to compromise the confidentiality, integrity, and availability of sensitive user data.

One of the most significant issues we discovered is the lack of proper authentication mechanisms. This allowed us to impersonate other users by editing and deleting messages they sent or modifying the contents of the files they uploaded. Furthermore, the lack of public key verification would allow for the interception of all encrypted communication by Sharekey through an active man-in-the-middle attack.

By combining our file-sharing related findings, an adversary could have potentially executed an extensive ransom attack. First, they could collect as many file IDs as possible. Starting from the knowledge of a single file ID, an attacker could traverse the directories to find the IDs of other files from the same user using Finding 5. When requesting information about a file, a user can also obtain a list of channels that this file is shared in. We have not labeled this as a finding on its own, but it could be used in this attack scenario to get a list of channel IDs for all channels that any of the files found before were shared in. These channel IDs can then be used with Finding 6 to find all file IDs that were shared in any of these channels. An adversary could then start over by traversing the directories of all newly found files, repeating these steps until they have collected an extensive list of file IDs. The attacker could then download the encrypted chunks for each of these files. Then they could use Findings 2, 3, and 4 to overwrite the contents of each file. Because the attacker has previously downloaded all of the encrypted chunks, they could demand a ransom, promising to restore the contents of the files if the ransom is paid.

Lastly, we noticed that the current protocols do not facilitate provable security. To create provably secure protocols it is often necessary to choose specific design decisions when building the protocol, which can result in a trade-off between complexity and provability. This is shown in Finding 17, which indicates that some best practices in the design of cryptographic protocols were not considered when creating the protocol.

3.3.2 Remediation Status

Sharekey has collaborated well and already put forth remediations for many issues we raised. While the limited time frame allocated for this thesis did not permit a comprehensive second review iteration of the remediations implemented by Sharekey, we still want to acknowledge the efforts made by the platform to address the identified concerns. In this section, we will present the updates that Sharekey has provided and their anticipated effects. It should be noted that these updates have not been subjected to a thorough investigation by us due to the aforementioned constraints.

Current changes

As a result of this thesis, Sharekey has made the following changes to its protocol:

1. Added back-end authentication of users when editing a message to remediate Finding 9.
2. Added back-end authentication of users when deleting a message to remediate Finding 11.
3. Put the file key of passwordless links into URL fragment instead of the path to remediate Finding 7.
4. Only include parent folder ID in responses to users that have access rights to the parent folder to remediate Finding 5.
5. Added back-end authentication for obtaining the list of shared objects of a channel to remediate Finding 6.
6. Parsing of message JSON is done more robustly at clients to remediate Finding 16.

Changes in near future

Due to time constraints, it was not possible for Sharekey to implement all of the changes proposed while the thesis was still ongoing. One notable example is the file upload feature, which was only completed towards the end of the thesis, leaving insufficient time for its adaption by Sharekey. Nevertheless, it is important to highlight the changes that are intended to be implemented in the near future:

1. Implementation of our improved file upload protocol to remediate Findings 2, 3, 4 and 8.
2. Further efforts in designing and verifying file upload protocol, also in regards to the suggestions made in Section 5.4.
3. Out of band verification of channel keys to remediate Finding 1.
4. Message content and timestamps will still be encrypted separately, allowing Sharekey to verify that the timestamp does not change when editing messages. They will however have one signature over both the encrypted message and timestamp to remediate Finding 13.

5. Message deletion requests will contain some information that proves that the message owner initiated the deletion to remediate Finding 12.
6. Email content provided by users in invitations will be sanitized to remediate Finding 18.
7. Message signatures are currently not verified by the client due to their computational overhead. However, Sharekey is planning to introduce a local database in which they could also cache which messages were verified. This would allow them to verify message signatures in their clients and remediate Finding 10.

3.3.3 Message Ordering Properties

Not all identified issues are easily resolvable and some may lack straightforward solutions. In particular, message ordering properties, such as speaker consistency, causality-preservingness, and global transcript, may pose challenges to achieving within Sharekey's current setting. To address speaker consistency, Sharekey intends to implement per-device message counters, which contain the number of messages sent from that device so far as well as the number of total messages that that device has seen in that particular chat so far. They also plan to make a step towards a causality-preserving protocol by adding a reference to the previous message to the metadata. We do however not believe that the properties can be fully achieved with these measures and complete compliance with these properties would require substantial research effort to attain within Sharekey's operational requirements. Sharekey is planning to implement features such as "pin to blockchain," where selected excerpts of message transcripts can be agreed upon, potentially mitigating the need for the complete fulfillment of other message ordering properties.

3.3.4 Validity of Sharekey's Security and Privacy Claims

This section revisits the security and privacy claims by Sharekey from Section 2.2 and discusses their validity given our review and findings.

We did not find a way to break the confidentiality of messaging and file sharing supporting their Claims 2, 3, 4 and 16 that the encrypted data stays confidential towards Sharekey. We can also confirm the lack of cookies on their web application as stated in Claim 9. None of our findings deny Claim 14 about locally encrypted files and we commend Sharekey for their transparency about forward secrecy in Claim 17. However, Findings 1, 3, 7, 10 and 12 managed to break either the authentication or integrity properties that were stated in Claim 15.

There are various claims regarding the amount of metadata known to Sharekey and we cannot agree with all of the claims made. Claim 6 could be interpreted as a lack of any metadata collected by Sharekey, which we cannot confirm. For example, the DB entries for files and folders as seen in Tables A.1 and A.2, include several unencrypted fields, such as the files MIME-Type, which we would classify as metadata. In Claim 7, they state that they cannot build a social graph of their users, which we also cannot confirm. We believe that not only Sharekey is able to passively build social graphs, but that this would also be possible for adversarial users to some degree. Similar to the ransom attack described in Section 3.3.1, Findings 5 and 6 can also be used to obtain a list of channels and files, that can be used to create a social graph that uses the shared files as connections between the different channels. In Claim 13, they introduce the term "App-to-App" encryption as requiring minimal metadata visible to Sharekey, with the website comparing App-to-App encryption with a VPN tunnel between two users [21]. While we feel like the comparison to VPN tunnels is stronger than what they offer, their security paper does state that some minimal metadata is visible to Sharekey. How much metadata constitutes "minimal" is up for interpretation but we did observe that Sharekey is aiming to minimize the amount of unencrypted metadata. Furthermore, they also clarify that they do have access to metadata in Claim 18, which lists the metadata they collect in a non-exhaustive list.

Lastly, Claims 1, 5, 8, 10, 11 and 12 cannot directly be confirmed or denied, as they rely on trust in Sharekey which is up to the customers to decide.

Chapter 4

Threat Modelling

This chapter shows a threat model for both Sharekey's messaging and file-sharing protocols to ensure a comprehensive analysis of potential security threats. By modeling the architecture and components of the protocol execution, we identified various adversaries with differing capabilities and evaluated the fulfillment of security properties for each. Since there was no concrete list of desired security properties for the protocols we analyzed, we have identified potentially desirable security properties ourselves, based on literature and the claims made by Sharekey. From there we reviewed the protocol to find ways of breaking these properties. This process allows us to consider and analyze potential attack vectors in depth.

In our analysis, we limited our focus to the main components of Sharekey's architecture, specifically the client, servers, database, and cloud storage of larger files. We have not conducted an in-depth investigation into additional services such as push notification providers, code delivery to clients, the building of iOS and Android apps, version control, and continuous integration - continuous delivery (CI-CD) processes.

4.1 Methods

This section contains our approach to creating a threat model of the Sharekey application and infrastructure. We outline the key assets of the system and identify potential attackers who may target them. These can then be used to reason about the adversarial capabilities of adversaries located in different regions.

4.1.1 Regions of threat model

The threat model is divided into several regions, which can be seen as an overview of the scheme in Figure 4.1. The different regions are:

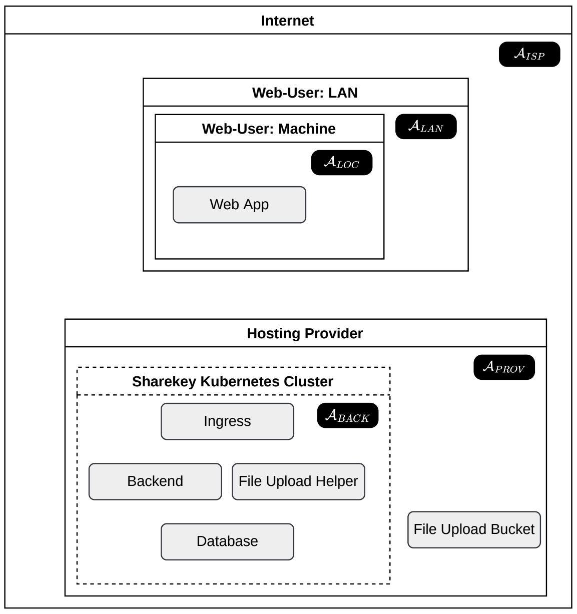


Figure 4.1: Scheme of different topological regions of the threat model. This figure contains all elements that appear in our threat models. The different topological regions are separated into dashed boxes, each of which contains a possible adversary. The Sharekey Kubernetes Cluster is outlined with a dotted line because it can be controlled by its hosting provider. Grey boxes within these regions show applications running within these regions.

- *Internet:* Contains all traffic that flows between services that are not in the same local network.
- *Web-User: LAN:* The local network of a web-application user.
- *Web-User: Machine:* The machine of a web-application user.
- *Hosting-Provider:* The cloud service provider, which Sharekey uses for hosting.
- *Sharekey Kubernetes Cluster:* The network between the collection of applications within the Kubernetes Cluster of Sharekey, which runs their back-end.

4.1.2 Adversaries

Each region is paired with a Dolev-Yao adversary [35], capable of full, active control of all data that is passing through its region.

- $\mathcal{A}_{LAN}$: Active adversary within local network of a web-application user.

- $\mathcal{A}_{LOC}$: Active adversary in control of the machine of a web-application user.
- $\mathcal{A}_{PROV}$: Active adversary in control of Exoscale, which is the cloud service provider that is hosting the Sharekey servers.
- $\mathcal{A}_{BACK}$: Active adversary within Sharekey's Kubernetes Cluster which runs their back-end.
- $\mathcal{A}_{ISP}$: Active adversary in control of any data sent through the internet.

4.1.3 Assets

The regions contain several assets, either of logical form or containing information.

- *Web App:* The web application running inside the browser of user P_1. This is where the front end runs and the user interacts with Sharekey.
- *Local Storage:* The local storage of the browser is used to store key material of P_1 between sessions.
- *Ingress:* The ingress server of Sharekey is used to forward traffic to the correct service.
- *Backend:* The backup server instance handles the logic of incoming requests
- *Database:* The DB of Sharekey stores all persistent data for the backend, except for uploaded files. Sharekey uses several different applications to manage its database, which are abstracted into one instance for brevity.
- *File Upload Bucket:* The buckets are used to store chunks of larger uploaded files.

4.1.4 Breakability

For every property defined, we argue for each adversary whether they can break the property, which we describe using the breakability symbols as defined in Table 4.1. To make the table more informative, we do not argue whether a property holds even with the adversary in question, as this could lead to cases where the property does not hold, not because of the examined adversary but due to some other issue of the protocol. Therefore we assume, that anything not controlled by the adversary is honest and that the property would hold without the adversary's interference. Effectively, we're not necessarily arguing whether a property holds, but if we found a way for this specific adversary to break this property.

Additionally, we also want to clarify that properties marked as not broken are not proven to be fulfilled, but that we have not found a way to break this property within the scope of this work.

Symbol	Name	Description
●	not broken	We did not manage to break this property with a given adversary.
◐	Breakable (expected)	The adversary can break this property, but we expected this property to be broken by this adversary and don't see any sensible way of preventing this.
○	Breakable	The adversary can break this property and there are ways to remediate the issue.

Table 4.1: Explanation of breakability symbols

4.2 Results

In this section, we discuss the security issues we found with the Sharekey Collaboration Protocols, as of October 2022. We begin by discussing the vulnerabilities we identified in the protocol's design and implementation. Specifically, we focus on the messaging protocol and the file upload functionality. At the end of this section, we also present some general findings that do not fit into the previous categories.

4.2.1 Messaging

The information flow of the messaging protocol is shown in the diagram in Figure 4.2. As of October 2022, their clients do not offer to verify the honest distribution of public keys as documented in Finding 1. However, this and the following properties are discussed under the assumption of honest public key distribution, as we assumed everybody except the adversary to be honest. We also assume that the clients are running the protocol as specified in the code we reviewed, disregarding the possibility of an adversary modifying the clients that are distributed to the users. The messaging protocol was evaluated on properties suggested in [36]. In this subsection, we argue for every adversary whether we managed to find a way of breaking the discussed property, as summarized in Table 4.2.

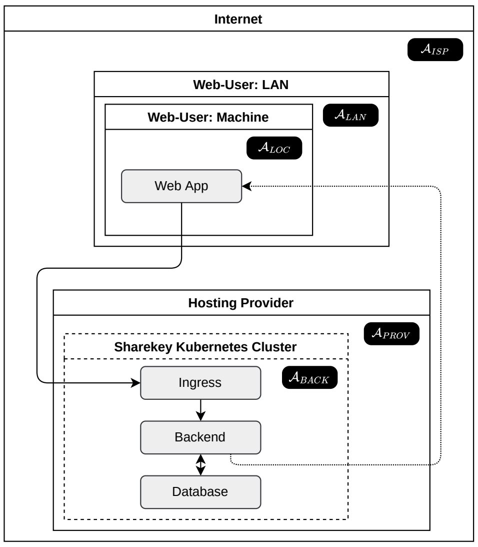


Figure 4.2: Information flow for messaging. The users send messages to Sharekey, which processes the messages and stores them in their DB. Messages received by a user are first retrieved from the DB by the back-end server and then sent back to the web application. Arrows show the flow of information between these instances. Information that originates from users is generally depicted with solid lines, while information that is going back to a user is depicted with dotted lines. These lines are semantically the same and only differentiated to improve readability.

	Confidentiality	Integrity	Authentication	Speaker Consistency	Causality Preserving	Global Transcript	Forward Secrecy	Post Compromise Security
$\mathcal{A}_{LAN}$	●	●	●	●	●	●	●	●
$\mathcal{A}_{LOC}$	◐	◐	○	●	○	○	◐	◐
$\mathcal{A}_{BACK}$	●	●	○	○	○	○	●	●
$\mathcal{A}_{PROV}$	●	●	○	○	○	○	●	●
$\mathcal{A}_{ISP}$	●	●	●	●	●	●	●	●

Table 4.2: Threat model summary for messaging. This table summarizes the breakability of several properties by the different adversaries. Refer to Table 4.1 for a definition of the symbols.

Confidentiality

This property states, that only the intended recipients can read a message. Specifically, the message must not be readable by a server operator or a user that is not a conversation participant.

●	$\mathcal{A}_{LAN}$	The messages between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel provides confidentiality for the messages sent through the channel.
◐	$\mathcal{A}_{LOC}$	The messages are decrypted in the user's browser. Anyone with access to a machine with an active session will therefore also be able to access the decrypted messages.
●	$\mathcal{A}_{BACK}$	Messages are encrypted using an `xSalsa20` cipher. We believe that the keys of these ciphers stay unknown to Sharekey under the assumption of honestly distributed public keys and we did not find a way to break this property as an adversary in control of the back-end.
●	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$ but for the control over the Exoscale cloud service provider.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Integrity

This property states, that no honest party will accept a message that has been modified in transit.

●	$\mathcal{A}_{LAN}$	The messages between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel provides integrity for the messages sent through the channel.
◐	$\mathcal{A}_{LOC}$	A local adversary would be in control of the participant's client and would therefore know the key to the *poly1305* MAC scheme. Therefore if an honest user sends a message from that client, the adversary could intercept that message, change it, recompute the MAC tag, and send the modified message instead.

●	$\mathcal{A}_{BACK}$	Messages are tagged using a `poly1305` MAC scheme. We believe that the keys of these MACs stay unknown to Sharekey under the assumption of honestly distributed public keys and we did not find a way to break this property as an adversary in control of the back-end.
●	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$ but for the control over the Exoscale cloud service provider.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Authentication

According to the authentication property, each participant in a conversation must receive proof of possession of a known long-term secret from all the other participants in the conversation, more specifically, from everyone they believe to participate in the conversation. Moreover, each participant needs to be able to verify that a message was sent from the claimed source.

●	$\mathcal{A}_{LAN}$	The messages between the web application and Sharekey are secured with TLSv1.2+. We assume that this is used properly and Sharekey is authenticated by proving ownership of a valid TLS certificate. Users are authenticated by Sharekey through proof of knowledge of their secret key material and access to their email addresses. Assuming honest behavior except for the adversary $\mathcal{A}_{LAN}$, we did not find a way to break the authentication of a user to Sharekey.
○	$\mathcal{A}_{LOC}$	While users are authenticated by Sharekey, their identity is not verified when editing or deleting messages, as seen in Findings 9, 11, 12 and 10, therefore an adversary with access to the machine of the user could exploit these findings.
○	$\mathcal{A}_{BACK}$	Finding 1 shows, that there is no way for one participant to prove possession of a known long-term secret to another participant, as the public key material of another participant cannot be verified. The adversary $\mathcal{A}_{BACK}$ can control the public keys sent to the users. Users can only authenticate that they are interacting with other users that know a secret belonging to their public key, but without a guarantee that these public keys are correct, they cannot authenticate other users.

○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$ but for the control over the Exoscale cloud service provider.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Speaker Consistency

All participants agree on the sequence of messages sent by each participant. This means, that for all the messages sent by one specific participant, everybody agrees on the order of these messages.

●	$\mathcal{A}_{LAN}$	Since this property is also not breakable by $\mathcal{A}_{LOC}$, we think that this property cannot be broken by this adversary.
●	$\mathcal{A}_{LOC}$	No matter how the user chooses to modify the message, Sharekey will always make sure to distribute them the same to all users. Therefore it is not possible to create a sequence of messages by one user that is not displayed in the same order for all honest participants. Even if the timestamps are identical, messages will be sorted by their content.
○	$\mathcal{A}_{BACK}$	Speaker consistency does not hold for adversaries in the back-end as seen in Finding 14.
○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Causality Preserving

Implementations can avoid displaying a message before messages that causally precede it. For example in a group channel, whenever a participant sends a message, it should be guaranteed that for all participants of the group chat, this message will show up after all messages that this user has already received at the time of sending the message.

●	$\mathcal{A}_{LAN}$	The messages between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel should prevent breaking this property, as it prevents modification or insertion of any messages by $\mathcal{A}_{LOC}$.

○	$\mathcal{A}_{LOC}$	As seen in Findings 15 and 14, an adversary can control the timestamps of their messages. This allows them to create messages with timestamps in the future, with messages of honest participants that respond to that message being shown before the adversarial message with the malicious timestamp.
○	$\mathcal{A}_{BACK}$	As seen in Finding 13, an adversary could change the order of timestamps to reorder messages and break this property.
○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Global Transcript

All participants see all messages in the same order. Note that this implies speaker consistency.

●	$\mathcal{A}_{LAN}$	No matter how the user chooses to modify the message, Sharekey will always distribute the same to all users. Therefore it is not possible to create a sequence of messages that are not displayed in the same order by all honest participants.
○	$\mathcal{A}_{LOC}$	Speaker consistency is not given, consequently, there is no global transcript.
○	$\mathcal{A}_{BACK}$	Speaker consistency is not given, consequently there is no global transcript.
○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Forward Secrecy

Forward secrecy requires that a compromise of all key material does not enable the decryption of previously encrypted data.

●	$\mathcal{A}_{LAN}$	messages between the web application and Sharekey are secured with TLSv1.2+, which provides forward secrecy.

◐ $\mathcal{A}_{LOC}$ Forward secrecy is not possible due to Sharekey's business requirement of having the possibility to access all data of an account, including past and future messages, using a long-term key. This business requirement enables multi-device usage and the recovery of an account on a new device if a device (and backups) are stolen, lost, or destroyed. Sharekey is aware that this property is not provided and also clarifies that in their security claims as seen in Section 2.2.

● $\mathcal{A}_{BACK}$ Sharekey does not have any secret keys for messaging, so none can be compromised.

● $\mathcal{A}_{PROV}$ Same as $\mathcal{A}_{BACK}$

● $\mathcal{A}_{ISP}$ Same as $\mathcal{A}_{LAN}$

Post Compromise Security

Compromising all key material that this adversary can access does not enable the decryption of succeeding encrypted data.

● $\mathcal{A}_{LAN}$ The local network does not have any secret keys for messaging, so none can be compromised.

◐ $\mathcal{A}_{LOC}$ Due to the same reasoning as with the forward secrecy property, post-compromise security is not possible due to the ability to recover the state of the application through the users' passphrase, as this also allows to receive future messages on that account. Sharekey is aware that this property is not provided and also clarifies that in their security claims as seen in Section 2.2.

● $\mathcal{A}_{BACK}$ The back-end does not have any secret keys for messaging, so none can be compromised.

● $\mathcal{A}_{PROV}$ Same as $\mathcal{A}_{BACK}$

● $\mathcal{A}_{ISP}$ Same as $\mathcal{A}_{LAN}$

4.2.2 File Storage

For file storage, adversaries could have the goal to break properties like confidentiality, integrity, authentication, or file undiscoverability. These properties will be discussed in the following paragraphs. Figure 4.3 shows the diagram for this threat model, specifically including the information flow during file storage.

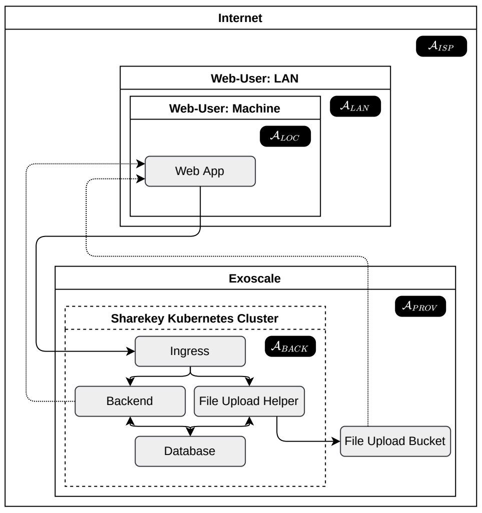


Figure 4.3: Diagram of information flow for file storage. Arrows show the flow of information between the instances. Information that originates from users is generally depicted with solid lines, while information that is going back to a user is depicted with dotted lines. These lines are semantically the same and only differentiated to improve readability.

The results of this section are summarized in Table 4.3

	Confidentiality	Integrity	Authentication	File undiscoverability
$\mathcal{A}_{LAN}$	●	●	●	●
$\mathcal{A}_{LOC}$	◐	○	○	○
$\mathcal{A}_{BACK}$	●	○	○	○
$\mathcal{A}_{PROV}$	●	○	○	○
$\mathcal{A}_{ISP}$	●	●	●	●

Table 4.3: Threat model summary for file storage.

Confidentiality

This property states, that only the intended participants can view a file. Specifically, the files must not be readable by a server operator or a user unless the file owner grants them access to view the file.

●	$\mathcal{A}_{LAN}$	Data exchanged between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel should provide confidentiality for any data sent through the channel.
◐	$\mathcal{A}_{LOC}$	The files are decrypted in the user's browser. Anyone with access to a machine with an active session will therefore also be able to view the decrypted files.
●	$\mathcal{A}_{BACK}$	File chunks and metadata are encrypted using an `xSalsa20` cipher. We believe that the keys of these ciphers stay unknown to Sharekey under the assumption of honestly distributed public keys and we did not find a way to view plaintext file contents or metadata as an adversary in control of the back-end.
●	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Integrity

No honest party accepts file contents and metadata that have not been created by someone with write permission for that file. In the current Sharekey protocol only the file owner should have permission to write a file, so integrity should imply that all files accepted by any honest participant were written by the file owner.

●	$\mathcal{A}_{LAN}$	Data exchanged between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel should provide integrity for any data sent through the channel.
○	$\mathcal{A}_{LOC}$	As seen in Finding 4, the adversary can modify a file of any user, such that integrity is not given anymore.
○	$\mathcal{A}_{BACK}$	Same as $\mathcal{A}_{LOC}$, but they can additionally change the number of chunks. Furthermore, Finding 8 shows the possibility of swapping two files that are in the same folder.

○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

Authentication

Every participant that either reads or writes a file must only be allowed to do so if proven to have knowledge of a long-term secret for that action. Writing files should only be allowed with knowledge of that file's secret signature key, reading a file should only be allowed with knowledge of that file's secret encryption key.

●	$\mathcal{A}_{LAN}$	Data exchanged between the web application and Sharekey are secured with TLSv1.2+. The attack that is possible for $\mathcal{A}_{LOC}$ cannot be performed by this adversary, as the confidentiality of the channel prevents them to learn any data signed by the file owner.
○	$\mathcal{A}_{LOC}$	Finding 4 is caused by broken authentication between users and the server. Possession of anything signed by the file owner does not prove possession of their long-term secret, since previous messages signed by the file owner can be replayed. Therefore it is not sufficient to just check for any data signed by the file owner for proper authentication.
○	$\mathcal{A}_{BACK}$	Same issue as with the adversary $\mathcal{A}_{BACK}$ for the authentication of messages.
○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

File Undiscoverability

This property ensures that the existence of objects can only be learned by their creator and the participants with whom the object is explicitly shared.

●	$\mathcal{A}_{LAN}$	The messages between the web application and Sharekey are secured with TLSv1.2+. If used properly, this communication channel should provide confidentiality for the messages sent through the channel and therefore also not allow the discovery of new files.

○	$\mathcal{A}_{LOC}$	As shown in Finding 5, participants can traverse folders to discover new files. Additionally, Finding 6 shows how they can find the files shared in any channel.
○	$\mathcal{A}_{BACK}$	As seen in Table A.2 and Table A.1, the back-end knows of the existence of every file and folder and can by querying the database learn of the existence of any file or folder in the database, as well as their relation.
○	$\mathcal{A}_{PROV}$	Same as $\mathcal{A}_{BACK}$.
●	$\mathcal{A}_{ISP}$	Same as $\mathcal{A}_{LAN}$.

4.3 Discussion

The threat models we have considered in this protocol review mainly focused on the communication between users and the Sharekey server, as well as potential attacks by malicious third parties. However, one significant factor we have not considered is the trustworthiness of the client applications that are run by the users, which are provided by Sharekey. In the first part of this section, we, therefore, discuss the delivery of the application running the protocol to the client, which was not considered in our threat models.

Another important point that one is often faced with when designing protocols and applications, is a trade-off between the usability and security of the application. In order to be widely adopted and useful, an application must be user-friendly and easy to use. However, often security and privacy measures can be cumbersome or difficult to use, which may lead to users abandoning the application or choosing not to use the security features. This is what we will discuss in the second part of this section, where we outline the usability-security-trade-off, and explain the design decisions that resulted in some properties not being fulfilled.

4.3.1 Application Delivery

Sharekey aims to allow more trust in their systems by opening their client source code. This would allow third parties trusted by the users to verify the proper implementation of the protocol. However, continuous monitoring and reviewing of the client source would take considerable resources and might not always be feasible, as every update to the client would need to be reviewed by a third party that is trusted by the user. The trust that users have to give to Sharekey is increased for their web clients. While the mobile application is reviewed by Apple or Google before being published in the respective stores, the code for the web application is directly delivered to the users by Sharekey. Every time a user loads their web client, the application is freshly delivered with no easy way of checking the application source for the users. It would also be possible for an adversary within Sharekey to only target selected users with malicious versions of the application. One possible way of establishing trust in the application received might be to have a browser extension that verifies the received JavaScript code to be a version that has previously been verified by a trusted review.

However, the issue of trusting client-side applications is not unique to Sharekey, but rather a challenge faced by virtually all providers of applications, especially web applications. This is why we did not consider this attack vector in our threat modeling. However, it is crucial to recognize that an adversary with full control of any service trusted by the user can provide a malicious application. This scenario underscores the critical importance of trust in the software delivery process and the need for robust measures to prevent such compromises.

4.3.2 Usability Security Trade-Off

Sharekey has made several design decisions that reflect the trade-off between usability and security. For example, one business requirement of Sharekey is that users must be able to log in and restore the state of their account on any new device with their passphrase and email address. In some cases, the sensitive nature of the information shared through Sharekey might require to uninstall the application while traveling through untrusted areas to avoid any compromise, and then later re-installing and recovering the state of the application again, which makes forward secrecy and post-compromise security impossible to achieve.

Another example of this trade-off is the ability to synchronize multiple devices. This is convenient for users, allowing them to transfer data from one device to another, back up the data in case one device is lost, and making it generally more accessible for users with many devices, such as the C-suite and board members that are a customer base targeted by Sharekey. In turn, this makes security properties such as speaker consistency harder to achieve, due to possible synchronization issues between different devices.

Chapter 5

Protocol Improvements

Achieving a high-security standard in a protocol also calls for formal verification of certain properties of the protocols used in addition to classic security reviews of the software and protocols. As a first step toward this end, we created a model for a simplified version of Sharekey's file upload protocol in Tamarin Prover [13–15] focusing on the file integrity property (see Section 4.2.2). Furthermore, since we were already aware of a vulnerability in the current protocol before we started our work on the model, we also designed an improved version of the protocol and modeled a simplified version of it in Tamarin Prover.

We were not able to formally verify our suggested protocol or Sharekey's implementation of it, since we simplified the protocol using several layers of abstraction for our analysis. However, our modeling still provides promising indications that the suggested updated protocol could achieve the desired properties.

5.1 Methods

This thesis suggests changes to the file upload protocol used in Sharekey, which are justified through modeling and verification with Tamarin Prover. We modeled both the current protocol and our suggestion in Tamarin, and defined security properties. Using these properties we could show how certain properties are fulfilled with our improved protocol but not in the old protocol. However, it should be noted that our Tamarin verification is not sufficient to prove the security of the suggested improvements since our modeling in Tamarin required some simplifications and only covers a subset of the complete Sharekey protocols, without conjunction with the rest of the protocols. Therefore, a proper redesign of the file upload protocol would require more time than was available for this thesis and is left for future work. Despite these limitations, the proofs still indicate that existing issues could be fixed with the suggested protocol.

5.1.1 Tamarin Prover

In this subsection, we explain how Tamarin works and how we used it to verify our proposed changes to the file upload protocol. Tamarin uses multiset rewriting rules to represent the protocols and their environments. To use Tamarin, we needed to translate the file upload algorithm from Algorithm 21 into these rules and model Sharekey's side. In Tamarin, the system's state is modeled as a multiset of facts, which we can manipulate using multiset rewriting rules. These facts can be seen as predicates storing state information. A rule can take some input facts, emit action facts, and produce output facts, as seen in the example rule of Listing 5.1. The special fact `Out(x)` models that the protocol sent out the message `x` on a public channel and the special `In(x)` fact indicates that the protocol received the message `x` on the public channel. Our setup modeled the attacker as a Dolev-Yao adversary [35], that controls the network and can delete, inject, modify, and intercept messages sent by `Out` and received from `In` facts.

The properties we want to evaluate are denoted by lemmata, which are proved using constraint solving. For an example, see Listing 5.2. Tamarin refines its constraint system about the property and the protocol until it can either conclude that the property holds in all possible cases or until it finds a counterexample to the lemma.

More complete information about Tamarin can be found in the Tamarin-Prover Manual [37].

```
rule Example_Rule:
    [ In(x), Fact1(y, z)]
    --[ Actionfact1(x, y, z), Actionfact2(x) ]->
    [Out(z), Fact2(x, y)]
```

Listing 5.1: Tamarin rule example.

```
lemma actionfacts_same_time:
"All x y z #i.
    Actionfact1(x, y, z)@i ==> Actionfact2(x)@i"
```

Listing 5.2: Tamarin lemma example.

5.1.2 Dishonest Participants

The standard communication channel in Tamarin is represented through the `Out()` and `In()` facts, but these are controlled by the adversary. To model secure channels where the adversary cannot see or control the transmitted data, one can create the facts that the adversary has no control over and use these as input in further rules. We created rules for communication from and to Sharekey that model confidential, authenticated, and integrity-protected channels. However, in some cases, we would like to argue about different adversarial capabilities such as an adversary within Sharekey's servers or a file uploader or reader. For this, we created the rules as seen in Listing 5.3, which allows adversarial control over these otherwise secured channels, as long as the dishonest party is marked as such. That way we can precisely specify which participants are modeled as being dishonest in our lemmata.

```
// Adversary learns data received by dishonest Sharekey
rule Dishonest_Sharekey_1:
    [ To_Sharekey($User,data)    ]
  --[ Dishonest_SK()             ]->
    [ Out(data)                  ]

// Adversary controls data sent by dishonest Sharekey
rule Dishonest_Sharekey_2:
    [ In(data)                   ]
  --[ Dishonest_SK()             ]->
    [ From_Sharekey($User,data)  ]

// Adversary controls data received by dishonest User
rule Dishonest_User_1:
    [ From_Sharekey($User, data) ]
  --[ Dishonest($User)           ]->
    [ Out(data)                  ]

// Adversary controls data sent by dishonest Users
rule Dishonest_User_2:
    [ In(data)                   ]
  --[ Dishonest($User)           ]->
    [ To_Sharekey($User, data)   ]
```

Listing 5.3: Modelling channels with potentially dishonest participants.

5.1.3 Simplifications

Precise modeling of the protocols in Tamarin was not in scope for this thesis and we had to make some simplifications. This subchapter describes the differences between the protocol we modeled and the one employed by Sharekey.

Firstly, we did not model the entire file system. In the Sharekey protocol, the encryption key of a file is encrypted with the encryption key of its parent folder, such that users can decrypt that file if they know the encryption key of its parent, as seen in Figure 2.3. In our model, we only encrypted the file key for the file owner, as parent folders were not modeled. We also did not model the two extra file chunks reserved for file preview and thumbnail, but these can be seen as being part of the regular chunks. Limiting our model to this subset of the entire protocol also might overlook issues that arise from its use in the big picture.

Since it is not easy to have arrays of variable length in Tamarin, we only managed to model the protocol for a fixed number of chunks, which we chose to be two in our proofs. It would be possible to prove it for a range of possible chunks, by duplicating certain rules and rewriting them with a different number of chunks, however, this would increase the number of possible states and cause slower proofs. However, the main flaws of the current protocol are still observable in two chunks and we can show that they are mitigated in our proposed protocol.

Furthermore, we limited ourselves to having just one metadata field instead of multiple. In the original protocol, there were separate fields for the file name, creation date, and last modified date which were all encrypted separately. However, we suggest adding them to the same data structure and encrypting them as one entity in the new protocol and therefore simplifying the original protocol to also only contain one metadata field. We still managed to find all the expected attacks in the current protocol and the modeled metadata reflects the suggested way of encrypting the metadata, which is why we think this simplification is justified.

Another difference is that the encryption primitive in Tamarin is deterministic and not nonce-based, unlike the one used in Sharekey. This also implies that encrypt-then-sign in Tamarin will sign the entire output of the encryption operation, whereas the encrypt-then-sign function used by Sharekey only signs the ciphertext but not the nonce. Such a missing commitment to the nonce might allow for an attack similar to the one found by Stäuble et al. [38], where changing the nonce would decrypt to a different plaintext. However, to achieve verification of the tag, they needed to change the cipher text slightly, which would be detected in signature verification. We do however still recommend including nonces with their ciphertext whenever a ciphertext is signed.

Lastly, the model assumes that a file can only be written once. Changing the metadata of files, such as renaming is not modeled. The addition of such a feature would allow replay attacks, where the metadata is overwritten by previous versions, which might be prevented in our improved model through the addition of revision numbers as part of the fingerprint and is left for future work.

While our Tamarin model's scope was limited due to these simplifications, it can still provide some indication of the effectiveness in fulfilling the desired properties in our improved model. However, a more comprehensive model and more time would be required for a proper redesign of the protocol, which is left for future work.

5.1.4 Security Properties

In Tamarin, properties are given through action facts, which are produced when a rule is used. The model of the current protocol and the improved protocol may be different in functionality, but they emit the same action facts. Thus, we can use the same Tamarin lemmata for both, allowing for a direct comparison of the properties that hold for each protocol. Tamarin either proves that there is no trace of fulfilling a lemma or if one exists, it produces a graph showing the steps taken to fulfill it in the interactive view. Studying these graphs can help understand precisely how Tamarin comes to its result.

The first lemmata we created are to ensure the correctness of our protocol. This helps to verify that the protocol works in the intended way. Otherwise, security properties might verify just because the modeled protocol is broken. First, we have functionality lemmata, for which we need to check that a trace exists that fulfills these lemmata and also analyze the produced graph to verify that it is what we expect. As an example, the functionality lemma can be seen in Listing 5.4.

```
lemma functionality_read_file:
exists-trace
"/* there exists a trace where */
Ex Owner Reader file_id metadata ptxt #i #j.
 /* A file owner can write a file */
 U_Write_File(Owner, file_id, metadata, ptxt)@i
 /* and some reader can read that file */
 & U_Read_File(Owner, Reader, file_id, metadata, ptxt)@j
 /* without any dishonest users */
 & not (Ex User #d. Dishonest(User)@d)
 /* and without dishonest Sharekey */
 & not (Ex #d. Dishonest_SK()@d)
"
```

Listing 5.4: Show that it is possible for a user to write a file and someone reading it again.

For the rest of our tests, we look at the case where a file owner uploads a file to Sharekey, and another reader receives the file key, downloads it, and reads the file. First, we aim to verify the confidentiality of the uploaded data regarding Sharekey. For both the metadata and the file contents, we aim to prove that there is no way Sharekey breaks confidentiality if users are honest and Sharekey is dishonest. The confidentiality lemma for the plaintext can be seen in Listing 5.5.

```
lemma confidentiality_ptxt_dishonest_sharekey:
"/* For any combinations of owners and files */
All Owner file_id metadata ptxt #i.
 /* when the owner writes a file */
 U_Write_File(Owner, file_id, metadata, ptxt)@i
 /* and no user is dishonest (except Sharekey) */
 & (not Ex User #d. Dishonest(User)@d)
==> (
 /* the adversary does not know the file content */
 not Ex #k. K(ptxt)@k
)"
```

Listing 5.5: Show that it is not possible for the adversary to know the contents of the file content even if Sharekey is dishonest.

Then we also aim to show the integrity of metadata or plaintext, where we argue for each case where the owner, reader, or Sharekey is dishonest. The adversary can view all incoming and outgoing data of the dishonest participants and modify what they are sending. We created tests to verify these properties when users were restricted to uploading at most one file, as well as the case where they can upload as many files as they want. This was done due to some issues we had where Tamarin would get stuck in a loop when proving a lemma, and restricting the users to one file upload would help in some cases. We justify why such a limitation is valid in Section 5.2.2. Listing 5.6 shows one such lemma where the owner is restricted to exactly one file upload and Listing 5.7 shows the same property without the restriction on the number of uploaded files.

```
lemma integrity_ptxt_dishonest_sharekey_u1:
"/* For any combination of owner reader and file */
All Owner Reader file_id metadata1 txt #i.
 /* If a user reads a file */
 U_Read_File(Owner, Reader, file_id, metadata1, ptxt) @ i
 /* and the reader is not dishonest */
 & (not Ex #d. Dishonest(Reader) @ d & d < i)
 /* and the owner is not dishonest */
 & (not Ex #d. Dishonest(Owner) @ d & d < i)
 /* and the owner is restricted to only one file upload */
 & (All #i #j.
  U_Created_File(Owner)@i & U_Created_File(Owner)@j
  ==> #i = #j)
==> (
 /* Then the plaintext read by the user must have been
    written by the owner */
 Ex metadata2 #w.
  U_Write_File(Owner, file_id, metadata2, ptxt) @ w & w <
     i
)"
```

Listing 5.6: Show that Sharekey cannot modify the content of a file if the owner can only upload one file.

```
lemma integrity_ptxt_dishonest_sharekey:
"/* For any combination of owner reader and file */
All Owner Reader file_id metadata1 ptxt #i.
```

```
 /* If a user reads a file */
 U_Read_File(Owner, Reader, file_id, metadata1, ptxt) @ i
 /* and the reader is not dishonest */
 & (not Ex #d. Dishonest(Reader) @ d & d < i)
 /* and the owner is not dishonest */
 & (not Ex #d. Dishonest(Owner) @ d & d < i)
==> (
 /* Then the plaintext read by the user must have been
    written by the owner */
 Ex metadata2 #w.
  U_Write_File(Owner, file_id, metadata2, ptxt) @ w & w <
    i
)"
```

Listing 5.7: Show that Sharekey can't modify the content of a file if the owner can upload an arbitrary number of files.

The full Tamarin files with all rules and lemmata can be viewed in this project's GitHub [34]. A list of all lemmata with descriptions is presented in Section 5.2.2.

5.2 Results

We aimed to improve the file upload protocol of Sharekey to also provide integrity over the entire file. While the original protocol provided integrity protection to each chunk, it did not protect the file as a whole. We designed a new protocol that was close to the original while containing some key improvements to provide file integrity. We initially also redesigned key distribution for files to achieve extra properties, but we could not verify our approach. Therefore, we decided to leave this for future work as seen in Section 5.4.2.

The new protocol we designed aimed to provide several features. The file would still need to be split into chunks, and there would still be a back-end for logical data about the file, such as IDs or metadata. We also maintained the need for an additional file upload helper to handle the upload buckets. Chunks needed to be uploaded separately, and each chunk needed to be verified one by one, not having to wait until the entire file was uploaded. We still wanted confidentiality regarding Sharekey, but we also wanted integrity for Sharekey and the readers of the file.

5.2.1 Suggested Protocol

Our modified protocol creates a fingerprint fp for the file that covers the file ID, file owner, encrypted and unencrypted metadata, and a hash of each chunk. This fingerprint is then signed by the signature key of the file, which, in turn, is signed by the file owner. Upon file upload, each incoming chunk can be verified by checking if its hash matches the hashes included in the fingerprint. This approach requires only two signatures, one for the signature key of the file and one for the fingerprint. This should also result in less computational cost, as a result, this method is more efficient than the initial version that required signing each chunk separately. Once a fingerprint is accepted, encrypted chunks only require hashing, which with the algorithms that are used by Sharekey uses less computation, providing a significant improvement in efficiency.

We introduce several functions to verify that the protocol is running as expected, which will abort the protocol run if not, namely

- *STORE*, which saves the second function argument in the database indexed by the first function argument and terminates the protocol if data already exists in the database.
- *LOAD*, which retrieves saved data from the database and terminates the protocol run if no data exists.
- *VERIFY-EQUAL*, which checks whether the first argument is equivalent to the second argument and aborts the protocol run otherwise.
- *VERIFY-CONTAINS*, which determines whether its second argument is contained in the first argument, aborting the protocol run if not.

Additionally, the protocol defines an upload process from the user's perspective, outlined in Algorithm 29, which interacts with the server in Algorithm 30, Algorithm 31, and Algorithm 32. In a later part of this section, we also describe our updated file-download protocol (see Algorithms 33, 35 and 34).

File upload

The previous protocol allowed for a single signature key for the file, which was the same as the user key, and lacked an integrity check for the signature key. In the new protocol, we have implemented a distinct signature key for every file and ensured encryption of the signature key for the file owner as well as metadata. The use of chunk hashes as their IDs prevents the overwriting of chunks with different data. The new chunk upload process is illustrated in Algorithm 29.

Algorithm 29 Upload file metadata and chunks

```
procedure NEW-UPLOAD-FILE($sk_{priv}$, $sk_{sig,owner}$, $meta$, $data$)
    // Generate file key
    $sk_{enc,file} \leftarrow_{\$} \{0,1\}^{256}$
    $sk_{enc,sig} \leftarrow_{\$} \{0,1\}^{256}$
    $pk_{sig,file} \leftarrow pk_{sig,owner}$

    // Encrypt file key and metadata
    $enc^{owner}_{sk_{enc,file}} \leftarrow ENCRYPT(sk_{enc,file}, sk_{priv})$
    $(ctxt_{meta}, n_{meta}) \leftarrow ENCRYPT(meta, sk_{enc,file})$

    // Obtain IDs
    $id_{file} \leftarrow INIT\text{-}UPLOAD'(id_{user})$

    // Encrypt and hash chunks
    for all $data_i \in data$ do
        $enc_{chk_i} \leftarrow ENCRYPT(data_i, sk_{enc,file})$
        $h_{chk_i} \leftarrow SHA256(enc_{chk_i})$
    end for
    $H_{chks} \leftarrow (h_{chk_0}, \ldots, h_{chk_n})$
    $h_{meta} \leftarrow SHA256((ctxt_{meta}, n_{meta}))$

    // Upload file
    $fp \leftarrow (id_{user}, id_{file}, h_{meta}, H_{chks})$
    $sig_{fp} \leftarrow SIGN(fp, sk_{sig,owner})$
    $header \leftarrow (fp, sig_{fp}, enc^{owner}_{sk_{enc,file}}, (ctxt_{meta}, n_{meta}), H_{chks}, pk_{file,sig}, sig_{pk_{sig}})$
    $UPLOAD\text{-}HEADER'(id_{user}, id_{file}, header)$
    for all $enc_{chk_i} \in (enc_{chk_0}, \ldots, enc_{chk_n})$ do
        $UPLOAD\text{-}CHUNK'(id_{user}, id_{file}, enc_{chk_i})$
    end for
end procedure
```

In the new protocol, the file ID is still determined by Sharekey, as it is a component of the fingerprint. However, this approach necessitates additional communication with the server. Rather than retrieving the file ID when sending the header, as was the case with the previous protocol, the client now receives the file ID from the server before proceeding with header and chunk uploads. Algorithm 30 outlines the steps taken by the server to select a file ID and store it in its database.

Algorithm 30 Server creates file ID

```
procedure INIT-UPLOAD'($id_{user}$)
    // Verify signature
    $pk_{sig,user} \leftarrow LOAD((id_{user}, "pk\_sig"))$

    // Generate file ID
    $id_{file} \leftarrow \{0,1\}^{256}$

    // Create DB entries
    $STORE((id_{file}, "owner"), id_{user})$

    // Return object ID
    return $id_{file}$
end procedure
```

Chunk IDs are no longer randomly generated upon receiving the file header. Instead, the chunk hashes are used for indexing purposes. The signature key for the file is verified against the owner's signature key, which may allow for sharing write access to a file in the future. Moreover, the information provided by the user is verified against the fingerprint of the file. Finally, the chunk hashes are stored in the database. This is shown in Algorithm 31.

Algorithm 31 Server receives header

```
procedure UPLOAD-HEADER'($id_{owner}, id_{file}, header$)
    $(fp, sig_{fp}, enc^{owner}_{sk_{enc,file}}, (ctxt_{meta}, n_{meta}), H_{chks}, pk_{file,sig}, sig_{pk_{sig}}) \leftarrow header$

    // Verify signatures
    $pk_{sig,owner} \leftarrow LOAD((id_{owner}, "pk\_sig"))$
    $VERIFY\text{-}SIGNATURE(pk_{file,sig}, sig_{pk_{sig}}, pk_{sig,owner})$
    $VERIFY\text{-}SIGNATURE(fp, sig_{fp}, pk_{file,sig})$

    // Verify fingerprint content
    $fp' \leftarrow (id_{owner}, id_{file}, SHA256((ctxt_{meta}, n_{meta})), H_{chks})$
    $VERIFY\text{-}EQUAL(fp, fp')$

    // Store file header
    $STORE((id_{file}, "header"), header)$
end procedure
```

As with the previous version of the protocol, chunks are received by a separate server that does not have a session with the user. However, this server shares database access with the primary server and can load the file's fingerprint. It verifies whether the uploaded chunk matches any hash in the database and aborts the protocol run if it fails. If the chunk is verified, the server stores the chunk in its provided file buckets that are designed to store large amounts of data, as outlined in Algorithm 32.

Algorithm 32 File upload helper receives chunk

```
procedure UPLOAD-CHUNK'($id_{user}, id_{file}, ctxt_{chk}$)
    // Verify hash
    $header \leftarrow LOAD((id_{file}, "header"))$
    $(fp, sig_{fp}, enc^{owner}_{sk_{enc,file}}, (ctxt_{meta}, n_{meta}), H_{chks}, pk_{file,sig}, sig_{pk_{sig}}) \leftarrow header$
    $h_{chk} \leftarrow SHA256(ctxt_{chk})$
    $VERIFY\text{-}CONTAINS(H_{chks}, h_{chk})$

    // Store in bucket
    $STORE\text{-}BUCKET(h_{chk}, ctxt_{chk})$
end procedure
```

File download

Files can still be downloaded by any user that knows the file ID and decrypted if they also know the file's encryption key $sk_{enc,file}$. The reader does however also perform integrity checks on the file, just like the back-end checks from Algorithms 31 and 32. The new protocol for file download including integrity checks can be seen in Algorithms 33, 34 and 34.

Algorithm 33 Download a file from the user's perspective (new protocol)

```
procedure DOWNLOAD-FILE'($id_{file}, sk_{enc,file}, pk_{sig,owner}$)
    // Obtain file header
    $header \leftarrow API\text{-}DOWNLOAD\text{-}HEADER'(id_{file})$
    $(fp, sig_{fp}, enc^{owner}_{sk_{enc,file}}, (ctxt_{meta}, n_{meta}), H_{chks}, pk_{file,sig}, sig_{pk_{sig}}) \leftarrow header$

    // Verify signatures
    $VERIFY\text{-}SIGNATURE(pk_{file,sig}, sig_{pk_{sig}}, pk_{sig,owner})$
    $VERIFY\text{-}SIGNATURE(fp, sig_{fp}, pk_{file,sig})$

    // Verify fingerprint content
    $fp' \leftarrow (id_{owner}, id_{file}, SHA256((ctxt_{meta}, n_{meta})), H_{chks})$
    $VERIFY\text{-}EQUAL(fp, fp')$

    // Decrypt metadata
    $meta \leftarrow DECRYPT(ctxt_{meta}, n_{meta})$

    // Download chunks and verify hashes
    for all $h_i \in H_{chks}$ do
        $enc_i \leftarrow API\text{-}DOWNLOAD\text{-}CHUNK(h_i)$
        $VERIFY\text{-}EQUAL(h_i, SHA256(enc_i)$
        $data_i \leftarrow DECRYPT(ctxt_i, n_i)$
    end for

    $N \leftarrow |ids_{chunk}|$
    return $(meta, (data_1, \ldots data_N))$
end procedure
```

Just like the original protocol, the back-end merely relays its storage to the user, with the only change that we used our *LOAD* function defined in this chapter, which will raise an error if the indexed value does not exist.

Algorithm 34 Provide a user with the encrypted file headers

```
procedure API-DOWNLOAD-HEADER'($id_{file}$)
    return $LOAD((id_{file}, "header"))$
end procedure
```

Algorithm 35 Provide a user with an encrypted chunk

```
procedure API-DOWNLOAD-CHUNK'($h_{chk}$)
    return $LOAD\text{-}BUCKET(h_{chk})$
end procedure
```

5.2.2 Formal Verification

The protocol was formally verified using Tamarin rules, as described in Section 5.1.1. Our approach aimed to stay as close as possible to the original protocol, and through the introduction of the file fingerprint, we were able to provide better integrity. The lemmata we derived in the process also revealed the issues we described in Findings 3 and 4 for the old protocol. All lemmata verify in the improved protocol, indicating that our fingerprint can provide stronger integrity guarantees. It is important to note that the layers of abstraction and simplification used in the verification process do not prove that our proposed protocol will work as intended, but this is a good first step toward ensuring a more secure protocol.

Tamarin was able to find traces of attacks on the security properties we specified. Whenever Tamarin finds such a trace, it creates a graph of it. However, due to the size of some of these graphs, we were not able to include them in this document while maintaining a readable A4 format. We have included the reasonably sized graphs of the functionality lemmata in Appendix B.1, while the graphs for the falsified properties are described in words. Readers who want to see the detailed graphs for these attacks can find them in the git repository of this thesis [34].

Functionality

These lemmata check for the correctness of the protocol and the results indicate the correctness of both the current protocol and the suggested protocol.

- `functionality_write_file`: Show that it is possible for a user to write a file if all participants are honest:
  - Current Protocol: **verified**, see Figure B.1. The owner creates its keys and Sharekey the file and chunk IDs, which are then used by the file owner to upload the file.
  - Our Protocol: **verified**, see Figure B.3. The owner creates its keys and Sharekey the file ID, which are then used by the file owner to upload the file.
- `functionality_read_file`: Show that it is possible for a user to write a file and someone reading it if all participants are honest.

  - Current Protocol: **verified**, see Figure B.1. The owner creates its keys and Sharekey the file and chunk IDs, which are then used by the file owner to upload the file. The file header is accepted by Sharekey, followed by the file chunks, for which the signatures are verified. At the same time, the owner shares the file's encryption key with the reader, who can use this in combination with the file content provided by Sharekey to obtain the decrypted metadata and chunks.
  - Our Protocol: **verified**, see Figure B.3. The owner creates its keys and Sharekey the file ID, which are then used by the file owner to upload the file. Chunk IDs are replaced with their hashed values. Sharekey verifies the signature over the file's public signature key and fingerprint. Then it receives the chunks which it hashes and compares with the fingerprint. At the same time, the owner shares the file's encryption key with the reader, who can use this in combination with the file content provided by Sharekey to obtain the decrypted metadata and chunks. The receiver checks the signature of the public signing key and the fingerprint, and they check whether the file content matches the fingerprint.

Confidentiality

We also wanted to verify, that we have the confidentiality of both the encrypted metadata and the encrypted file content towards Sharekey. Verification of both our simplified protocols gives further indications that the encrypted data is confidential towards Sharekey in the current protocol and that this property was not broken through our suggested improvement.

- `confidentiality_metadata_dishonest_sharekey`: Show that it is not possible for the adversary to know the metadata even if Sharekey is dishonest:
  - Current Protocol: **verified**
  - Our Protocol: **verified**
- `confidentiality_ptxt_dishonest_sharekey`: Show that it is not possible for the adversary to know the file content even if Sharekey is dishonest:
  - Current Protocol: **verified**
  - Our Protocol: **verified**

Integrity towards Sharekey

Now we come to the properties that we hope to gain from the suggested protocol. We can rediscover our attacks against the integrity of files through

Tamarin while showing that these attacks are no longer possible in our modeled version of the suggested protocol.

- `integrity_metadata_dishonest_sharekey_u1`: Show that if the file owner and the file reader are honest and the owner only uploads one file, the adversary cannot change the contents of the file metadata:
  - Current Protocol: **falsified**: Tamarin prover shows an attack against this property, where the metadata of the file is replaced by one of the file chunks. As both are just authenticated to the user by their signatures, they can be swapped with no cryptographic way of detecting the integrity breach.
  - Our Protocol: **verified**
- `integrity_ptxt_dishonest_sharekey_u1`: Show that if the file owner and the file reader are honest and the owner only uploads one file, the adversary cannot change the contents of the file plaintext:
  - Current Protocol: **falsified**: Tamarin prover shows an attack against this property, where it replaces the two encrypted chunks with the metadata of that file, which can be substituted with no cryptographic way of detecting the integrity breach.
  - Our Protocol: **verified**
- `integrity_metadata_dishonest_sharekey`: Show that if the file owner and the file reader are honest, the adversary cannot change the contents of the file metadata:
  - Current Protocol: **timeout**: We could not prove this property within reasonable use of Tamarin Prover and believe that it got stuck in a loop. However, the trace found in `integrity_metadata_dishonest_sharekey_u1` is also a valid trace for this lemma.
  - Our Protocol: **verified**
- `integrity_ptxt_dishonest_sharekey`: Show that if the file owner and the file reader are honest, the adversary cannot change the contents of the file plaintext:
  - Current Protocol: **timeout**: We could not prove this property within a reasonable time using Tamarin Prover and believe that it got stuck in a loop. However, the trace found in `integrity_ptxt_dishonest_sharekey_u1` is also a valid trace for this lemma.
  - Our Protocol: **verified**

Integrity towards readers

Similarly, we could also find attacks on the file's integrity by file readers. Following our findings, Tamarin could only find attacks on the integrity of the plaintext, but not the metadata, as the metadata can only be written once. While a reader could come up with some metadata that would break its integrity, it cannot distribute it to other users through an honest Sharekey, as Sharekey does not allow overwriting the metadata and the reader can only obtain the file id once the metadata has been uploaded. These lemmata show the authentication of the user when uploading a file, as a user that can maliciously overwrite file chunks to break the integrity of the file, also manages to break the authentication of the file uploader towards Sharekey.

- `integrity_metadata_dishonest_reader_u1`: Show that if Sharekey and the file owner are honest and the owner only uploads one file, the adversary cannot change the contents of the file metadata:
  - Current Protocol: **verified**
  - Our Protocol: **verified**
- `integrity_ptxt_dishonest_reader_u1`: Show that if Sharekey and the file owner are honest and the owner only uploads one file, the adversary cannot change the contents of the file plaintext:
  - Current Protocol: **falsified**: Tamarin prover shows an attack against this property, where the attacker overwrites the second chunk with the contents of the first chunk.
  - Our Protocol: **verified**
- `integrity_metadata_dishonest_reader`: Show that if Sharekey and the file owner are honest, the adversary cannot change the contents of the file metadata:
  - Current Protocol: **verified**
  - Our Protocol: **verified**
- `integrity_ptxt_dishonest_reader`: Show that if Sharekey and the file owner are honest, the adversary cannot change the contents of the file plaintext:
  - Current Protocol: **timeout**: We could not prove this property within a reasonable time using Tamarin Prover and believe that it got stuck in a loop. However, the trace found in `integrity_ptxt_dishonest_reader_u1` is also a valid trace for this lemma.
  - Our Protocol: **verified**

5.3 Discussion

We managed to improve the current protocol and introduce better integrity properties for file upload that we can support through modeling with the Tamarin Prover. Due to time constraints, we had to forgo formal verification of the exact file upload protocol and made some simplifications in modeling it in Tamarin Prover. Moreover, we also modeled our proposed improvement with these simplifications. However, despite the modeling simplifications, we believe that the results of the symbolic analysis using Tamarin Prover show that the proposed improvements to our protocol are in the right direction. Although our results serve as a positive indication that the proposed improvements to our protocol show a promising way to address the integrity concerns, the resulting proofs are only applicable to the simplified version. Further investigations are required to discover what guarantees can be proven for the actual non-simplified protocol. This undertaking is complex and time-consuming, which is why we did not manage to complete it in its entirety within the scope of this thesis.

5.4 Open Issues

Several issues could not yet be considered in this thesis. This section offers a non-exhaustive list of topics that might need further addressing.

5.4.1 File sharing replay protection

The current approach for file sharing used by Sharekey, as well as our suggested protocol, is relying on the fact that a file can only be written once. If we allowed files to be changed or overwritten, which would include changes to the metadata such as renaming, the current protocol is not protected against replay attacks. Such attacks might be mitigated through an additional revision number as part of the fingerprint, which is unique for every version of the file. When a new file is created, the file owner could put the revision number 0. Whenever anyone wants to edit the file, they could first request Sharekey to provide them with the next revision number and include that into the fingerprint. Sharekey could verify that updates to the file contain the correct revision number. Clients could verify that they only accept updates to a file that they have already read if the revision number is strictly larger than the last accepted revision.

5.4.2 File sharing key hierarchy

We had initially planned to implement a more advanced file-sharing protocol than what was presented in this thesis. One key aspect of our redesign involved overhauling the key hierarchy for folders and files to accommodate Sharekey's plans to expand collaboration and access control capabilities. Currently, files and folders are signed by the object owner, but we proposed a new key hierarchy that would enable the sharing of both read and write rights to entire folders.

To achieve this goal, we devised a key hierarchy (depicted in Figure 5.1) in which each object would have a secret for writing and a secret for reading ($secret_{write}$ and $secret_{read}$, respectively). These keys would then be used to derive the encryption or signature key of that object. Additionally, each object would encrypt its read secret with a key derived from its parent's read secret and its write secret with a key derived from its parent's write secret. This would allow anyone with knowledge of a folder's read or write secret to derive the same secret of all its children.

By employing such an approach, it might be possible to enforce read and write access rights cryptographically. To grant read or write access, the corresponding secret could be encrypted with a newly generated secret encryption key sk_{share}, which would further encrypt the secret. This key sk_{share} could be disseminated via Sharekey's messaging channels. Revoking access would entail changing the read or write secret, refreshing signatures or encryption of the object and all its descendants. The new key would have to be encrypted with the key derived from the parent's corresponding secret, and for all users with whom the object was shared, except for the revoked ones.

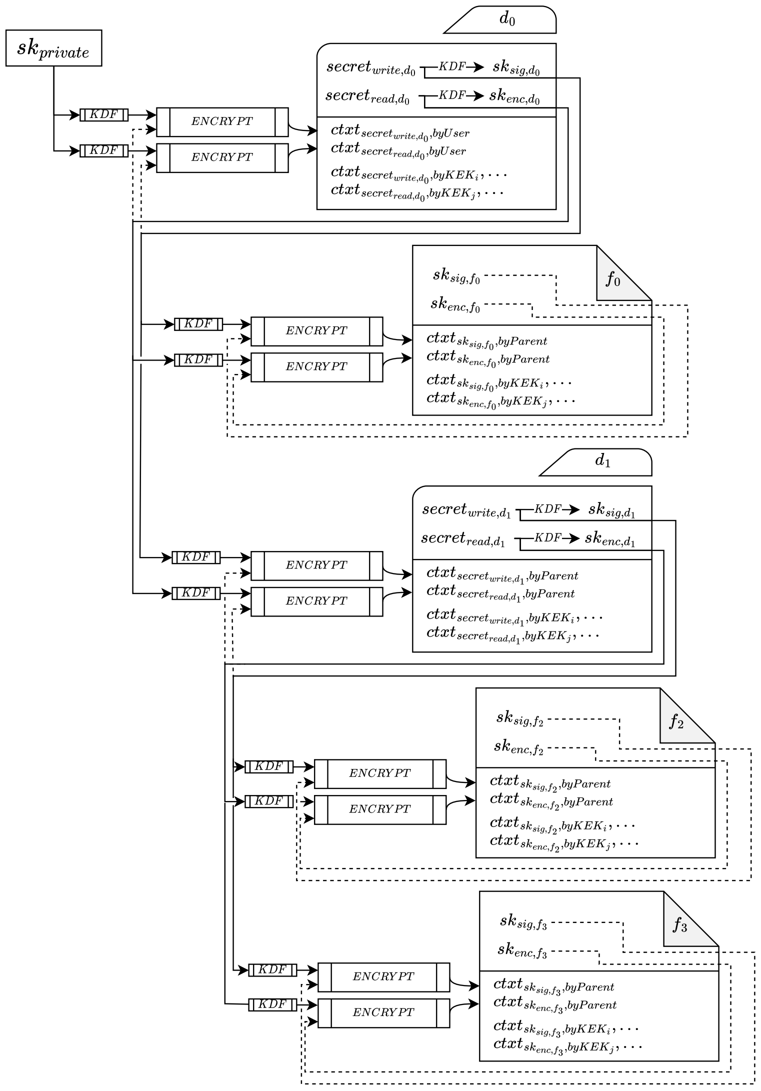
Unfortunately, we could not provide a more concrete definition of the algorithms or prove their security in conjunction with our proposed enhancement. This lack of analysis entails that this concept does not come with any guarantees and is presented here solely as a starting point for the future development of Sharekey's file upload protocol.

5.5 Further Development

In this section, we assess the possible ways of further developing the Sharekey protocol in light of the issues that have been identified. Despite the use of a robust cryptographic library, building a secure protocol on top of it remains challenging. We present three possible means for continuing the development of the protocol, each with its own set of challenges, advantages, and disadvantages.

5.5.1 Iterative improvement

One approach to address the issues with the Sharekey protocol is to undertake iterative improvements that aim to fix all the identified issues step by step. This approach would involve testing the effectiveness of the fixes and



$sk_{private}$
d_0
$secret_{write,d_0}$ KDF sk_{sig,d_0}
$secret_{read,d_0}$ KDF sk_{enc,d_0}
KDF
ENCRYPT
$ctxt_{secret_{write,d_0},byUser}$
$ctxt_{secret_{read,d_0},byUser}$
$ctxt_{secret_{write,d_0},byKEK_i},\cdots$
$ctxt_{secret_{read,d_0},byKEK_j},\cdots$
f_0
sk_{sig,f_0}
sk_{enc,f_0}
$ctxt_{sk_{sig,f_0},byParent}$
$ctxt_{sk_{enc,f_0},byParent}$
$ctxt_{sk_{sig,f_0},byKEK_i},\cdots$
$ctxt_{sk_{enc,f_0},byKEK_j},\cdots$
d_1
$secret_{write,d_1}$ KDF sk_{sig,d_1}
$secret_{read,d_1}$ KDF sk_{enc,d_1}
$ctxt_{secret_{write,d_1},byParent}$
$ctxt_{secret_{read,d_1},byParent}$
$ctxt_{secret_{write,d_1},byKEK_i},\cdots$
$ctxt_{secret_{read,d_1},byKEK_j},\cdots$
f_2
sk_{sig,f_2}
sk_{enc,f_2}
$ctxt_{sk_{sig,f_2},byParent}$
$ctxt_{sk_{enc,f_2},byParent}$
$ctxt_{sk_{sig,f_2},byKEK_i},\cdots$
$ctxt_{sk_{enc,f_2},byKEK_j},\cdots$
f_3
sk_{sig,f_3}
sk_{enc,f_3}
$ctxt_{sk_{sig,f_3},byParent}$
$ctxt_{sk_{enc,f_3},byParent}$
$ctxt_{sk_{sig,f_3},byKEK_i},\cdots$
$ctxt_{sk_{enc,f_3},byKEK_j},\cdots$

Figure 5.1: Diagram of a possible update of the key hierarchy. The KDFs are expected to be seeded differently for each object, for example, $"Parent" + id_{parent} + "reads" + id_{child}$. In addition to the keys encrypted for the parent folder, we suggest having further fields for keys encrypted by any KEK, such as KEKs shared in channels to provide other users access to read or write to that object.

repeating the process of fixing the rediscovered and new findings after six months. This would require minimal changes in the development process of Sharekey and would allow them to keep most of the existing changes.

However, such a "fix-break" cycle is not an efficient or secure way of designing a protocol. The process would likely necessitate migrations of their database after each cycle, which would become increasingly difficult as the user base grows. Additionally, they would likely be left with legacy features that need to be maintained, leading to the accumulation of complexity over time, which might again hinder their security.

5.5.2 Redesign

Another option is to take a step back, assess the requirements, and redesign a new protocol from scratch. This approach would enable the development of a secure protocol with provable security from the outset, which could potentially cover all the needs of Sharekey.

However, such a process is time-intensive and requires substantial expertise, which may exceed the resources available to Sharekey to undertake independently. To design a provably secure protocol, they would require external institutions to provide funding and expertise. While designs such as TLSv1.3 demonstrate the possibility of such a design process, the amount of time and resources required is significant. In any case, the slow process of such an approach would most likely mean that Sharekey continues with iterative improvement at first, which will make a later change more difficult.

5.5.3 Integration of existing protocols

An alternative approach to building a secure protocol from scratch is to leverage existing open protocols, such as Signal [23] for messaging and NextCloud [39] for file sharing, to build Sharekey's application. Overall, there would be less security responsibility with Sharekey and they could achieve faster development of new features while also being able to focus on other aspects of the protocol. There are open-source libraries that implement these protocols which could be used and would be maintained by other parties. Sharekey could take advantage of these libraries, thereby reducing the amount of work required to build and maintain the protocol. Their business requirements could still be achieved by building on top of the other protocol, for example, multi-device usage could be done by using Signal's forward secure protocol but backing up the messages, allowing to restore the state on new devices.

However, it is important to carefully consider the potential vulnerabilities that could arise from combining multiple protocols. Moreover, this approach could introduce additional complexity to the system and increase dependencies on external parties. While open-source libraries that implement these protocols could be utilized and maintained by other parties, the integration process should be performed with caution to ensure the security of the entire system. Furthermore, licensing considerations should also be taken into account when utilizing existing protocols.

Chapter 6

Conclusion

Sharekey presents itself as a secure app, with great measures to ensure the security and privacy of its users. It is marketed towards subjects who require confidentiality, such as high-level business executives or government officials, which makes it imperative to review Sharekey's security claims thoroughly. This thesis contributes an extensive analysis of the Sharekey protocol, identifying several security weaknesses, performing a thorough review with threat modeling, and providing possible remediations and proposed updates to their protocol to improve the general security of Sharekey.

We reviewed and documented their current protocol and found 19 security issues in their protocol and implementation, which could have allowed adversarial users to significantly hinder the business continuity of Sharekey's customers. We have suggested possible remediations for these findings and Sharekey is putting a lot of effort into fixing the problems we identified. While we found that we cannot fully agree with all security claims that were brought forward by Sharekey, our results did not contradict Sharekeys main claims that the data encrypted by the users remains confidential towards Sharekey, provided their customers trust them to distribute public keys honestly. However, we did find issues contradicting their claims towards authentication and integrity of the data shared through Sharekey and we feel like our interpretation of some claims in regards to the collected metadata contradicts the actual extent of metadata available to Sharekey.

In addition, this thesis started the effort of redesigning the file-sharing protocol, presenting a way of fingerprinting files that requires fewer signatures than the current protocol while providing better integrity of the encrypted files, as supported through modeling of the protocol and the desired properties in the Tamarin Prover. The redesign is not complete and we hope to mark the start of a thorough and comprehensive design process, empowering Sharekey to improve its security and functionality while retaining a good user experience.

While our analysis revealed several security issues in Sharekey's protocol and implementation, we are encouraged by the company's commitment to addressing these concerns and improving the security of its platform. We are grateful for their collaboration and transparency throughout this process. As Sharekey continues to evolve, we hope that its protocol design process will continue to adapt and empower them to produce robust and secure solutions. We believe in Sharekey's vision of secure communication and the importance of solving this critical problem, and we look forward to seeing the company succeed in achieving its goals.

Appendix A

Tables

A.1 Database entries

Field	**Description**
_id	Random folder id
owner	Identifier of owner
$(sk_{enc,byUser}, sk_{enc,byFolder})$	Encrypted secret keys
pk_{sig}	Public signature key of owner
is-public	Currently unused
parent-folder	Identifier of parent folder
ancestors	Currently just parent folder
$name_{enc}$	Encrypted folder name
$created_{enc}$	Encrypted creation timestamp
$modified_{enc}$	Encrypted last modified timestamp
shared-in-channels	List of channels the folder is shared in
is-shared	If folder is shared in any channel
$id_{external\text{-}link}$	Identifier to external link if it exists
external-link-availability	Currently unused
items-amount	The number of children
revision	How many changes have been done on this document

Table A.1: DB entry for a folder.

Field	Description
_id	Random file ID
owner	Identifier of owner
$(enc^{owner}_{sk_{enc,file}}, enc^{parent}_{sk_{enc,file}})$	Encrypted secret keys
pk_{sig}	Public Signature key of owner
is-public	Currently unused
uploaded-for-sharing	Currently unused
$meta_{enc}$	Encrypted file name, creation and last modified date
$MIMEtype$	MIME-Type of file
parent-folder	Identifier of parent folder
shared-in-channels	List of channels the file is shared in
file-size	File size in bytes
$chunks_{file} : \{(id_0, offset_0, n_0, sig_0), \ldots, (id_n, offset_n, n_n, sig_n)\}$	Where to find the actual encrypted file data
uploading-finished$_{file}$	If all chunks were uploaded
$chunk_{preview} : (id, offset, n, sig)$	Where to find the encrypted preview
uploading-finished$_{preview}$	Wether preview chunk finished uploading
$chunk_{thumbnail_{64}} : (id, offset, n, sig)$	Where to find the encrypted 64x64 thumbnail
uploading-finished$_{thumbnail_{64}}$	Wether thumbnail chunk finished uploading
$chunk_{thumbnail_{128}} : (id, offset, n, sig)$	Where to find the encrypted 128x128 thumbnail
uploading-finished$_{thumbnail_{128}}$	Wether thumbnail chunk finished uploading
is-hidden	Hides file in directory
is-shared	if file is shared in any channel
ts_{upload}	Time of upload
upload-finished	If upload has finished
revision	How many changes have been done on this document

Table A.2: DB entry for a file.

A.2 Fuzzed API Endpoints

- `PersonalInvitationCode.create`
- `PersonalInvitationCode.sendEmails`

- `SAFE_ROOT_FOLDER`
- `auth.generateToken`
- `auth.removeToken`
- `auth.requestLoginAuthCode`
- `callSignals`
- `calls.checkCallsInQueue`
- `calls.dropCall`
- `calls.initiate`
- `externalLinks.internal.create`
- `externalLinks.internal.getForOwner`
- `externalLinks.internal.remove`
- `login`
- `messages.channel.getShared`
- `messages.currentUserChannels`
- `messages.delete`
- `messages.deleteWithAttachments`
- `messages.getForChannel`
- `messages.getMessagesViaIdsList`
- `messages.read`
- `messages.send`
- `messages.sendWithAttachments`
- `messages.userDrafts`
- `meteor.loginServiceConfiguration`
- `meteor_autoupdate_clientVersions`
- `notifications.channels.state.set`
- `notifications.update`
- `notifications.v2.byId`
- `people.addToContacts`
- `people.currentUserContacts`
- `people.get20FromAllUsers`

- `people.search`
- `random.data.get`
- `rights.byIds`
- `safe.favoriteItems`
- `safe.favorites`
- `safe.favorites.mark`
- `safe.favorites.unmark`
- `safe.folder.copy`
- `safe.folder.create`
- `safe.folder.get.keys`
- `safe.folder.get.size`
- `safe.folder.rename`
- `safe.folder.update.copied`
- `safe.folderContent.byId`
- `safe.get.folders_with_rights`
- `safe.get.safe_iems_with_rights`
- `safe.get.users_with_access`
- `safe.rights.revokeAccess`
- `safe.upload.encryptedChunks`

Appendix B

Tamarin

B.1 Tamarin Graphs

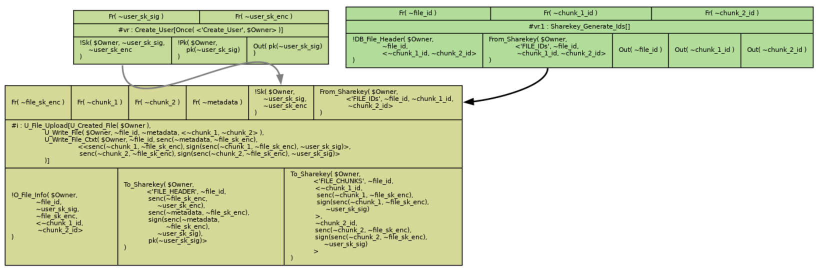

Fr(~user_sk_sig)
Fr(~user_sk_enc)
#vr : Create_User[Once(<'Create_User', $Owner>)]
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
!Pk($Owner, pk(~user_sk_sig))
Out(pk(~user_sk_sig))
Fr(~file_id)
Fr(~chunk_1_id)
Fr(~chunk_2_id)
#vr.1 : Sharekey_Generate_Ids[]
!DB_File_Header($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>)
From_Sharekey($Owner, <'FILE_IDs', ~file_id, ~chunk_1_id, ~chunk_2_id>)
Out(~file_id)
Out(~chunk_1_id)
Out(~chunk_2_id)
Fr(~file_sk_enc)
Fr(~chunk_1)
Fr(~chunk_2)
Fr(~metadata)
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
From_Sharekey($Owner, <'FILE_IDs', ~file_id, ~chunk_1_id, ~chunk_2_id>)
#i : U_File_Upload[U_Created_File($Owner), U_Write_File($Owner, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Write_File_Ctxt($Owner, ~file_id, senc(~metadata, ~file_sk_enc), <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)]
!O_File_Info($Owner, ~file_id, ~user_sk_sig, ~file_sk_enc, <~chunk_1_id, ~chunk_2_id>)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig)>)
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, <~chunk_1_id, senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig) >, ~chunk_2_id, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig) >)

Figure B.1: Tamarin trace for the lemma `functionality_write_file` of the current protocol.

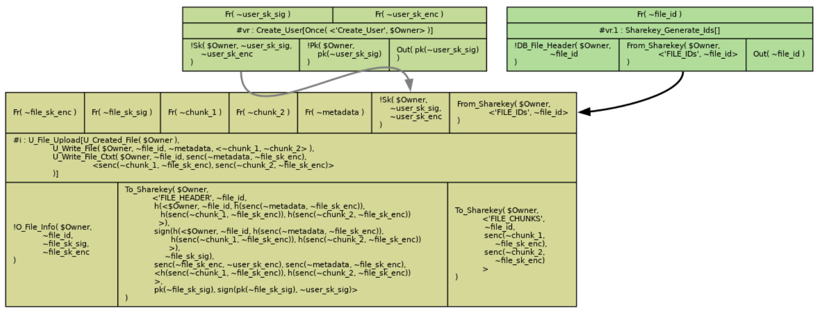

Fr(~user_sk_sig)
Fr(~user_sk_enc)
#vr : Create_User[Once(<'Create_User', $Owner>)]
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
!Pk($Owner, pk(~user_sk_sig))
Out(pk(~user_sk_sig))
Fr(~file_id)
#vr.1 : Sharekey_Generate_Ids[]
!DB_File_Header($Owner, ~file_id)
From_Sharekey($Owner, <'FILE_IDs', ~file_id>)
Out(~file_id)
Fr(~file_sk_enc)
Fr(~file_sk_sig)
Fr(~chunk_1)
Fr(~chunk_2)
Fr(~metadata)
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
From_Sharekey($Owner, <'FILE_IDs', ~file_id>)
#i : U_File_Upload[U_Created_File($Owner), U_Write_File($Owner, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Write_File_Ctxt($Owner, ~file_id, senc(~metadata, ~file_sk_enc), <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)]
!O_File_Info($Owner, ~file_id, ~file_sk_sig, ~file_sk_enc)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig)>)
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc) >)

Figure B.2: Tamarin trace for the lemma `functionality_write_file` of the proposed protocol.

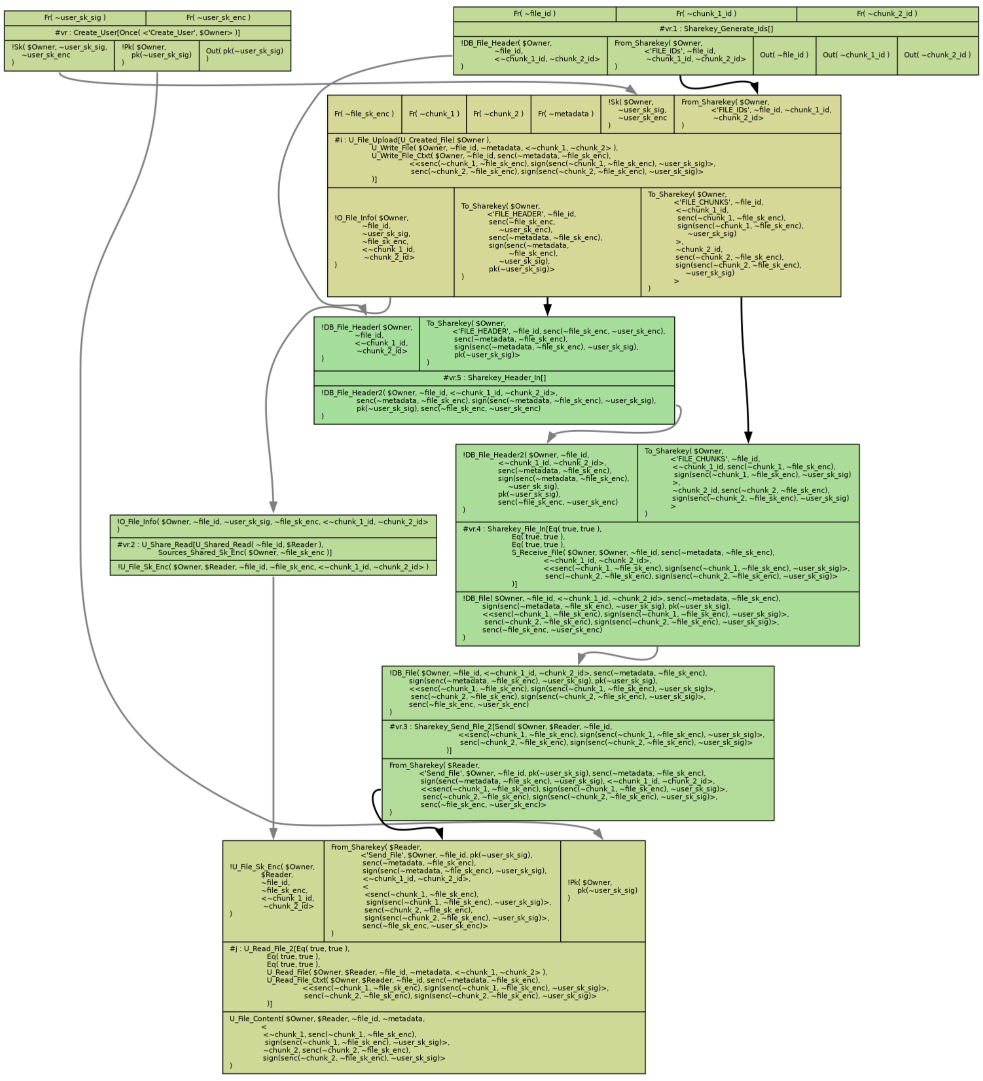

Fr(~user_sk_sig)
Fr(~user_sk_enc)
#vr : Create_User[Once(<'Create_User', $Owner>)]
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
!Pk($Owner, pk(~user_sk_sig))
Out(pk(~user_sk_sig))
Fr(~file_id)
Fr(~chunk_1_id)
Fr(~chunk_2_id)
#vr.1 : Sharekey_Generate_Ids[]
!DB_File_Header($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>)
From_Sharekey($Owner, <'FILE_IDs', ~file_id, ~chunk_1_id, ~chunk_2_id>)
Out(~file_id)
Out(~chunk_1_id)
Out(~chunk_2_id)
Fr(~file_sk_enc)
Fr(~chunk_1)
Fr(~chunk_2)
Fr(~metadata)
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
From_Sharekey($Owner, <'FILE_IDs', ~file_id, ~chunk_1_id, ~chunk_2_id>)
#i : U_File_Upload[U_Created_File($Owner), U_Write_File($Owner, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Write_File_Ctxt($Owner, ~file_id, senc(~metadata, ~file_sk_enc), <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)]
!O_File_Info($Owner, ~file_id, ~user_sk_sig, ~file_sk_enc, <~chunk_1_id, ~chunk_2_id>)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig)>)
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, <~chunk_1_id, senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig) >, ~chunk_2_id, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig) >)
!DB_File_Header($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig)>)
#vr.5 : Sharekey_Header_In[]
!DB_File_Header2($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>, senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig), senc(~file_sk_enc, ~user_sk_enc))
!DB_File_Header2($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>, senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig), senc(~file_sk_enc, ~user_sk_enc))
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, <~chunk_1_id, senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig) >, ~chunk_2_id, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig) >)
#vr.4 : Sharekey_File_In[Eq(true, true), Eq(true, true), Eq(true, true), S_Receive_File($Owner, $Owner, ~file_id, senc(~metadata, ~file_sk_enc), <~chunk_1_id, ~chunk_2_id>, <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)]
!DB_File($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>, senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig), <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>, senc(~file_sk_enc, ~user_sk_enc))
!O_File_Info($Owner, ~file_id, ~user_sk_sig, ~file_sk_enc, <~chunk_1_id, ~chunk_2_id>)
#vr.2 : U_Share_Read[U_Shared_Read(~file_id, $Reader), Sources_Shared_Sk_Enc($Owner, ~file_sk_enc)]
!U_File_Sk_Enc($Owner, $Reader, ~file_id, ~file_sk_enc, <~chunk_1_id, ~chunk_2_id>)
!DB_File($Owner, ~file_id, <~chunk_1_id, ~chunk_2_id>, senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), pk(~user_sk_sig), <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>, senc(~file_sk_enc, ~user_sk_enc))
#vr.3 : Sharekey_Send_File_2[Send($Owner, $Reader, ~file_id, <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)]
From_Sharekey($Reader, <'Send_File', $Owner, ~file_id, pk(~user_sk_sig), senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), <~chunk_1_id, ~chunk_2_id>, <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>, senc(~file_sk_enc, ~user_sk_enc)>)
!U_File_Sk_Enc($Owner, $Reader, ~file_id, ~file_sk_enc, <~chunk_1_id, ~chunk_2_id>)
From_Sharekey($Reader, <'Send_File', $Owner, ~file_id, pk(~user_sk_sig), senc(~metadata, ~file_sk_enc), sign(senc(~metadata, ~file_sk_enc), ~user_sk_sig), <~chunk_1_id, ~chunk_2_id>, < <senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>, senc(~file_sk_enc, ~user_sk_enc)>)
!Pk($Owner, pk(~user_sk_sig))
#j : U_Read_File_2[Eq(true, true), Eq(true, true), Eq(true, true), U_Read_File($Owner, $Reader, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Read_File_Ctxt($Owner, $Reader, ~file_id, senc(~metadata, ~file_sk_enc), <<senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)]
U_File_Content($Owner, $Reader, ~file_id, ~metadata, < <~chunk_1, senc(~chunk_1, ~file_sk_enc), sign(senc(~chunk_1, ~file_sk_enc), ~user_sk_sig)>, ~chunk_2, senc(~chunk_2, ~file_sk_enc), sign(senc(~chunk_2, ~file_sk_enc), ~user_sk_sig)>)

Figure B.3: Tamarin trace for the lemma `functionality_read_file` of the current protocol.

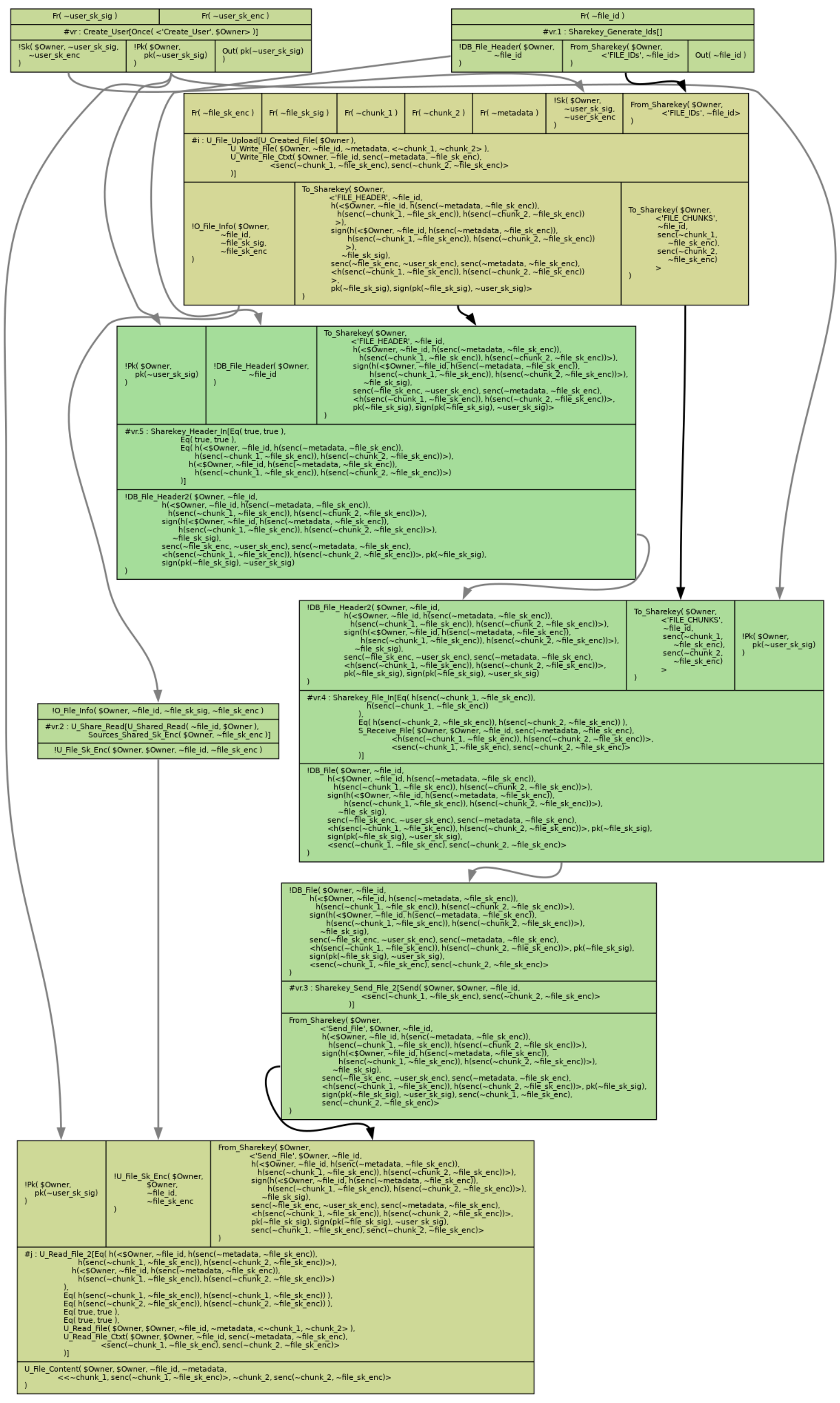

Fr(~user_sk_sig)
Fr(~user_sk_enc)
#vr : Create_User[Once(<'Create_User', $Owner>)]
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
!Pk($Owner, pk(~user_sk_sig))
Out(pk(~user_sk_sig))
Fr(~file_id)
#vr.1 : Sharekey_Generate_Ids[]
!DB_File_Header($Owner, ~file_id)
From_Sharekey($Owner, <'FILE_IDs', ~file_id>)
Out(~file_id)
Fr(~file_sk_enc)
Fr(~file_sk_sig)
Fr(~chunk_1)
Fr(~chunk_2)
Fr(~metadata)
!Sk($Owner, ~user_sk_sig, ~user_sk_enc)
From_Sharekey($Owner, <'FILE_IDs', ~file_id>)
#i : U_File_Upload[U_Created_File($Owner), U_Write_File($Owner, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Write_File_Ctxt($Owner, ~file_id, senc(~metadata, ~file_sk_enc), <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)]
!O_File_Info($Owner, ~file_id, ~file_sk_sig, ~file_sk_enc)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc)) >, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig)>)
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc) >)
!Pk($Owner, pk(~user_sk_sig))
!DB_File_Header($Owner, ~file_id)
To_Sharekey($Owner, <'FILE_HEADER', ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig)>)
#vr.5 : Sharekey_Header_In[Eq(true, true), Eq(true, true), Eq(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>))]
!DB_File_Header2($Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig))
!DB_File_Header2($Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig))
To_Sharekey($Owner, <'FILE_CHUNKS', ~file_id, senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc) >)
!Pk($Owner, pk(~user_sk_sig))
#vr.4 : Sharekey_File_In[Eq(h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc))), Eq(h(senc(~chunk_2, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))), S_Receive_File($Owner, $Owner, ~file_id, senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)]
!DB_File($Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig), <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)
!O_File_Info($Owner, ~file_id, ~file_sk_sig, ~file_sk_enc)
#vr.2 : U_Share_Read[U_Shared_Read(~file_id, $Owner), Sources_Shared_Sk_Enc($Owner, ~file_sk_enc)]
!U_File_Sk_Enc($Owner, $Owner, ~file_id, ~file_sk_enc)
!DB_File($Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig), <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)
#vr.3 : Sharekey_Send_File_2[Send($Owner, $Owner, ~file_id, <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)]
From_Sharekey($Owner, <'Send_File', $Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig), senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)
!Pk($Owner, pk(~user_sk_sig))
!U_File_Sk_Enc($Owner, $Owner, ~file_id, ~file_sk_enc)
From_Sharekey($Owner, <'Send_File', $Owner, ~file_id, h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), sign(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), ~file_sk_sig), senc(~file_sk_enc, ~user_sk_enc), senc(~metadata, ~file_sk_enc), <h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>, pk(~file_sk_sig), sign(pk(~file_sk_sig), ~user_sk_sig), senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)
#j : U_Read_File_2[Eq(h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>), h(<$Owner, ~file_id, h(senc(~metadata, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))>)), Eq(h(senc(~chunk_1, ~file_sk_enc)), h(senc(~chunk_1, ~file_sk_enc))), Eq(h(senc(~chunk_2, ~file_sk_enc)), h(senc(~chunk_2, ~file_sk_enc))), Eq(true, true), Eq(true, true), U_Read_File($Owner, $Owner, ~file_id, ~metadata, <~chunk_1, ~chunk_2>), U_Read_File_Ctxt($Owner, $Owner, ~file_id, senc(~metadata, ~file_sk_enc), <senc(~chunk_1, ~file_sk_enc), senc(~chunk_2, ~file_sk_enc)>)]
U_File_Content($Owner, $Owner, ~file_id, ~metadata, <<~chunk_1, senc(~chunk_1, ~file_sk_enc)>, ~chunk_2, senc(~chunk_2, ~file_sk_enc)>)

Figure B.4: Tamarin trace for the lemma `functionality_read_file` of the proposed protocol.

Appendix C

Python Client

C.1 CLI Functions

Besides diretly using the client API in a Python Shell to test our exploits, we have collected some exploits into a CLI with the following attacks

- **View Message Details**: This function lets an attacker choose a channel that they are a member of and then a message by any user within that channel for, which all information known to the client was displayed. This was useful for debugging, showing the JSON format of the different message types (text, image, voice message) and to display message signatures.
- **Edit a Message**: This function lets an attacker choose a channel that they are a member of and then a message by any user within that channel, which they can edit to contain any text they want.
- **Delete a Message**: This function lets an attacker choose a channel that they are a member of and then a message by any user within that channel, which they can delete for everyone in the channel.
- **Email Info**: This function lets an attacker know for any E-Mail address if it belongs to a Sharekey account and displays the cryptographic keys of that user if it exists.
- **Break app**: This function lets and attacker choose a channel that they are a member of and then sends a message into that chat which breaks the web client when viewed.
- **Recover File Tree**: This function lists all files that were ever shared with the attacker and will for each file traverse its file tree to find other files by the same owner.

C.1.1 Example Execution

```
$ python3 main.py

Choose a function:
  0: View Message Details
  1: Edit a Message
  2: Delete a Message
  3: Email Info
  4: Break app
  5: Recover File Tree
  6: Quit
1

Choose a Channel:
  0: [D]: Alice
  1: [G]: Group Channel (Alice, Bob)
1

Choose a Message:
  0: Alice (c57RwHoLPAataxqRWZysJq4W)
    Hi, I'm Alice
  1: Bob (Pr3iCkp7Y2F5EXNastkzPEKS)
    Hi, I'm Bob
  2: Alice (cwjubtW2D6XGRpJX82MXiDH9)
    How have you been, Bob?
2

Please enter the new message text:
I would like to profess, that Mallory is my best friend
    for ever

{'data': {'message': 'successfully edited', 'status':
    200},
 'updatedChannelId': 'qfguGkH326ZXPRC45'}
```

Listing C.1: Example execution of CLI for our custom python client to edit a message. The attacker can first choose the action it wants to take and they choose to edit a message. They can then choose one of the channels that they are a member of in which they want to edit a message, then specify which exact message to edit and finally write specify the new message content. Sharekey will then display this message instead of the original message for all users of that channel.

C.2 Client API

- `login_keyring`: Logs in to an account with the provided credentials
- `sign`: Signs a message with the given signing key
- `encrypt`: Encrypts a message with the givven encryption key
- `decrypt`: Decrypts a message with the given encryption key

- `encrypt_then_sign`: Encrypts then signs a message with the given encryption and singature key
- `get_user_by_email_hash`: Queries the back-end for a given email hash
- `get_user_by_id`: Gets information about a user given its user id
- `get_user_by_email`: Hashes an email address and queries the back-end for that address
- `get_user_channels`: Obtains a list of channels of the current user, decrypting any data that the user knows the key of
- `get_user_contacts`: Obtains a list of al contacts of the current user
- `get_messages`: Obtains a list of all messages within a channel, decrypting them with the channel key
- `channel_key`: Obtains the channel key from a channel object by decrypting the right encrypted key
- `decrypt_channel`: Obtains the channel key and decrypts all decryptable information about it
- `send_message`: Sends a message to a given channel, with the possibility to invalidate signatures or ciphertexts of the message and timestamp.
- `edit_message`: Sends a given message, inserting a new JSON instead
- `delete_message`: Deletes a given message
- `sendInvite`: Send an invitation E-Mail to the given email list
- `get_user_by_invitation`: Get a list of users by invitation
- `change_name`: Changes the name of the current user
- `spam_mail`: Sends a mail address to a list of users, with any message, adding the HTML escape code to remove indicators that the E-Mail was sent through Sharekey
- `search_people`: Searches people by Regular expression
- `get_all_public`: Returns a list of all people with public profiles
- `upload_chunk`: Uploads a chunk given a file id and a chunk id, signing it with the user's signature key
- `upload_file`: Uploads a new file to sharekey
- `reverse_chunks`: Given a file id reverses the chunks of that file
- `delete_files`: Deletes a list of files or folders owned by the user
- `get_root_folder`: Obtain the root folder of that user

- `get_folder`: Obtain a folder by folder id, decrypting its fields if the secret key is known
- `get_object`: Return a file file by ID
- `get_folders_with_rights`: Return a folder with rights by ID
- `get_tree`: Recursively list all children of any folder and its subfolders etc
- `get_root_tree`: List all files owned by the user

Bibliography

[1] *Microsoft Teams Revenue and Usage Statistics (2023)*. Business of Apps. URL: https://www.businessofapps.com/data/microsoft-teams-statistics/ (visited on 04/10/2023).

[2] *Signal Revenue & Usage Statistics (2023)*. Business of Apps. URL: https://www.businessofapps.com/data/signal-statistics/ (visited on 04/10/2023).

[3] *Zoom Revenue and Usage Statistics (2023)*. Business of Apps. URL: https://www.businessofapps.com/data/zoom-statistics/ (visited on 04/10/2023).

[4] Plumely, Mike and Gorzelany Andres Mariano and Davis, Chris and Jupudi, Alekhya and Buck, Alex and Rinas, Martin and Payne, Heidi and Soysal, Serdar and Strome, David. *Require End-to-End Encryption for Sensitive Teams Meetings*. Microsoft. Mar. 27, 2023. URL: https://learn.microsoft.com/en-us/microsoftteams/end-to-end-encrypted-meetings (visited on 04/10/2023).

[5] Slack. *Security at Slack, Enterprise-grade Data Protection*. Slack. URL: https://slack.com/trust/security (visited on 04/10/2023).

[6] Albrecht, M. R., Mareková, L., Paterson, K. G., and Stepanovs, I. "Four attacks and a proof for Telegram". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 87–106.

[7] Backendal, M., Haller, M., and Paterson, K. G. "MEGA: Malleable Encryption Goes Awry". In: *Cryptology ePrint Archive* (2022).

[8] Paterson, K. G., Scarlata, M., and Truong, K. T. *Three Lessons From Threema: Analysis of a Secure Messenger*. Tech. rep. Technical report, 2023.

[9] SHAREKEY Swiss AG. *Privacy Policy*. 2020. URL: https://sharekey.com/info/privacy-policy.pdf.

[10] SHAREKEY Swiss AG. *Security Paper*. 2021. URL: https://sharekey.com/ (visited on 04/10/2023).

[11] SecurityAdvisor. *Top Riskiest Behaviors and Employees in a Hybrid Workplace*. 2021.

[12] Aumasson, J. P. *Sharekey Security Review*. 2022.

[13] Meier, S., Schmidt, B., Cremers, C., and Basin, D. "The TAMARIN prover for the symbolic analysis of security protocols". In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pp. 696–701.

[14] Basin, D., Cremers, C., Dreier, J., Meier, S., Sasse, R., and Schmidt, B. *The Tamarin Prover Repository*. Tamarin prover, Apr. 6, 2023.

[15] Basin, D., Cremers, C., Dreier, J., and Sasse, R. "Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols". In: *IEEE Security & Privacy* 20.3 (May 2022), pp. 24–32. ISSN: 1540-7993, 1558-4046. DOI: 10.1109/MSEC.2022.3154689.

[16] *Internet Engineering Task Force*. IETF. URL: https://www.ietf.org/ (visited on 04/11/2023).

[17] *Internet Research Task Force*. IRTF. URL: https://irtf.org/ (visited on 04/11/2023).

[18] Dowling, B., Fischlin, M., Günther, F., and Stebila, D. "A cryptographic analysis of the TLS 1.3 handshake protocol candidates". In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 2015, pp. 1197–1210.

[19] Davis, H., Diemert, D., Günther, F., and Jager, T. "On the Concrete Security of TLS 1.3 PSK Mode". In: *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part II*. 2022, pp. 876–906.

[20] Davis, H. and Günther, F. "Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols". In: *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II*. 2021, pp. 448–479.

[21] SHAREKEY Swiss AG. *Secure Business Privacy*. URL: https://sharekey.com/ (visited on 04/10/2023).

[22] SHAREKEY Swiss AG. *Sharekey Brochure*. URL: https://sharekey.com/info/sharekey-brochure.pdf?v=2 (visited on 04/10/2023).

[23] *Signal Protocol*. URL: https://signal.org/docs/ (visited on 04/11/2023).

[24] *Clarifying Lawful Overseas Use of Data Act*. 18 U.S.C. § 2713. 2018.

[25] Bernstein, D. J., Lange, T., and Schwabe, P. "The security impact of a new cryptographic library". In: *Progress in Cryptology–LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings 2*. 2012, pp. 159–176.

[26] Bernstein, D. J. "Extending the Salsa20 nonce". In: *Workshop record of Symmetric Key Encryption Workshop*. Vol. 2011. 2011.

[27] Bernstein, D. J. "The Poly1305-AES message-authentication code". In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers 12*. Springer. 2005, pp. 32–49.

[28] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B.-Y. "High-speed high-security signatures". In: *Journal of cryptographic engineering* 2.2 (2012), pp. 77–89.

[29] Percival, C. and Josefsson, S. *The scrypt password-based key derivation function*. 2016.

[30] Moore, R. *Scrypt*. URL: https://github.com/ricmoo/scrypt-js (visited on 04/10/2023).

[31] Rescorla, E. *RFC8446 - Transport Layer Security Protocol*. 2018.

[32] Bernstein, D. J. "Curve25519: new Diffie-Hellman speed records". In: *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer. 2006, pp. 207–228.

[33] Woolf, M. *Big List of Naughty Strings*. URL: https://github.com/minimaxir/big-list-of-naughty-strings/ (visited on 04/11/2023).

[34] *Repository of Security Assessment of the Sharekey Collaboration App*. Cyber-Defence Campus. URL: https://github.com/cyber-defence-campus/sharekey-review.

[35] Dolev, D. and Yao, A. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[36] Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I., and Smith, M. "SoK: secure messaging". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 232–249.

[37] The Tamarin Team. *Tamarin-Prover Manual Security Protocol Analysis in the Symbolic Model*. Tech. rep. 2016.

[38] Stäuble, M. and Paterson, K. *Actually Good Encryption? Confusing Users by Changing Nonces Semester Project*. Tech. rep. 2022.

[39] *Nextcloud*. URL: https://docs.nextcloud.com/ (visited on 04/11/2023).

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):

Security Assessment of the Sharekey Collaboration App

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

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

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)**
Zürich, 2023-04-17	[illegible]

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